AEROSPACE REPORT NO.
ATR-94(4778)-4

A Formal Description of the Incremental Translation of Stage 4 VHDL into State Deltas in SDVS

30 September 1994

Prepared by

I. V. FILIPPENKO
Trusted Computer Systems Department
Computer Science and Technology Subdivision
Computer Systems Division
Engineering and Technology Group

Prepared for

DEPARTMENT OF DEFENSE
Ft. George G. Meade, MD 20744-6000

Engineering and Technology Group

THE AEROSPACE CORPORATION
El Segundo, California

AEROSPACE REPORT NO.
ATR-94(4778)-4

A FORMAL DESCRIPTION OF THE INCREMENTAL TRANSLATION OF STAGE 4 VHDL INTO STATE DELTAS IN SDVS

Prepared by

I. V. FILIPPENKO
Trusted Computer Systems Department
Computer Science and Technology Subdivision
Computer Systems Division
Engineering and Technology Group

30 September 1994

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

Prepared for

DEPARTMENT OF DEFENSE
Ft. George G. Meade, MD 20744-6000

A FORMAL DESCRIPTION OF THE INCREMENTAL TRANSLATION OF STAGE 4 VHDL INTO STATE DELTAS IN SDVS

Prepared by

I. V. Filippenko

I. V. Filippenko

Approved by

L. G. Marcus, Principal Investigator
Computer Assurance Section

D. B. Baker, Director
Trusted Computer Systems Department

Abstract

This report documents a formal semantic specification of Stage 4 VHDL, a subset of the VHSIC Hardware Description Language (VHDL), via translation into the temporal logic of the State Delta Verification System (SDVS). Stage 4 VHDL is the fifth of successively more sophisticated VHDL subsets to be interfaced to SDVS.

The specification is a continuation-style denotational semantics of Stage 4 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 4 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.

Acknowledgments

The author thanks his colleagues in the Computer Assurance Section for their support and contributions to the adaptation of SDVS to VHDL.

Contents

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 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] and [2], while detailed information on the system may be found in [3] and [4].

Several features distinguish SDVS from other verification systems (refer to [5] 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 (VHSIC Hardware Description Language), 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. 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" [6].

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 [7], 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; predefined types **BOOLEAN**, **BIT**, **BIT_VECTOR**, and **INTEGER**; 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 [8].

In fiscal year 1990, SDVS was enhanced to provide the capability of verifying hardware descriptions written in Core VHDL [9, 10]. In fiscal year 1991, the translator underwent extensive revision to accommodate a second VHDL subset, Stage 1 VHDL [11], 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 provided a considerably more complex and capable VHDL language subset. Stage 2 VHDL extended 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, default object class **SIGNAL** for ports, and general expressions of type **TIME** in **AFTER** clauses.

The VHDL language subset implemented in fiscal year 1993, Stage 3 VHDL, extended Stage 2 VHDL with the addition of subtypes of scalar types, integer type definitions, and type conversions between integer types. Furthermore, the capability was added to set "statement marks" (in the form of interpreted comments) for sequential statements. Finally, a facility for specifying, proving, and invoking the behavior of a VHDL subprogram — *VHDL offline characterization* — was implemented [3]. The SDVS VHDL and Ada translators were reengineered to a uniform implementation reflecting language similarities where these exist, and optimized for greater space- and time-efficiency.

Stage 4 VHDL, implemented in fiscal year 1994, comprises a significantly more powerful subset of VHDL than did previous stages, in that Stage 4 VHDL admits the *structural description* of hardware in terms of its hierarchical decomposition into connected subcomponents as outlined in [12]. The previous versions of the SDVS VHDL translator handled only behavioral (e.g., algorithmic or dataflow) hardware descriptions. Thus, Stage 4 VHDL incorporates language constructs such as component declarations, component instantiation statements, **BLOCK** statements, generics, generic maps, port maps, and configuration declarations.

The purpose of the present report is to provide a formal description of the translation of Stage 4 VHDL hardware descriptions into state deltas. This amounts to a formal semantic specification of Stage 4 VHDL, presented herein as a continuation-style denotational seman-

tics [13] for which the state delta language provides the semantic domain. The translation basically consists of parsing followed by two semantic analysis phases.

The first phase receives the abstract syntax tree generated by the Stage 4 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 4 VHDL hardware description with their attributes;
- appropriately disambiguates 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 4 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 4 VHDL formal description is an extensive revision and expansion of the formal specifications of the Core VHDL, Stage 1 VHDL, Stage 2, and Stage 3 VHDL translators [8, 11, 14, 15]. The Stage 4 VHDL translator specification was written in DL, the semantic metalanguage of a denotational semantics specification system called DENOTE [16]. 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 4 VHDL is discussed in Section 2.
- Section 3 contains an overview of the Stage 4 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 4 VHDL.
- Section 6 presents the Stage 4 VHDL static semantics.
- Section 7 presents the interphase abstract syntax tree transformation.
- Section 8 presents the Stage 4 VHDL dynamic semantics in terms of state deltas.
- Finally, some concluding remarks are made in Section 9.

2 History of Our Semantic Approach to VHDL

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 (fiscal years 1989 and 1990), the translator transformed possibly multiple Core VHDL statements: sequential statements between `WAIT` statements within a process were all translated and then *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 *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 (fiscal year 1991), 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 *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 `WAIT`-bracketed code in `PROCESS` statements would be tantamount to the automatic construction of correctness proofs without user intervention — a theoretically undecidable problem.

Therefore, we abandoned composition for Stage 1 VHDL and subsequent 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 *resolved signal*), and the ultimate signal value is obtained by application of a user-defined *resolution function*.[1]

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 *VHDL Language Reference Manual* [6]), but with the crucial aspect of symbolic execution (use of abstract values rather than concrete) thrown in. The current Stage 4 VHDL translator thus functions as a "symbolic simulator," with the effect of being reasonably intuitive as a proof engine.

[1] As of Stage 4 VHDL, however, *resolved signals* are still disallowed.

3 Overview of Stage 4 VHDL

The primary VHDL abstraction for modeling a digital device is the *design entity*. A design entity consists of two parts: an *entity declaration* and an *architecture body*. The entity declaration provides the "external view" of the device: it defines the interface between the design entity and its environment, including the number, direction, and type of ports, and corresponds to a *symbol* in a traditional CAE (Computer-Aided Engineering) design methodology. The architecture body provides the "internal view" of the device, describing its behavior or structure, and thereby expressing the relationship between its inputs and outputs. A given entity declaration may be shared by several design entities, each with a different architecture body.

In Stage 4 VHDL, each architecture body consists of a set of *declarations* and *concurrent statements* defining the structure or function of the device being modeled. The allowable concurrent statements are of four kinds, to be discussed below: `PROCESS` statements, *concurrent signal assignment* statements (conditional and selected), `BLOCK` statements, and *component instantiation statements*.

The special case of a structural architecture, in particular, corresponds to the CAE notion of a *schematic*. A structural architecture for a design entity is described by declaring internal signals and connecting these, as well as the ports of the entity declaration, to the ports of various subcomponents declared in *component declarations* and created by component instantiation statements in the architecture body.

Component declarations provide a "template" mechanism, whereby an architecture body containing component instantiations can be *analyzed* — checked for syntactic and semantic correctness — independently of prior analysis of entity declarations for those components. This is accomplished by having the instantiations refer not to entity declarations, but to component declarations.

The *configuration declaration* provides the mechanism whereby architecture bodies are paired with entity declarations to configure specific design entities. A configuration declaration is analogous to a "parts list," describing which part to use for each component of a design. (The *configuration specification*, an essentially equivalent alternative, is not supported by Stage 4 VHDL.)

A component instantiation statement specifies an instance of a *child component* occurring inside a *parent component*. At the point of instantiation, only the external view of the child component — the names, types, and directions of its ports — is visible; the child component's internal signals are not visible. The component instantiation statement identifies the child component and specifies which ports or signals in the parent component are connected to which ports in the child component. Component instantiation statements are transformed, in a manner prescribed by the VHDL LRM [6], to pairs of nested `BLOCK` statements during the elaboration of a VHDL design entity prior to its execution. A `BLOCK` statement provides a block-structured scope with local declarations and a body consisting of concurrent statements. Elaboration of a design entity recursively transforms component instantiation statements occuring in `BLOCK` statements until the innermost blocks contain only `PROCESS` and concurrent signal assignment statements.

A PROCESS statement, the most fundamental kind of concurrent statement in VHDL, is a block of sequential *zero-time statements* that execute sequentially but "instantaneously" in *zero time* [17], 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 4 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.

Concurrent signal assignment statements 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 before translating them 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 4 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.

Currently Stage 4 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. Resolved signals and their resolution functions

will be implemented in a future version of SDVS.

The Stage 4 VHDL data types are: BOOLEAN, BIT, UNIVERSAL_INTEGER, 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*, user-defined *array types*, *subtypes* of scalar types, and *integer type definitions*. Furthermore, explicit *type conversions* between integer types are allowed. The preliminary implementation allows VHDL descriptions involving type REAL to be parsed and translated, but provides no support for reasoning about floating point numbers.

Concrete and abstract syntaxes for Stage 4 VHDL have been defined — see Section 5 — as required, of course, for the implementation of the Stage 4 VHDL translator. The following is a convenient synopsis of the Stage 4 VHDL language subset.

- VHDL design files
  - design units
- 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*: implementation of type REAL is preliminary
- user-defined packages
  - package declarations
  - package bodies
- USE clauses for accessing packages
- entity declarations
  - entity header: generics, port declarations
  - entity declarative part: other declarations
- architecture bodies
- configuration declarations
  - generic maps, port maps
- object declarations
  - CONSTANT, VARIABLE, SIGNAL
  - octal and hexadecimal representations of bitstrings
  - entity ports of default object class SIGNAL
- array type declarations
  - arrays (bidirectional; constrained or not) of arbitrary element type

  - attributes `'low` and `'high` for lower and upper bounds of an array type (*restriction*: but not of an object of type array)

- user-defined enumeration types
- subtypes of scalar types
- integer type definitions
- type conversion
- signals of arbitrary 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
  - `BLOCK` statements
  - component instantiation statements
- sequential statements
  - null statement: `NULL`
  - variable assignments (scalar and composite)
  - signal assignments (scalar and 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 4 VHDL operators defined for 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 [13]; and
- the theory and practice of state deltas [3, 18, 19].

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

$$\mathbf{Env} = \mathbf{Id} \rightarrow (\mathbf{Dv} + \mathbf{*UNBOUND*})$$

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)** = ***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

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

where (a → 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 4 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$_1$, ..., elbl$_n$))**, **t((elbl$_1$, ..., elbl$_n$))** is the most *local* set of bindings, ..., and **t(ϵ)** is the most *global* set of bindings, where ϵ 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$, ..., elbl$_n$)**, then **$(p)(id)** is represented as **elbl$_1$.elbl$_2$. ... .elbl$_n$.id**. When **p** = ϵ (empty path), **$($\epsilon$)(id)** is simply represented by **id**. **$** is defined by

$(path)(id) = (path = ϵ → 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 4 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 4 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 semantic 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 4 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** = ⟨**qid**, **tag**, ... ⟩, then **d'** = ⟨**id**, **qid**, **tag**, ... ⟩.

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

$$\textbf{extend(t)(p)(elbl)}$$

denotes the TSE that is the same as **t** except that

$$\textbf{(extend(t)(p)(elbl))(\%(p)(elbl))} = \textbf{r0}.$$

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 (**goto**s, 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 4 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 4 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 [13]) is used in this report, except that structured arguments are permitted in lambda-abstractions. Lambda-abstractions normally have the form λ**x.body**, where **body** is a lambda-term and **x** may be free in **body**. The term λ**x.**λ**y.body** is printed as λ**x,y.body**. If **x** is, for example, a *pair*, then the components of **x** can be represented in **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 **body** (instead of projection functions applied to **x**) by using the abstraction λ**(y,z).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 **cons(x)(y)** and **append(x)(y)**. If **x** is a *nonempty* sequence (list), then **hd(x)** denotes its first element and **tl(x)** the sequence (list) of its remaining components; **x** = **cons(hd(x))(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) = hd(tl(x))

third(x) = hd(tl(tl(x)))

fourth(x) = hd(tl(tl(tl(x))))

fifth(x) = hd(tl(tl(tl(tl(x)))))

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

$\text{last}(\text{id}^+) = (\text{null}(\text{tl}(\text{id}^+)) \rightarrow \text{hd}(\text{id}^+), \text{last}(\text{tl}(\text{id}^+)))$

$\text{rest}(\text{id}^+) = (\text{null}(\text{tl}(\text{id}^+)) \rightarrow \epsilon, \text{cons}(\text{hd}(\text{id}^+), \text{rest}(\text{tl}(\text{id}^+))))$

$\text{length}(x) = (\text{null}(x) \rightarrow 0, 1 + \text{length}(\text{tl}(x)))$

5 Syntax of Stage 4 VHDL

Three Stage 4 VHDL syntaxes are used by the translator: a *concrete syntax*, which is SLR(1) and is used for parsing Stage 4 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 4 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 4 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 4 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 syntactic representation is accomplished by semantic action routines in the Stage 4 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 4 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 4 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 4 VHDL.

5.1 Syntactic Domains

Primitive Syntactic Domains

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

Compound Syntactic Domains

`design-file` : **Design**	design files
`design-unit` : **DUnit**	design units
`context-item` : **CItem**	context items
`library-unit` : **LUnit**	library units
`configuration-decl` : **Config**	configuration declarations
`block-config` : **BConf**	block configurations
`component-config` : **CConf**	component configurations
`binding-indication` : **Bind**	binding indications
`entity-decl` : **Ent**	entity declarations
`architecture-body` : **Arch**	architecture bodies
`generic-decl` : **GDec**	generic declarations
`port-decl` : **PDec**	port declarations
`generic-map-aspect` : **GMap**	generic map aspects
`port-map-aspect` : **PMap**	port map aspects
`decl, package-decl, package-body, use-clause, component-decl` : **Dec**	declarations
`conc-stat` : **CStat**	concurrent statements
`seq-stat` : **SStat**	sequential statements
`case-alt` : **Alt**	case alternatives
`discrete-range` : **Drg**	discrete ranges
`waveform` : **Wave**	waveforms
`transaction` : **Trans**	transactions
`expr` : **Expr**	expressions
`ref` : **Ref**	references
`unary-op` : **Uop**	unary operators
`binary-op` : **Bop**	binary operators
`relational-op` : **Bop**	relational operators

5.2 Syntax Equations

In Sections 5.2.1, 5.2.2, and 5.2.3 we present, respectively, the concrete syntax for Stage 4 VHDL hardware descriptions admissible as input to the SDVS VHDL language parser,

the syntax of VHDL abstract parse trees generated by the parser for use by Phase 1 of the VHDL translator, and the syntax of transformed parse trees produced during Phase 1 for use by translator Phase 2.

5.2.1 Concrete Syntax

The concrete syntax for Stage 4 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 uppercase letters, while nonterminal symbols and pseudo-terminals (terminals denoting a set of values) are in lowercase; pseudo-terminals are prefixed by a "dot" (.).

STAGE 4 VHDL CONCRETE SYNTAX

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

 2  design-file
       ::=  init  design-unit-list

 3  init
       ::=

 4  design-unit-list
       ::=  design-unit
 5       |  design-unit-list  design-unit

 6  design-unit
       ::=  context-item-list  library-unit

 7  context-item-list
       ::=
 8       |  context-item-list  context-item

 9  context-item
       ::=  use-clause

10  library-unit
       ::=  primary-unit
11       |  secondary-unit

12  primary-unit
       ::=  configuration-decl
13       |  package-decl
14       |  entity-decl

15  secondary-unit
       ::=  package-body
16       |  architecture-body
```

```
17  configuration-decl
      ::= CONFIGURATION .id OF .id IS config-decl-part
          block-config END opt-id ;

18  config-decl-part
      ::= config-decl-item-list

19  config-decl-item-list
      ::=
20      | config-decl-item-list config-decl-item

21  config-decl-item
      ::= use-clause

22  block-config
      ::= FOR block-spec use-clause-list config-item-list END
          FOR ;

23  block-spec
      ::= .id

24  config-item-list
      ::=
25      | config-item-list config-item

26  config-item
      ::= block-config
27      | component-config

28  component-config
      ::= FOR component-spec USE binding-indication ;
          block-config END FOR ;
29      | FOR component-spec USE binding-indication ; END FOR
          ;
30      | FOR component-spec END FOR ;

31  component-spec
      ::= instantiation-list : dotted-name

32  instantiation-list
      ::= id-list
33      | all
34      | others

35  binding-indication
      ::= entity-aspect opt-generic-map-aspect opt-port-map-aspect

36  entity-aspect
      ::= ENTITY dotted-name ( .id )
37      | ENTITY dotted-name
38      | CONFIGURATION dotted-name

39  package-decl-list
      ::=
40      | package-decl-list package-decl

41  package-decl
```

```
      ::= PACKAGE .id IS package-decl-part END opt-id ;

42  package-decl-part
      ::= package-decl-item-list

43  package-decl-item-list
      ::=
44      | package-decl-item-list package-decl-item

45  package-decl-item
      ::= const-decl
46      | sig-decl
47      | type-decl
48      | subtype-decl
49      | subprog-decl
50      | use-clause

51  opt-id
      ::=
52      | .id

53  package-body-list
      ::=
54      | package-body package-body-list

55  package-body
      ::= PACKAGE BODY .id IS package-body-decl-part END
          opt-id ;

56  package-body-decl-part
      ::= package-body-decl-item-list

57  package-body-decl-item-list
      ::=
58      | package-body-decl-item-list package-body-decl-item

59  package-body-decl-item
      ::= const-decl
60      | type-decl
61      | subtype-decl
62      | subprog-decl
63      | subprog-body
64      | use-clause

65  use-clause-list
      ::=
66      | use-clause-list use-clause

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

68  dotted-name-list
      ::= dotted-name
69      | dotted-name-list , dotted-name

70  dotted-name
      ::= .id
```

```
71        | dotted-name  .  .id
72        | dotted-name  .  all

73  all
        ::=  ALL

74  others
        ::=  OTHERS

75  entity-decl
        ::=  ENTITY  .id  IS  opt-generic-clause  opt-port-clause  END
             opt-id  ;
76        | ENTITY  .id  IS  opt-generic-clause  opt-port-clause
             ent-decl-part  END  opt-id  ;

77  opt-generic-clause
        ::=
78        | generic-clause

79  opt-port-clause
        ::=
80        | port-clause

81  generic-clause
        ::=  GENERIC  (  generic-list  )  ;

82  generic-list
        ::=  generic-decl
83        | generic-list  ;  generic-decl

84  generic-decl
        ::=  opt-constant  id-list  :  opt-in  type-mark  opt-init
85        | opt-constant  id-list  :  opt-in  slice-name  opt-init

86  opt-constant
        ::=
87        | CONSTANT

88  opt-in
        ::=
89        | IN

90  ent-decl-part
        ::=  ent-decl-item-list

91  ent-decl-item-list
        ::=  ent-decl-item
92        | ent-decl-item-list  ent-decl-item

93  ent-decl-item
        ::=  const-decl
94        | sig-decl
95        | type-decl
96        | subtype-decl
97        | subprog-decl
98        | subprog-body
99        | use-clause
```

```
100  architecture-body
        ::=  ARCHITECTURE  .id  OF  .id  IS  arch-decl-part  BEGIN
             arch-stat-part  END  opt-id  ;

101  arch-decl-part
        ::=  arch-decl-item-list

102  arch-decl-item-list
        ::=
103       |  arch-decl-item-list  arch-decl-item

104  arch-decl-item
        ::=  const-decl
105       |  sig-decl
106       |  type-decl
107       |  subtype-decl
108       |  subprog-decl
109       |  subprog-body
110       |  use-clause
111       |  component-decl

112  arch-stat-part
        ::=  conc-stats

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

114  port-list
        ::=  port-decl
115       |  port-list  ;  port-decl

116  port-decl
        ::=  opt-signal  id-list  :  opt-mode  type-mark  opt-init
117       |  opt-signal  id-list  :  opt-mode  slice-name  opt-init

118  opt-signal
        ::=
119       |  SIGNAL

120  id-list
        ::=  .id
121       |  id-list  ,  .id

122  opt-mode
        ::=
123       |  mode

124  mode
        ::=  IN
125       |  OUT
126       |  INOUT
127       |  BUFFER

128  type-mark
        ::=  dotted-name
```

```
129  slice-name
       ::= type-mark ( discrete-range )

130  discrete-range
       ::= range

131  range
       ::= simple-expr direction simple-expr

132  direction
       ::= TO
133      | DOWNTO

134  opt-init
       ::=
135      | := expr

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

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

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

142  type-decl
       ::= enum-type-decl
143      | array-type-decl
144      | integer-type-decl

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

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

148  char-list
       ::= character-literal
149      | char-list , character-literal

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

151  array-type-def
       ::= ARRAY ( discrete-range ) OF type-mark

152  integer-type-decl
       ::= TYPE .id IS RANGE discrete-range ;

153  subtype-decl
       ::= SUBTYPE .id IS type-mark opt-constraint ;
```

```
154  opt-constraint
        ::=
155       | constraint

156  constraint
        ::= range-constraint

157  range-constraint
        ::= RANGE discrete-range

158  component-decl
        ::= COMPONENT .id opt-generic-clause opt-port-clause END
            COMPONENT ;

159  subprog-decl
        ::= subprog-spec ;

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

162  opt-procedure-formal-part
        ::=
163       | ( procedure-par-spec-list )

164  opt-function-formal-part
        ::=
165       | ( function-par-spec-list )

166  procedure-par-spec-list
        ::= procedure-par-spec
167       | procedure-par-spec-list ; procedure-par-spec

168  function-par-spec-list
        ::= function-par-spec
169       | function-par-spec-list ; function-par-spec

170  procedure-par-spec
        ::= proc-object-class id-list : procedure-par-mode
            type-mark opt-expr
171       | id-list : IN type-mark opt-expr
172       | id-list : OUT type-mark opt-expr
173       | id-list : INOUT type-mark opt-expr

174  function-par-spec
        ::= fn-object-class id-list : function-par-mode type-mark
            opt-expr

175  proc-object-class
        ::= CONSTANT
176       | VARIABLE

177  fn-object-class
        ::=
178       | CONSTANT

179  procedure-par-mode
```

```
       ::=
180      | IN
181      | OUT
182      | INOUT

183  function-par-mode
       ::=
184      | IN

185  subprog-body
       ::= subprog-spec IS subprog-decl-part BEGIN
           subprog-stat-part END opt-id ;

186  subprog-decl-part
       ::= subprog-decl-item-list

187  subprog-decl-item-list
       ::=
188      | subprog-decl-item-list subprog-decl-item

189  subprog-decl-item
       ::= const-decl
190      | var-decl
191      | type-decl
192      | subtype-decl
193      | subprog-decl
194      | subprog-body
195      | use-clause

196  subprog-stat-part
       ::= seq-stats

197  conc-stats
       ::=
198      | conc-stats conc-stat

199  conc-stat
       ::= block-stat
200      | process-stat
201      | concurrent-sig-assn-stat
202      | component-instantiation-stat

203  block-stat
       ::= unit-label BLOCK block-header BEGIN block-stat-part
           END BLOCK opt-id ;
204      | unit-label BLOCK block-header block-decl-part BEGIN
           block-stat-part END BLOCK opt-id ;

205  block-header
       ::=
206      | generic-part
207      | port-part
208      | generic-part port-part

209  generic-part
       ::= generic-clause
210      | generic-clause generic-map-aspect ;
```

```
211  port-part
        ::=  port-clause
212       |  port-clause  port-map-aspect  ;

213  block-decl-part
        ::=  block-decl-item-list

214  block-decl-item-list
        ::=  block-decl-item
215       |  block-decl-item-list  block-decl-item

216  block-decl-item
        ::=  const-decl
217       |  sig-decl
218       |  type-decl
219       |  subtype-decl
220       |  subprog-decl
221       |  subprog-body
222       |  use-clause
223       |  component-decl

224  block-stat-part
        ::=  conc-stats

225  process-stat
        ::=  opt-unit-label  PROCESS  process-decl-part  BEGIN
             process-stat-part  END  PROCESS  opt-id  ;
226       |  opt-unit-label  PROCESS  (  sensitivity-list  )
             process-decl-part  BEGIN  process-stat-part  END  PROCESS
             opt-id  ;

227  opt-unit-label
        ::=
228       |  unit-label

229  unit-label
        ::=  .id  :

230  process-decl-part
        ::=  process-decl-item-list

231  process-decl-item-list
        ::=
232       |  process-decl-item-list  process-decl-item

233  process-decl-item
        ::=  const-decl
234       |  var-decl
235       |  type-decl
236       |  subtype-decl
237       |  subprog-decl
238       |  subprog-body
239       |  use-clause

240  process-stat-part
        ::=  seq-stats
```

```
241  concurrent-sig-assn-stat
        ::=  selected-sig-assn-stat
242       |  conditional-sig-assn-stat

243  selected-sig-assn-stat
        ::=  opt-unit-label  WITH  expr  SELECT  target  <=
             opt-transport  selected-waveforms  ;
244       |  .atmark  opt-unit-label  WITH  expr  SELECT  target  <=
             opt-transport  selected-waveforms  ;

245  opt-transport
        ::=
246       |  transport

247  transport
        ::=  TRANSPORT

248  selected-waveforms
        ::=  selected-waveform
249       |  selected-waveforms  ,  selected-waveform

250  selected-waveform
        ::=  waveform  WHEN  choices

251  conditional-sig-assn-stat
        ::=  target  <=  opt-transport  conditional-waveforms  waveform
             ;
252       |  .atmark  target  <=  opt-transport  conditional-waveforms
             waveform  ;
253       |  .id  :  target  <=  opt-transport  conditional-waveforms
             waveform  ;
254       |  .atmark  .id  :  target  <=  opt-transport
             conditional-waveforms  waveform  ;

255  conditional-waveforms
        ::=
256       |  conditional-waveforms  conditional-waveform

257  conditional-waveform
        ::=  waveform  WHEN  expr  ELSE

258  waveform
        ::=  waveform-elt-list

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

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

263  component-instantiation-stat
        ::=  .id  :  name  opt-generic-map-aspect  opt-port-map-aspect
             ;
```

```
264  opt-generic-map-aspect
        ::=
265       | generic-map-aspect

266  generic-map-aspect
        ::= GENERIC MAP ( assoc-list )

267  opt-port-map-aspect
        ::=
268       | port-map-aspect

269  port-map-aspect
        ::= PORT MAP ( assoc-list )

270  assoc-list
        ::= assoc-elt
271       | assoc-list , assoc-elt

272  assoc-elt
        ::= formal-part => actual-part

273  formal-part
        ::= formal-designator

274  formal-designator
        ::= name

275  actual-part
        ::= actual-designator

276  actual-designator
        ::= expr

277  seq-stats
        ::=
278       | seq-stats seq-stat

279  seq-stat
        ::= null-stat
280       | var-assn-stat
281       | sig-assn-stat
282       | if-stat
283       | case-stat
284       | loop-stat
285       | exit-stat
286       | return-stat
287       | proc-call-stat
288       | wait-stat

289  null-stat
        ::= NULL ;
290       | .atmark NULL

291  var-assn-stat
        ::= name := expr ;
292       | .atmark name := expr ;
```

```
293  sig-assn-stat
        ::= target <= opt-transport waveform ;
294       | .atmark target <= opt-transport waveform ;

295  if-stat
        ::= if-head if-tail
296       | .atmark if-head if-tail

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

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

301  case-stat
        ::= CASE expr IS case-alt-list END CASE ;
302       | .atmark CASE expr IS case-alt-list END CASE ;

303  case-alt-list
        ::= case-alt
304       | case-other-alt
305       | case-alt case-alt-list

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

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

308  choices
        ::= choice
309       | choices | choice

310  choice
        ::= simple-expr
311       | discrete-range

312  loop-stat
        ::= simple-loop
313       | while-loop
314       | for-loop

315  simple-loop
        ::= opt-unit-label LOOP seq-stats END LOOP opt-id ;
316       | .atmark opt-unit-label LOOP seq-stats END LOOP
            opt-id ;

317  while-loop
        ::= opt-unit-label WHILE expr LOOP seq-stats END LOOP
            opt-id ;
318       | .atmark opt-unit-label WHILE expr LOOP seq-stats END
            LOOP opt-id ;

319  for-loop
        ::= opt-unit-label FOR name IN discrete-range LOOP
```

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

321   exit-stat
        ::= EXIT opt-dotted-name opt-when-cond ;
322       | .atmark EXIT opt-dotted-name opt-when-cond ;

323   opt-dotted-name
        ::=
324       | dotted-name

325   opt-when-cond
        ::=
326       | WHEN expr

327   proc-call-stat
        ::= name ;
328       | .atmark name ;

329   return-stat
        ::= RETURN ;
330       | .atmark RETURN ;
331       | RETURN expr ;
332       | .atmark RETURN expr ;

333   wait-stat
        ::= WAIT opt-sensitivity-clause opt-condition-clause
            opt-timeout-clause ;
334       | .atmark WAIT opt-sensitivity-clause
            opt-condition-clause opt-timeout-clause ;

335   opt-sensitivity-clause
        ::=
336       | sensitivity-clause

337   sensitivity-clause
        ::= ON sensitivity-list

338   sensitivity-list
        ::= name-list

339   name-list
        ::= name
340       | name-list , name

341   opt-condition-clause
        ::=
342       | condition-clause

343   condition-clause
        ::= UNTIL expr

344   opt-timeout-clause
        ::=
345       | timeout-clause
```

```
346  timeout-clause
        ::=  FOR  expr

347  expr-list
        ::=  expr
348       |  expr-list  ,  expr

349  opt-expr
        ::=
350       |  expr

351  expr
        ::=  rel
352       |  rel  and-expr
353       |  rel  nand-expr
354       |  rel  or-expr
355       |  rel  nor-expr
356       |  rel  xor-expr

357  rel
        ::=  simple-expr
358       |  simple-expr  relop  simple-expr

359  and-expr
        ::=  and-part
360       |  and-part  and-expr

361  and-part
        ::=  AND  rel

362  nand-expr
        ::=  nand-part
363       |  nand-part  nand-expr

364  nand-part
        ::=  NAND  rel

365  or-expr
        ::=  or-part
366       |  or-part  or-expr

367  or-part
        ::=  OR  rel

368  nor-expr
        ::=  nor-part
369       |  nor-part  nor-expr

370  nor-part
        ::=  NOR  rel

371  xor-expr
        ::=  xor-part
372       |  xor-part  xor-expr

373  xor-part
        ::=  XOR  rel
```

```
374  simple-expr
        ::=  simple-expr1
375       |  +  simple-expr1
376       |  -  simple-expr1

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

379  term
        ::=  factor
380       |  term  mulop  factor

381  factor
        ::=  primary
382       |  primary  **  primary
383       |  ABS  primary
384       |  NOT  primary

385  primary
        ::=  primary1
386       |  aggregate
387       |  (  expr  )

388  primary1
        ::=  literal
389       |  .atmark
390       |  name

391  literal
        ::=  boolean-literal
392       |  bit-literal
393       |  character-literal
394       |  numeric-literal
395       |  time-literal
396       |  bitstring-literal
397       |  string-literal

398  boolean-literal
        ::=  FALSE
399       |  TRUE

400  bit-literal
        ::=  .bit

401  character-literal
        ::=  .char

402  numeric-literal
        ::=  .constant

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

404  opt-time-constant
        ::=
```

```
405       |  .constant

406   time-unit
          ::=  FS
407         |  PS
408         |  NS
409         |  US
410         |  MS
411         |  SEC
412         |  MIN
413         |  HR

414   bitstring-literal
          ::=  .bitstring
415         |  .octstring
416         |  .hexstring

417   string-literal
          ::=  .string

418   aggregate
          ::=  (  2-expr-list  )

419   2-expr-list
          ::=  expr  ,  expr
420         |  2-expr-list  ,  expr

421   target
          ::=  name

422   name
          ::=  name1

423   name1
          ::=  selector
424         |  name1  .  selector
425         |  name1  (  expr-list  )

426   selector
          ::=  .id

427   relop
          ::=  =
428         |  /=
429         |  <
430         |  <=
431         |  >
432         |  >=

433   addop
          ::=  +
434         |  -
435         |  &

436   mulop
          ::=  *
437         |  /
```

```
438         | MOD
439         | REM
```

5.2.2 Abstract Syntax: Phase 1

The abstract syntax of Stage 4 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 4 VHDL ABSTRACT SYNTAX: PHASE 1

design-file ::= **DESIGN-FILE** id design-unit$^+$
design-unit ::= **DESIGN-UNIT** context-item* library-unit
context-item ::= use-clause
use-clause ::= **USE** dotted-name$^+$
library-unit ::= primary-unit | secondary-unit
primary-unit ::= configuration-decl | package-decl | entity-decl
secondary-unit ::= package-body | architecture-body
configuration-decl ::= **CONFIGURATION** id_1 id_2 use-clause* block-config opt-id
package-decl ::= **PACKAGE** id decl* opt-id
entity-decl ::= **ENTITY** id generic-decl* port-decl* decl* opt-id
package-body ::= **PACKAGEBODY** id decl* opt-id
architecture-body ::= **ARCHITECTURE** id_1 id_2 decl* conc-stat* opt-id
opt-block-config ::= ϵ | block-config
block-config ::= **BLOCK-CONFIG** id use-clause* component-config*
component-config ::= **COMP-CONFIG** component-spec opt-binding-indication opt-block-config
component-spec ::= id$^+$ dotted-name
opt-binding-indication ::= ϵ | binding-indication
binding-indication ::= **BIND** entity-aspect opt-generic-map-aspect opt-port-map-aspect
entity-aspect ::= **BOUND-ENTITY** dotted-name opt-id |
BOUND-CONFIGURATION dotted-name
opt-generic-map-aspect ::= ϵ | generic-map-aspect
generic-map-aspect ::= **GENERICMAP** assoc-elt$^+$
opt-port-map-aspect ::= ϵ | port-map-aspect
port-map-aspect ::= **PORTMAP** assoc-elt$^+$
generic-decl ::= **DEC GENERIC** id$^+$ type-mark opt-expr |
SLCDEC GENERIC id$^+$ slice-name opt-expr
port-decl ::= **DEC PORT** id$^+$ mode type-mark opt-expr |
SLCDEC PORT id$^+$ mode slice-name opt-expr
mode ::= **IN** | **OUT** | **INOUT** | **BUFFER**
atmark ::= **AT** id

type-mark ::= dotted-name
dotted-name ::= id$^+$
slice-name ::= type-mark discrete-range
discrete-range ::= direction expr$_1$ expr$_2$
direction ::= **TO** | **DOWNTO**
decl ::= object-decl | type-decl | subtype-decl | package-decl |
package-body | subprog-decl | subprog-body | use-clause |
component-decl
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 ::= **ETDEC** id id$^+$ | **ATDEC** id discrete-range type-mark |
ITDEC id discrete-range
subtype-decl ::= **STDEC** id type-mark opt-discrete-range
subprog-decl ::= subprog-spec
subprog-spec ::= **PROCEDURE** id proc-par-spec* |
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**
subprog-body ::= **SUBPROGBODY** subprog-spec decl* seq-stat* opt-id
component-decl ::= **COMPONENT** id generic-decl* port-decl*
conc-stat ::= block-stat | process-stat | selected-sig-assn-stat |
conditional-sig-assn-stat | component-instantiation-stat
block-stat ::= **BLOCK** id block-header decl* conc-stat* opt-id
block-header ::= generic-part port-part
generic-part ::= generic-decl* generic-map-aspect
port-part ::= port-decl* port-map-aspect
process-stat ::= **PROCESS** id ref* decl* seq-stat* opt-id
selected-sig-assn-stat ::= **SEL-SIGASSN** atmark delay-type id expr ref selected-waveform$^+$
selected-waveform ::= **SEL-WAVE** waveform discrete-range$^+$
conditional-sig-assn-stat ::= **COND-SIGASSN** atmark delay-type id ref cond-waveform* waveform
cond-waveform ::= **COND-WAVE** waveform expr
component-instantiation-stat ::= **COMPINST** id ref opt-generic-map-aspect opt-port-map-aspect
assoc-elt ::= ref expr
seq-stat ::= null-stat | var-assn-stat | sig-assn-stat | if-stat | case-stat |
loop-stat | while-stat | for-stat | exit-stat | call-stat |
return-stat | wait-stat
null-stat ::= **NULL** atmark
var-assn-stat ::= **VARASSN** atmark ref expr
sig-assn-stat ::= **SIGASSN** atmark delay-type ref waveform
delay-type ::= **INERTIAL** | **TRANSPORT**
waveform ::= **WAVE** transaction$^+$
transaction ::= **TRANS** expr opt-expr
if-stat ::= **IF** atmark cond-part$^+$ else-part

cond-part ::= expr seq-stat*
else-part ::= seq-stat*
case-stat ::= CASE atmark expr case-alt+
case-alt ::= CASECHOICE discrete-range+ seq-stat* |
CASEOTHERS seq-stat*
loop-stat ::= LOOP atmark id seq-stat* opt-id
while-stat ::= WHILE atmark id expr seq-stat* opt-id
for-stat ::= FOR atmark id ref discrete-range seq-stat* opt-id
exit-stat ::= EXIT atmark opt-dotted-name opt-expr
call-stat ::= CALL atmark ref
return-stat ::= RETURN atmark opt-expr
wait-stat ::= WAIT atmark ref* opt-expr$_1$ opt-expr$_2$
expr ::= ϵ | bool-lit | bit-lit | num-lit | time-lit | char-lit |
bitstr-lit | str-lit | ref | positional-aggregate | unary-op expr |
binary-op expr$_1$ expr$_2$ | relational-op expr$_1$ expr$_2$
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
name ::= 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-discrete-range ::= ϵ | discrete-range
opt-dotted-name ::= ϵ | dotted-name
opt-expr ::= ϵ | expr

5.2.3 Abstract Syntax: Phase 2

The abstract syntax of Stage 4 VHDL used during Phase 2 translation differs in certain respects from that employed by Phase 1. An abstract syntax transformation is performed at the very end of Phase 1, and just prior to the invocation of Phase 2, as described in Section 7.

The most significant transformations of Phase 1 syntax to that of Phase 2 are: (1) the "desugaring" (i.e., reduction to more basic constructs) of concurrent signal assignment statements (conditional signal assignment and selected signal assignment) into equivalent

PROCESS statements; (2) the desugaring of component instantiation statements into equivalent pairs of nested BLOCK statements, and (3) the disambiguation of REFs into simple references, array references, record field accesses (not fully supported by Stage 4 VHDL), and subprogram calls.

STAGE 4 VHDL ABSTRACT SYNTAX: PHASE 2

design-file ::= **DESIGN-FILE** id design-unit$^+$
design-unit ::= **DESIGN-UNIT** context-item* library-unit
context-item ::= use-clause
use-clause ::= **USE** dotted-name$^+$
library-unit ::= primary-unit | secondary-unit
primary-unit ::= configuration-decl | package-decl | entity-decl
secondary-unit ::= package-body | architecture-body
configuration-decl ::= **CONFIGURATION** id_1 id_2 use-clause* block-config opt-id
package-decl ::= **PACKAGE** id decl* opt-id
entity-decl ::= **ENTITY** id $decl*_1$ $decl*_2$ $decl*_3$ opt-id phase1-hook
package-body ::= **PACKAGEBODY** id decl* opt-id
architecture-body ::= **ARCHITECTURE** id_1 id_2 decl* conc-stat* opt-id
opt-block-config ::= ϵ | block-config
block-config ::= **BLOCK-CONFIG** id use-clause* component-config*
component-config ::= **COMP-CONFIG** component-spec opt-binding-indication opt-block-config
component-spec ::= id$^+$ dotted-name
opt-binding-indication ::= ϵ | binding-indication
binding-indication ::= **BIND** entity-aspect opt-generic-map-aspect opt-port-map-aspect
entity-aspect ::= **BOUND-ENTITY** dotted-name opt-id |
BOUND-CONFIGURATION dotted-name
opt-generic-map-aspect ::= ϵ | generic-map-aspect
generic-map-aspect ::= **GENERICMAP** assoc-elt$^+$
opt-port-map-aspect ::= ϵ | port-map-aspect
port-map-aspect ::= **PORTMAP** assoc-elt$^+$
assoc-elt ::= ref expr
decl ::= object-decl | type-decl | subtype-decl | package-decl | package-body |
subprog-decl | subprog-body | use-clause | component-decl
object-decl ::= **DEC** object-class id$^+$ type-mark opt-expr |
SLCDEC object-class id$^+$ slice-name opt-expr
object-class ::= **CONST** | **VAR** | **SIG**
type-mark ::= dotted-name
dotted-name ::= id$^+$
slice-name ::= type-mark discrete-range
discrete-range ::= direction $expr_1$ $expr_2$
direction ::= **TO** | **DOWNTO**
type-decl ::= **ETDEC** id id$^+$ | **ATDEC** id discrete-range type-mark |
ITDEC id discrete-range

subtype-decl ::= **STDEC** id type-mark opt-discrete-range
subprog-decl ::= subprog-spec
subprog-spec ::= **PROCEDURE** id proc-par-spec* |
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**
subprog-body ::= **SUBPROGBODY** subprog-spec decl* seq-stat* opt-id
component-decl ::= **COMPONENT** id $decl^*_1$ $decl^*_2$ phase1-hook
conc-stat ::= block-stat | process-stat
block-stat ::= **BLOCK** id decl* conc-stat* opt-id phase1-hook
process-stat ::= **PROCESS** id decl* seq-stat* opt-id phase1-hook
seq-stat ::= null-stat | var-assn-stat | sig-assn-stat | if-stat | case-stat |
loop-stat | while-stat | for-stat | exit-stat | call-stat |
return-stat | wait-stat
atmark ::= **AT** id
null-stat ::= **NULL** atmark
var-assn-stat ::= **VARASSN** atmark ref expr
sig-assn-stat ::= **SIGASSN** atmark delay-type ref waveform
delay-type ::= **INERTIAL** | **TRANSPORT**
waveform ::= **WAVE** $transaction^+$
transaction ::= **TRANS** expr opt-expr
if-stat ::= **IF** atmark $cond\text{-}part^+$ else-part
cond-part ::= expr seq-stat*
else-part ::= seq-stat*
case-stat ::= **CASE** atmark expr $case\text{-}alt^+$
case-alt ::= **CASECHOICE** $discrete\text{-}range^+$ seq-stat* |
CASEOTHERS seq-stat*
loop-stat ::= **LOOP** atmark id seq-stat* opt-id
while-stat ::= **WHILE** atmark id expr seq-stat* opt-id
for-stat ::= **FOR** atmark id ref discrete-range seq-stat* opt-id
exit-stat ::= **EXIT** atmark opt-dotted-name opt-expr
call-stat ::= **CALL** atmark ref
return-stat ::= **RETURN** atmark opt-expr
wait-stat ::= **WAIT** atmark ref* $opt\text{-}expr_1$ $opt\text{-}expr_2$
expr ::= ϵ | bool-lit | bit-lit | num-lit | time-lit | char-lit |
enum-lit | bitstr-lit | str-lit | ref | positional-aggregate |
type-conversion | unary-op expr | binary-op $expr_1$ $expr_2$ |
relational-op $expr_1$ $expr_2$
bool-lit ::= **FALSE** | **TRUE**
bit-lit ::= **BIT** bitlit
num-lit ::= **NUM** constant
time-lit ::= **TIME** constant **FS**
char-lit ::= **CHAR** constant
enum-lit ::= **ENUMLIT** id

bitstr-lit ::= **BITSTR** bit-lit*
str-lit ::= **STR** char-lit*
ref ::= **REF** modifier+
modifier ::= **SREF** id+ id | **INDEX** expr | **SELECTOR** id | **PARLIST** expr*
positional-aggregate ::= **PAGGR** expr*
type-conversion ::= **TYPECONV** expr type-mark
unary-op ::= **NOT** | **BNOT** | **PLUS** | **NEG** | **ABS** | **RNEG** | **RABS**
binary-op ::= **AND** | **NAND** | **OR** | **NOR** | **XOR** | **BAND** | **BNAND** | **BOR** | **BNOR** |
BXOR | **ADD** | **SUB** | **MUL** | **DIV** | **MOD** | **REM** | **EXP** | **RPLUS** | **RMINUS** |
RTIMES | **RDIV** | **REXPT** | **CONCAT**
relational-op ::= **EQ** | **NE** | **LT** | **LE** | **GT** | **GE** | **RLT** | **RLE** | **RGT** | **RGE**
opt-id ::= ϵ | id
opt-discrete-range ::= ϵ | discrete-range
opt-dotted-name ::= ϵ | dotted-name
opt-expr ::= ϵ | expr

The occurrences of `phase1-hook` in the Phase 2 abstract syntax for certain constructs point to the Phase 1 abstract syntax for the respective constructs, for the purposes of the (experimental) SDVS VHDL Symbolic Execution Trace Window.

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 4 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 4 VHDL hardware description is first parsed according to the Stage 4 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 ϵ. 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.

Fourteen kinds of descriptor are employed in Phase 1: *object*, *design file*, *configuration*, *package*, *entity*, *architecture*, *component*, *block name*, *process name*, *loop name*, *function*, *procedure*, *enumeration type element*, and *type*. Their structures are as follows:

object :

< 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 4 VHDL).

Finally, the object descriptor's **type** field contains the *type* of the identifier, represented by a pair $<$ **tmode, tdesc** $>$:

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

To introduce a bit more terminology, a type in which the **ref-mode** is **REF** or **OUT** will be called a *reference type*, while one whose **ref-mode** is **VAL** will be called a *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** = **tdesc** is the *basic type* of the type $<$ **tmode, tdesc** $>$ of which it is the second component.

design file :

$$<\textbf{id, qid, *DESIGN - FILE*}, \epsilon>$$

The **id** and **qid** fields are as above. ***DESIGN-FILE*** constitutes the **tag** field, and the **path** field contains ϵ.

configuration :

$$<\textbf{id, qid, *CONFIGURATION*, entity}>$$

The **id** and **qid** fields are as above. ***DESIGN-FILE*** constitutes the **tag** field, and the **entity** field contains the name of the configured entity.

package :

$$<\textbf{id, qid, *PACKAGE*, path, exported, pbody}>$$

The **id**, **qid**, **path**, and **exported** fields are as above. The **tag** field contains ***PACKAGE***. If this package has a body, the **pbody** field contains the transformed abstract syntax tree of the package body; otherwise it contains ϵ.

entity :

$$<\textbf{id, qid, *ENTITY*, path, exported}>$$

The **id**, **qid**, **path**, and **exported** fields are as above. ***ENTITY*** constitutes the **tag** field.

architecture :

< id, qid, *ARCHITECTURE*, path, exported >

The **id**, **qid**, **path**, and **exported** fields are as above. ***ARCHITECTURE*** constitutes the **tag** field.

component :

< id, qid, *COMPONENT*, path, exported >

The **id**, **qid**, **path**, and **exported** fields are as above. ***COMPONENT*** constitutes the **tag** field.

block name :

< id, qid, *BLOCKNAME*, path >

The **id** and **path** fields are as above. The **tag** field contains ***BLOCKNAME*** (the block label).

process name :

< id, qid, *PROCESSNAME*, path >

The **id** and **path** fields are as above. The **tag** field contains ***PROCESSNAME*** (the process label).

loop name :

< id, qid, *LOOPNAME*, path >

The **id**, **qid**, and **path** fields are as in a process name. The **tag** field contains ***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.

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 six kinds of type descriptor: those for *standard types*, *enumeration types*, *array types*, *subtypes*, *integer definition types*, and *record types*. Although record types are not actually incorporated in the Stage 4 VHDL language subset, the Stage 4 VHDL translator contains support for their eventual implementation.

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 4 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 4 VHDL standard types:

id = BOOLEAN, **tag = *BOOL***

id = BIT, **tag = *BIT***

id = UNIVERSAL_INTEGER, **tag = *INT***

id = INTEGER, **tag = *INT***

id = REAL, **tag = *REAL***

id = TIME, **tag = *TIME***

id = BIT_VECTOR, **tag = *ARRAYTYPE***

id = STRING, **tag = *ARRAYTYPE***

For completeness, we also provide *void* and *polymorphic* standard types for Stage 4 VHDL:

id = VOID, **tag = *VOID***

id = POLY, **tag = *POLY***

Functions are available that look up the type descriptors for the standard types; during translation Phase 1, these type descriptors are bound to the type identifiers in the **t((STANDARD))** component environment of the TSE **t**:

bool-type-desc(t) = t((**STANDARD**))(**BOOLEAN**)

bit-type-desc(t) = t((**STANDARD**))(**BIT**)

univint-type-desc(t) = t((**STANDARD**))(**UNIVERSAL_INTEGER**)

int-type-desc(t) = t((**STANDARD**))(**INTEGER**)

real-type-desc(t) = t((**STANDARD**))(**REAL**)

time-type-desc(t) = t((**STANDARD**))(**TIME**)

void-type-desc(t) = t((**STANDARD**))(**VOID**)

poly-type-desc(t) = t((**STANDARD**))(**POLY**)

In each of the above cases, the type descriptor has the form:

$$< \mathbf{id}, \epsilon, \mathbf{tag}, (\mathbf{STANDARD}), \mathbf{tt}, \mathbf{lb}, \mathbf{ub} >$$

char-type-desc(t) = t((**STANDARD**))(**CHARACTER**)

The type descriptor for the **CHARACTER** type has the form:

$$< \mathbf{CHARACTER}, \epsilon, *\mathbf{ENUMTYPE}*, (\mathbf{STANDARD}), \mathbf{tt}, (\mathbf{CHAR\ 0}), (\mathbf{CHAR\ 127}), \mathbf{literals} >$$

bitvector-type-desc(t) = t((**STANDARD**))(**BIT_VECTOR**)

The type descriptor for the **BIT_VECTOR** type has the form:

< **BIT_VECTOR**, ϵ, ***ARRAYTYPE***, **(STANDARD)**, **tt**, **TO**, **(NUM 0)**, ϵ, **bittypedesc** >

string-type-desc(t) = t(**(STANDARD)**)(**STRING**)

The type descriptor for the **STRING** type has the form:

< **STRING**, ϵ, ***ARRAYTYPE***, **(STANDARD)**, **tt**, **TO**, **(NUM 1)**, ϵ, **chartypedesc** >

enumeration type :

< **id**, **qid**, ***ENUMTYPE***, **path**, **exported**, **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 4 VHDL.

array type :

< **id**, **qid**, ***ARRAYTYPE***, **path**, **exported**, **direction**, **lb**, **ub**, **elty** >

Every array type has a name; unique names are generated for anonymous array types. Arrays in Stage 4 VHDL are *one-dimensional*, of index type **UNIVERSAL_INTEGER**. Note that the standard types **BIT_VECTOR** and **STRING** are array types.

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").

The following function accepts arguments for the creation of an array type:

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>

subtype :

< **id**, **qid**, ***SUBTYPE***, **path**, **exported**, **lb**, **ub**, **basetype** >

The **lb** and **ub** fields contain, respectively, abstract syntax trees for expressions that denote the subtype's lower and upper bounds. The **basetype** field contains the descriptor of the subtype's base type.

integer definition type :

< **id**, **qid**, ***INT_TYPE***, **path**, **exported**, **lb**, **ub**, **parenttype** >

The **lb** and **ub** fields contain, respectively, abstract syntax trees for expressions that denote the integer definition type's lower and upper bounds. The **parenttype** field contains the descriptor of the integer definition type's parent type, which is always **UNIVERSAL_INTEGER**.

record type :

< id, qid, *RECORDTYPE*, path, exported, 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 4 VHDL, and record types are included simply in preparation for the anticipated implementation of records.

6.2.1 Type and type descriptor predicates

Predicates are available for distinguishing specific types and type descriptors:

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

is-boolean-tdesc?(d) = idf(d)= **BOOLEAN**

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

is-bit-tdesc?(d) = idf(d)= **BIT**

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

is-integer-tdesc?(d) = tag(d)∈ **(*INT* *INT_TYPE*)**

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

is-real-tdesc?(d) = idf(d)= **REAL**

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

is-time-tdesc?(d) = idf(d)= **TIME**

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

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**

6.2.2 Additional primitive accessors and predicates

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.

idf(d) = hd(d)

qid(d) = hd(tl(d))

```
tag(d) = hd(tl(tl(d)))

path(d) = hd(tl(tl(tl(d))))

exported(d) = hd(tl(tl(tl(tl(d)))))

configured-entity(d) = hd(tl(tl(tl(tl(d)))))

component-name(d) = hd(tl(tl(tl(tl(tl(d))))))

type-tick-low(d) = hd(tl(tl(tl(tl(tl(d))))))

type-tick-high(d) = hd(tl(tl(tl(tl(tl(tl(d)))))))

base-type(d) = hd(tl(tl(tl(tl(tl(tl(tl(d))))))))

parent-type(d) = hd(tl(tl(tl(tl(tl(tl(tl(d))))))))

literals(d) = hd(tl(tl(tl(tl(tl(tl(tl(d))))))))

pbody(d) = hd(tl(tl(tl(tl(tl(d))))))

type(d) = hd(tl(tl(tl(tl(tl(d))))))

value(d) = hd(tl(tl(tl(tl(tl(tl(d)))))))

sources(d) = hd(tl(tl(tl(tl(tl(tl(d))))))))

signatures(d) = hd(tl(tl(tl(tl(tl(d))))))

body(d) = hd(tl(tl(tl(tl(tl(tl(d)))))))

characterizations(d) = hd(tl(tl(tl(tl(tl(tl(tl(d))))))))

direction(d) = hd(tl(tl(tl(tl(tl(d))))))

lb(d) = hd(tl(tl(tl(tl(tl(tl(d)))))))

ub(d) = hd(tl(tl(tl(tl(tl(tl(tl(d))))))))

elty(d) = hd(tl(tl(tl(tl(tl(tl(tl(tl(d))))))))))

components(d) = hd(tl(tl(tl(tl(tl(d))))))

mk-real-dotted-name(id*)
= (null(id*)→ ε,
   let first-id = hd(id*)
       and rest-ids = tl(id*) in
     (null(rest-ids)→ first-id,
      catenate(first-id,".",mk-real-dotted-name(rest-ids))))


pars(signature) = hd(signature)

rtype(signature) = hd(tl(signature))
```

```
get-base-type(d) = (tag(d)= *SUBTYPE* → base-type(d), d)

get-parent-type(d)
= (tag(d)∈ (*INT_TYPE* *DERIVED_TYPE*) → parent-type(d),
    error(cat("Not a derived type: ")(d)))


mk-type(tmode)(tdesc) = (tmode,tdesc)

tmode(type) = hd(type)

tdesc(type) = hd(tl(type))


mk-tmode(object-class)(ref-mode) = (object-class,ref-mode)

object-class(tmode) = hd(tmode)

ref-mode(tmode) = hd(tl(tmode))


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

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

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


is-readable?(type) = ref-mode(tmode(type))∈ (VAL REF)

is-writable?(type) = ref-mode(tmode(type))∈ (REF OUT)


is-ref?(expr) = consp(expr)∧ length(expr)= 2

is-paggr?(expr) = hd(expr)= PAGGR


is-unary-op?(op) = op ∈ (NOT PLUS NEG ABS)

is-binary-op?(op)
= op ∈ (AND NAND OR NOR XOR ADD SUB MUL DIV MOD REM EXP CONCAT)

is-relational-op?(op) = op ∈ (EQ NE LT LE GT GE)
```

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, configuration, package, entity, architecture, component, block, process, procedure, function, or loop):

UNIT :

r(*UNIT*) contains a tag that identifies what kind of program unit led to the creation of **r**. These tags are ***DESIGN-FILE*** (design file), ***PACKAGE*** (package), ***ENTITY*** (entity), ***ARCHITECTURE*** (architecture), ***PROCESS*** (process), ***PROCEDURE*** (procedure), ***FUNCTION*** (function), and ***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.

LAB :

When the **tag** of **r(*UNIT*)** is ***ARCHITECTURE***, the value bound to **r(*LAB*)** contains an identifier list of all the labels of concurrent statements (blocks or processes) in the corresponding architecture body. When the **tag** of **r(*UNIT*)** is ***PROCESS***, ***PROCEDURE***, ***FUNCTION***, or ***LOOP***, the value bound to **r(*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.

USED :

The environment corresponding to any program unit admitting USE clauses in its declarative part has a ***USED*** component. In this case, **r(*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 4 VHDL translator can remain *fixed* as that generated by Phase 1, a slight restriction is imposed on the concrete syntax of Stage 4 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.

IMPT :

Whenever a program unit has a ***USED*** component, it also has a ***IMPT*** component. **r(*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.

SENS :

When the **tag** of **r(*UNIT*)** is ***PROCESS***, the value bound to **r(*SENS*)** con-

tains a list of the transformed abstract syntax trees of the **ref**s 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 4 VHDL language constructs for which they perform static analysis) are: **DFT** (design files), **DUT** (design units), **CIT** (context items), **LUT** (library-units), **CFT** (configuration declarations), **BCT** (block configurations), **CMT** (component configurations), **BIT** (binding indications), **ENT** (entity declarations), **ART** (architecture bodies), **GDT** (generic declarations), **PDT** (port declarations), **GMT** (generic maps), **PMT** (port maps), **DT** (declarations), **CST** (concurrent statements), **SLT** (sensitivity lists), **SST** (sequential statements), **AT** (case alternatives), **DRT** (discrete ranges), **WT** (waveforms), **TRT** (transactions), **MET** (reference lists), **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 4 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; and
- 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.)

6.4.1 Phase 1 Semantic Domains

The semantic domains and function types for Phase 1 of the Stage 4 VHDL translator are as follows.

Primitive Semantic Domains

Bool = {**FALSE, TRUE**}	boolean constants
Bit = {**0, 1**}	bit constants
Char = {**(CHAR 0)**, ..., **(CHAR 127)**}	character constants (ASCII-128 representations)
n : N = {0, 1, 2, ...}	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 → (Dv + {*UNBOUND*})	environments
Tmode = {PATH} × Id* + ({CONST, VAR, SIG, DUMMY} × {VAL, OUT, REF, OBJ, ACC, TYP})	type modes
w : Type = Tmode × Desc	types
e : Value	values
h : CSet = P(Bool) + P(Char) + P_f(N) + {INT} + {ENUM}	case selection sets [P(•) denotes "powerset of" and P_f(•) denotes "set of finite subsets of"]
u : TDc = TEnv → Ans	declaration & concurrent statement continuations
c : TSc = TDc	sequential statement continuations
k : TEc = (Type × Value) → TSc	expression continuations
h : TMc = (Type* × Value*) → TSc	reference list continuations
y : TAc = CSet → TSc	case alternative continuations
v : TTc = Type → Ans	type continuations
z : Desc → TDc	descriptor continuations
Ans = (SD + Assert)* + Error	final answers

6.4.2 Phase 1 Semantic Functions

The semantic functions for Phase 1 of the Stage 4 VHDL translator are as follows.

DFT :	**Design → Id → Ans**	design file static semantics
DUT :	**DUnit* → Id → Path → Bool → TDc → TDc**	design unit static semantics
CIT :	**CItem* → Path → Bool → TDc → TDc**	context item static semantics
LUT :	**LUnit → Id → Path → Bool → TDc → TDc**	library unit static semantics

CFT :	**Config → Path → TDc → TDc**	configuration declaration static semantics
BCT :	**BConf → Id → Path → TDc → TDc**	block configuration static semantics
CMT :	**CConf → Id → Path → TDc → TDc**	component configuration static semantics
BIT :	**Bind → Id → BConf → TDc → TDc**	binding indication static semantics
ENT :	**Ent → Path → TDc → TDc**	entity declaration static semantics
ART :	**Arch → Path → TDc → TDc**	architecture body static semantics
GDT :	**GDec* → Path → Bool → TDc → TDc**	generic declaration static semantics
PDT :	**PDec* → Path → Bool → TDc → TDc**	port declaration static semantics
GMT :	**GMap → Id → Id → Path → Path → TDc → TDc**	generic map static semantics
PMT :	**PMap → Id → Id → Path → Path → TDc → TDc**	port map static semantics
DT :	**Dec* → Path → Bool → TDc → TDc**	declaration static semantics
CST :	**CStat* → Path → TDc → TDc**	concurrent statement static semantics
SLT :	**Ref* → Path → TDc → TDc**	sensitivity list static semantics
SST :	**SStat* → Path → TSc → TSc**	sequential statement static semantics
AT :	**Alt* → Type → Path → TAc → TSc**	case alternative static semantics
DRT :	**Drg → Type → Path → TAc → TSc**	discrete range static semantics
WT :	**Wave → Path → TEc → TSc**	waveform static semantics
TRT :	**Trans* → Path → TEc → TSc**	transaction static semantics
MET :	**Ref* → Path → TMc → TSc**	reference list static semantics
ET :	**Expr → Path → TEc → TSc**	expression static semantics
RT :	**Expr → Path → TEc → TSc**	expression static semantics
OT1 :	**Uop → TEc → TEc**	unary operator static semantics

OT2 :	$\mathbf{Bop} \rightarrow \mathbf{TEc} \rightarrow (\mathbf{Type} \times \mathbf{Value}) \rightarrow \mathbf{TEc}$	binary, relational operator static semantics
B :	$\mathbf{BitLit} \rightarrow \mathbf{Bit}$	bit values of bit literals (primitive)
N :	$\mathbf{NumLit} \rightarrow \mathbf{N}$	integer values of numeric literals (primitive)

6.5 Phase 1 Semantic Equations

6.5.1 Stage 4 VHDL Design Files

```
(DFT1) DFT [[ DESIGN-FILE id design-unit+ ]] (using-configuration)
       = let t0 = mk-initial-tse() in
           let p = %(ε)(id) in
            let t1 = enter-standard(t0) in
             let t2 = enter-textio(t1) in
              let t3 = enter(t2)(ε)(id)(<ε,*DESIGN-FILE* ,ε>) in
               let t4 = enter(extend(t3)(ε)(id))(p)(*UNIT* )(<ε,*DESIGN-FILE* >) in
                let t5 = enter(t4)(p)(*LAB* )(<ε,ε>) in
                 let t6 = enter(t5)(p)(*USED* )(<ε,ε>) in
                  let t7 = enter(t6)(p)(*IMPT* )(<ε,ε,ε>) in
                   enter-objects
                     ((VHDLTIME ,VHDLTIME_PREVIOUS ))
                     (<ε,*OBJECT* ,ε,tt,
                        ((DUMMY ,VAL ),vhdltime-type-desc(t0)),*UNDEF* ,ε>)(t7)(ε)(u)
                   where
                   u = λt.let use-clause = (USE ,((STANDARD ,ALL ))) in
                         DT [[ use-clause ]] (ε)(tt)(u1)(t)
                   where
                   u1 = λt.DUT [[ design-unit+ ]] (using-configuration)(p)(tt)(u2)
                                (t)
                    where
                    u2 = λt.phase1-tail(t)(using-configuration)(p)(u3)
                           where
                           u3 = λt.let transformed-abstract-syntax-tree = intermediate-phase
                                                                          (design-file)
                                                                          (using-configuration)
                                                                          (t) in
                                phase2
                                  (id)
                                  (transformed-abstract-syntax-tree)(t)
                                  (using-configuration)

enter-standard(t)
= let t1 = enter-package(t)(ε)(STANDARD ) in
    let t2 = enter(t1)(ε)(*USED* )(<ε,ε>) in
     let t3 = enter(t2)(ε)(*IMPT* )(<ε,ε,ε>) in
      let t4 = enter-standard-predefined(t3)((STANDARD) ) in
       t4

enter-textio(t)
= let t1 = enter-package(t)(ε)(TEXTIO ) in
    let t2 = enter(t1)(ε)(*USED* )(<ε,ε>) in
     let t3 = enter(t2)(ε)(*IMPT* )(<ε,ε,ε>) in
      let t4 = enter-textio-predefined(t3)((TEXTIO) ) in
       t4

enter-objects(id*)(field-values)(t)(p)(u)
= (null(id*)→ u(t),
    let id = hd(id*) in
     (t(p)(id)≠ *UNBOUND* → error(cat("Duplicate object declaration: ")($(p)
                                                                        (id))),
      let t1 = enter(t)(p)(id)(field-values) in
       enter-objects(tl(id*))(field-values)(t1)(p)(u)))
```

```
phase1-tail(t)(using-configuration)(p)(u)
= let t1 = update-tse-wrt-component-instantiations(using-configuration)(t) in
    let t2 = update-tse-wrt-configuration(t1)(using-configuration)(p) in
     u(t2)

intermediate-phase(design-file)(using-configuration)(t)
= DFX [[ design-file ]] (using-configuration)(t)

enter-standard-predefined(t)(p)
= let t1 = enter(t)(p)(BOOLEAN )(<ε,*BOOL* ,(STANDARD) ,tt,FALSE ,TRUE >) in
    let t2 = enter
                (t1)(p)(BIT )
                (<ε,*BIT* ,(STANDARD) ,tt,mk-bit-simp-symbol(0),
                   mk-bit-simp-symbol(1)>) in
     let t3 = enter(t2)(p)(UNIVERSAL_INTEGER )(<ε,*INT* ,(STANDARD) ,tt,ε,ε>) in
      let t4 = enter(t3)(p)(INTEGER )(<ε,*INT* ,(STANDARD) ,tt,ε,ε>) in
       let t5 = enter(t4)(p)(REAL )(<ε,*REAL* ,(STANDARD) ,tt,ε,ε>) in
        let t6 = enter(t5)(p)(TIME )(<ε,*TIME* ,(STANDARD) ,tt,ε,ε>) in
         let t7 = enter(t6)(p)(VHDLTIME )(<ε,*VHDLTIME* ,(STANDARD) ,tt,ε,ε>) in
          let t8 = enter(t7)(p)(VOID )(<ε,*VOID* ,(STANDARD) ,tt,ε,ε>) in
           let t9 = enter(t8)(p)(POLY )(<ε,*POLY* ,(STANDARD) ,tt,ε,ε>) in
            let t10 = enter
                        (t9)(p)(BIT_VECTOR )
                        (tl(array-type-desc
                              (BIT_VECTOR )(ε)((STANDARD) )(tt)(TO )((NUM 0) )(ε)
                              (bit-type-desc(t9)))) in
             let t11 = enter-characters(t10)(p) in
              let t12 = enter-string(t11)(p) in
               t12

enter-textio-predefined(t)(p) = t

enter-package(t)(p)(id)
= let p1 = %(p)(id) in
    let package-desc = <ε,*PACKAGE* ,p,tt,ε> in
     let t1 = enter(t)(p)(id)(package-desc) in
      let t2 = enter(extend(t1)(p)(id))(p1)(*UNIT* )(<ε,*PACKAGE* >) in
       let t3 = enter(t2)(p1)(*USED* )(<ε,ε>) in
        let t4 = enter(t3)(p1)(*IMPT* )(<ε,ε,ε>) in
         t4

enter-characters(t)(p)
= let id+ = gen-characters(0)(127) in
    let field-values1 = <ε,*ENUMTYPE* ,p,tt,hd(id+),last(id+),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)

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

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

6.5.2 Design Units

(DUT0) **DUT** ⟦ ε ⟧ (using-configuration)(p)(vis)(u)(t) = u(t)

(DUT1) **DUT** ⟦ design-unit design-unit* ⟧ (using-configuration)(p)(vis)(u)(t)
= **DUT** ⟦ design-unit ⟧ (using-configuration)(p)(vis)(u_1)(t)
where u_1 = λt.**DUT** ⟦ design-unit* ⟧ (using-configuration)(p)(vis)(u)(t)

(DUT2) **DUT** ⟦ **DESIGN-UNIT** context-item* library-unit ⟧ (using-configuration)(p)(vis)(u)(t)
= **CIT** ⟦ context-item* ⟧ (p)(vis)(u_1)(t)
where u_1 = λt.**LUT** ⟦ library-unit ⟧ (using-configuration)(p)(vis)(u)(t)

6.5.3 Contex Items

(CIT0) **CIT** ⟦ ε ⟧ (p)(vis)(u)(t) = u(t)

(CIT1) **CIT** ⟦ context-item context-item* ⟧ (p)(vis)(u)(t)
= **CIT** ⟦ context-item ⟧ (p)(vis)(u_1)(t)
where u_1 = λt.**CIT** ⟦ context-item* ⟧ (p)(vis)(u)(t)

(CIT2) **CIT** ⟦ **USE** dotted-name+ ⟧ (p)(vis)(u)(t)
= **let** decl = context-item **in**
DT ⟦ decl ⟧ (p)(vis)(u)(t)

6.5.4 Library Units

(LUT1) **LUT** ⟦ **CONFIGURATION** id_1 id_2 use-clause* block-config opt-id ⟧ (using-configuration)(p)(vis)(u)(t)
= u(t)

(LUT2) **LUT** ⟦ **PACKAGE** id decl* opt-id ⟧ (using-configuration)(p)(vis)(u)(t)
= **let** decl = library-unit **in**
DT ⟦ decl ⟧ (p)(vis)(u)(t)

(LUT3) **LUT** ⟦ **ENTITY** id generic-decl* port-decl* decl* opt-id ⟧ (using-configuration)(p)(vis)(u)(t)
= **let** entity-decl = library-unit **in**
ENT ⟦ entity-decl ⟧ (p)(u)(t)

(LUT4) **LUT** ⟦ **PACKAGEBODY** id decl* opt-id ⟧ (using-configuration)(p)(vis)(u)(t)
= **let** decl = library-unit **in**
DT ⟦ decl ⟧ (p)(vis)(u)(t)

(LUT5) **LUT** ⟦ **ARCHITECTURE** id_1 id_2 decl* conc-stat* opt-id ⟧ (using-configuration)(p)(vis)(u)(t)
= **let** architecture-body = library-unit **in**
ART ⟦ architecture-body ⟧ (using-configuration)(p)(u)(t)

6.5.5 Configuration Declarations

(CFT1) **CFT** ⟦ **CONFIGURATION** id_1 id_2 use-clause* block-config opt-id ⟧ (p)(u)(t)
= (¬null(opt-id)∧ opt-id ≠ id_1
→ error
(cat("Configuration declaration ")(id_1)
(" ended with incorrect identifier: ")(opt-id)),
let d = t(p)(id_2) in
(d = ***UNBOUND*** ∨ tag(d)≠ ***ENTITY***
→ error
(cat("No entity ")(id_2)(" for configuration declaration ")(id_1)),
(t(p)(id_1)≠ ***UNBOUND***
→ error(cat("Duplicate configuration declaration: ")($(p)($id_1$))),
let t_1 = enter(t)(p)(id_1)(<ε,***CONFIGURATION***,p,id_2>) in
let p_1 = %(p)(id_1)
and p_2 = %(p)(id_2) in
let t_2 = enter
(extend(t_1)(p)(id_1))(p_1)(***UNIT***)(<ε,***CONFIGURATION*** >) in
let t_3 = enter(t_2)(p_1)(***LAB***)(<ε,ε>) in
let t_4 = enter(t_3)(p_1)(***USED***)(<ε,ε>) in
let t_5 = enter(t_4)(p_1)(***IMPT***)(<ε,ε,ε>) in
DT ⟦ use-clause* ⟧ (p_1)(tt)(u_1)(t_5)
where u_1 = λt.**BCT** ⟦ block-config ⟧ (id_1)(p_2)(u_2)(t)
where u_2 = λt.u(t))))

(BCT1) **BCT** ⟦ **BLOCK-CONFIG** id use-clause* component-config* ⟧ (configuration-id)(p)(u)(t)
= let d = t(p)(id) in
(d = ***UNBOUND*** ∨ tag(d)≠ ***ARCHITECTURE***
→ error
(cat("In configuration declaration ")(configuration-id)
("the identifier ")(id)
("fails to refer to an architecture of entity ")
(last(p))),
let p_1 = %(p)(id) in
DT ⟦ use-clause* ⟧ (p)(tt)(u_1)(t)
where u_1 = λt.**CMT** ⟦ component-config* ⟧ (configuration-id)(p_1)(u)(t))

(CMT0) **CMT** ⟦ ε ⟧ (configuration-id)(p)(u)(t) = u(t)

(CMT1) **CMT** ⟦ component-config component-config* ⟧ (configuration-id)(p)(u)(t)
= **CMT** ⟦ component-config ⟧ (configuration-id)(p)(u_1)(t)
where u_1 = λt.**CMT** ⟦ component-config* ⟧ (configuration-id)(p)(u)(t)

(CMT2) **CMT** ⟦ **COMP-CONFIG** component-spec opt-binding-indication opt-block-config ⟧ (configuration-id)(p)(u)(t)
= let id^+ = hd(component-spec)
and component-name = second(component-spec)
and arch-id = last(p) in
let d = lookup-desc-for-ref((**REF** ,component-name))(p)(t) in
(d = ***UNBOUND*** ∨ tag(d)≠ ***COMPONENT***
→ error
(cat("In configuration declaration ")(configuration-id)
("there is no component declaration ")
(mk-real-dotted-name(component-name))
("for component instances ")(id^+)
("in architecture body ")(arch-id)),

```
            let p1 = %(p)(idf(d)) in
             process-component-spec(id+)(configuration-id)(arch-id)(p)(u1)(t)
              where
              u1 = λt.(null(opt-binding-indication)
                    → (null(opt-block-config)→ u(t),
                       error
                         (cat("Configuration ")(configuration-id)
                          ("has no binding indication for component specification ")
                          (component-spec))),
                    let binding-indication = opt-binding-indication in
                     BIT ⟦ binding-indication ⟧ (configuration-id)
                         (opt-block-config)(p1)(u)(t)))

process-component-spec(id*)(configuration-id)(arch-id)(p)(u)(t)
= (null(id*)→ u(t),
   let component-id = hd(id*) in
    let d = t(p)(component-id) in
     (d = *UNBOUND* ∨ tag(d)≠ *BLOCKNAME*
      → error
          (cat("In configuration ")(configuration-id)("the label ")(component-id)
           ("matches no component instantiation statement in architecture body ")
           (arch-id)),
      process-component-spec(tl(id*))(configuration-id)(arch-id)(p)(u)(t)))


(BIT1) BIT ⟦ BIND entity-aspect opt-generic-map-aspect opt-port-map-aspect ⟧
       (configuration-id)(opt-block-config)(p)(u)(t)
       = (hd(entity-aspect)= BOUND-ENTITY
          → process-bound-entity
              (entity-aspect)(opt-generic-map-aspect)(opt-port-map-aspect)
              (configuration-id)(opt-block-config)(p)(u)(t),
          process-bound-configuration
            (entity-aspect)(opt-generic-map-aspect)(opt-port-map-aspect)
            (configuration-id)(opt-block-config)(p)(u)(t))


process-bound-entity(entity-aspect)
                    (opt-generic-map-aspect)(opt-port-map-aspect)(configuration-id)(opt-block-config)
                    (p)(u)(t)
= let dotted-name = second(entity-aspect)
      and opt-id = third(entity-aspect) in
   let real-dotted-name = mk-real-dotted-name(dotted-name)
       and d = lookup-desc-for-ref((REF ,dotted-name))(p)(t) in
    (d = *UNBOUND* ∨ tag(d)≠ *ENTITY*
     → error
         (cat("Configuration declaration ")(configuration-id)
          ("refers to unknown entity ")(real-dotted-name)),
     let q = %(path(d))(idf(d)) in
      (opt-generic-map-aspect
       → let generic-map-aspect = opt-generic-map-aspect in
          GMT ⟦ generic-map-aspect ⟧ (configuration-id)(CONFIGURATION )(q)(p)(u1)(t),
       (null(third(t(p)(*GENERICS* )))→ u1(t),
        error
          (cat("Configuration declaration ")(configuration-id)
           ("requires a generic map for entity aspect ")
           (entity-aspect))))
       where
```

```
        u1 = λt.(opt-port-map-aspect
                → let port-map-aspect = opt-port-map-aspect in
                    PMT ⟦ port-map-aspect ⟧ (configuration-id)(CONFIGURATION )(q)
                        (p)(u2)(t),
                (null(third(t(p)(*PORTS* )))→ u2(t),
                 error
                    (cat("Configuration declaration ")(configuration-id)
                     ("requires a port map for entity aspect ")
                     (entity-aspect))))
         where
         u2 = λt.(null(opt-id)→ u(t),
                 (null(opt-block-config)→ u(t),
                  (opt-id ≠ second(opt-block-config)
                  → error
                       (cat("In configuration declaration ")(configuration-id)
                        ("the block specification identifier ")
                        (second(opt-block-config))
                        ("does not match the architecture identifier ")
                        (opt-id)("of the associated bound entity ")
                        (entity-aspect)),
                  u3(t))))
         where
         u3 = λt.let block-config = opt-block-config in
                 BCT ⟦ block-config ⟧ (configuration-id)(q)(u)(t))

process-bound-configuration(entity-aspect)
                           (opt-generic-map-aspect)(opt-port-map-aspect)(configuration-id)(opt-block-config)
                           (p)(u)(t)
= let dotted-name = second(entity-aspect) in
    let real-dotted-name = mk-real-dotted-name(dotted-name)
        and d = lookup-desc-for-ref((REF ,dotted-name))(p)(t) in
     (d = *UNBOUND* ∨ tag(d)≠ *CONFIGURATION*
      → error
           (cat("Configuration declaration ")(configuration-id)
            ("refers to unknown configuration ")(real-dotted-name)),
      u(t))
```

6.5.6 Entity Declarations

```
(ENT1) ENT ⟦ ENTITY id generic-decl* 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)),
          (t(p)(id)≠ *UNBOUND*
           → error(cat("Duplicate entity declaration: ")($(p)(id))),
           let t1 = enter(t)(p)(id)(<ε,*ENTITY* ,p,ff>) 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,ε>) in
               let t4 = enter(t3)(p1)(*USED* )(<ε,ε>) in
                let t5 = enter(t4)(p1)(*IMPT* )(<ε,ε,ε>) in
                 let t6 = enter(t4)(p1)(*GENERICS* )(<ε,ε>) in
                  let t7 = enter(t4)(p1)(*PORTS* )(<ε,ε>) in
                   GDT ⟦ generic-decl* ⟧ (p1)(tt)(u1)(t7)
```

where $u_1 = \lambda t.\underline{\mathbf{PDT}}\ [\![$ port-decl* $]\!]\ (p_1)(tt)(u_2)(t)$
where $u_2 = \lambda t.\underline{\mathbf{DT}}\ [\![$ decl* $]\!]\ (p_1)(tt)(u)(t)))$

6.5.7 Architecture Bodies

(ART1) $\underline{\mathbf{ART}}\ [\![$ **ARCHITECTURE** $id_1\ id_2$ decl* conc-stat* opt-id $]\!]$ (using-configuration)(p)(u)(t)
= ($\neg$null(opt-id)$\wedge$ opt-id $\neq id_1$
→ error
(cat("Architecture body ")(id_1)
(" ended with incorrect identifier ")(opt-id)),
let d = t(p)(id_2) **in**
(d = ***UNBOUND*** $\vee$ tag(d)$\neq$ ***ENTITY***
→ error(cat("No entity ")(id_2)(" for architecture body ")(id_1)),
let p_1 = %(p)(id_2) **in**
($t(p_1)(id_1)\neq$ ***UNBOUND***
→ error(cat("Duplicate architecture body: ")($(p_1)(id_1)$)),
let p_2 = %(p_1)(id_1) **in**
let t_1 = enter(t)(p_1)(id_1)(<ε,***ARCHITECTURE****,p_1,ff>) **in**
let t_2 = enter
(extend(t_1)(p_1)(id_1))(p_2)(***UNIT****)(<ε,***ARCHITECTURE****>) **in**
let t_3 = enter(t_2)(p_2)(***LAB****)(<ε,ε>) **in**
let t_4 = enter(t_3)(p_2)(***USED****)(<ε,ε>) **in**
let t_5 = enter(t_4)(p_2)(***IMPT****)(<ε,ε,ε>) **in**
$\underline{\mathbf{DT}}\ [\![$ decl* $]\!]\ (p_2)(tt)(u_1)(t_5)$
where
$u_1 = \lambda t_6.\underline{\mathbf{CST}}\ [\![$ conc-stat* $]\!]$ (using-configuration)(p_2)(u)(t_6))))

6.5.8 Generic Declarations

(GDT0) $\underline{\mathbf{GDT}}\ [\![\ \varepsilon\ ]\!]\ (p)(vis)(u)(t) = u(t)$

(GDT1) $\underline{\mathbf{GDT}}\ [\![$ generic-decl generic-decl* $]\!]\ (p)(vis)(u)(t)$
= $\underline{\mathbf{GDT}}\ [\![$ generic-decl $]\!]\ (p)(vis)(u_1)(t)$
where $u_1 = \lambda t.\underline{\mathbf{GDT}}\ [\![$ generic-decl* $]\!]\ (p)(vis)(u)(t)$

(GDT2) $\underline{\mathbf{GDT}}\ [\![$ **DEC GENERIC** id^+ type-mark opt-expr $]\!]\ (p)(vis)(u)(t)$
= lookup-type(type-mark)(p)(z)(t)
where
z = λd.**let** type = mk-type(**(REF VAL)**)(d) **in**
let generic* = third(t(p)(***GENERICS****)) **in**
let $generic_1^*$ = append(id^+,generic*) **in**
(duplicates?($generic_1^*$)
→ error
(cat("Duplicate generics declared in generic clause: ")
(generic-decl)),
let t_1 = enter(t)(p)(***GENERICS****)((ε,$generic_1^*$)) **in**
process-dec(id^+)(type)(opt-expr)(p)(vis)(u)(t_1))

(GDT3) $\underline{\mathbf{GDT}}\ [\![$ **SLCDEC GENERIC** id^+ slice-name opt-expr $]\!]\ (p)(vis)(u)(t)$
= **let** (type-mark,discrete-range) = slice-name **in**
lookup-type(type-mark)(p)(z)(t)
where
z = λd.**let** type = mk-type(**(REF VAL)**)(d) **in**

```
let generic* = third(t(p)(*GENERICS* )) in
 let generic1* = append(id+,generic*) in
  (duplicates?(generic1*)
   → error
        (cat("Duplicate generics declared in generic clause: ")
         (generic-decl)),
   let t1 = enter(t)(p)(*GENERICS* )((ε,generic1*)) in
    process-slcdec
      (id+)(type)(discrete-range)(opt-expr)(p)(vis)(u)(t1))
```

6.5.9 Port Declarations

(PDT0) **PDT** ⟦ ε ⟧ (p)(vis)(u)(t) = u(t)

(PDT1) **PDT** ⟦ port-decl port-decl* ⟧ (p)(vis)(u)(t)
= **PDT** ⟦ port-decl ⟧ (p)(vis)(u1)(t)
where u1 = λt.**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)(z)(t)
          where
          z = λd.let type = (case mode
                              IN → mk-type((SIG VAL) )(d),
                              OUT → mk-type((SIG OUT) )(d),
                              (INOUT ,BUFFER ) → mk-type((SIG REF) )(d),
                              OTHERWISE
                              → error
                                   (cat("Illegal mode in port declaration: ")
                                    (port-decl))) in
                  let port* = third(t(p)(*PORTS* )) in
                   let port1* = append(id+,port*) in
                    (duplicates?(port1*)
                     → error
                          (cat("Duplicate ports declared in port clause: ")
                           (port-decl)),
                     let t1 = enter(t)(p)(*PORTS* )((ε,port1*)) in
                      process-dec(id+)(type)(opt-expr)(p)(vis)(u)(t1))

duplicates?(things)
= (null(things)→ ff,
   let first-thing = hd(things)
       and rest-things = tl(things) in
    (first-thing ∈ rest-things → tt, duplicates?(rest-things)))
```

Refer to the discussion following semantic equation **DT5** in Section 6.5.11.

```
(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)(z)(t)
           where
```

```
z = λd.let type = (case mode
                     IN → mk-type((SIG VAL) )(d),
                     OUT → mk-type((SIG OUT) )(d),
                     (INOUT ,BUFFER ) → mk-type((SIG REF) )(d),
                     OTHERWISE
                     → error
                          (cat("Illegal mode in port declaration: ")
                           (port-decl))) in
          let port* = third(t(p)(*PORTS* )) in
           let port1* = append(id+,port*) in
            (duplicates?(port1*)
             → error
                  (cat("Duplicate ports declared in port clause: ")
                   (port-decl)),
             let t1 = enter(t)(p)(*PORTS* )((ε,port1*)) in
              process-slcdec
                (id+)(type)(discrete-range)(opt-expr)(p)(vis)(u)(t1))
```

Refer to the discussion following semantic equation **DT6** in Section 6.5.11.

6.5.10 Generic Maps and Port Maps

```
(GMT1) GMT ⟦ GENERICMAP assoc-elt+ ⟧ (id)(context)(formals-path)(actuals-path)(u)(t)
       = let formal* = get-refs-identifiers(map-hd(assoc-elt+))
             and actual* = get-refs-identifiers(map-second(assoc-elt+))
             and formal-generic* = third(t(formals-path)(*GENERICS* ))
             and local-generic* = third(t(actuals-path)(*GENERICS* )) in
         (duplicates?(formal*)
          → error
               (cat("Duplicate formal parts in association list: ")(assoc-elt+)),
          (context = CONFIGURATION
           → check-existence-formals(id)(formal*)(formal-generic*)(u1)(t),
          check-formal-local-correspondence
             (id)(formal*)(local-generic*)(u1)(t))
           where
           u1 = λt.(context = CONFIGURATION
                    → check-coverage-locals
                         (id)(local-generic*)(actual*)(u2)(t),
                    u(t))
            where
            u2 = λt.type-check-genericmap-elements
                      (assoc-elt+)(formals-path)(actuals-path)(u)(t))

type-check-genericmap-elements(assoc-elt*)(formals-path)(actuals-path)(u)(t)
= (null(assoc-elt*)→ u(t),
   let assoc-elt = hd(assoc-elt*) in
    type-check-genericmap-element(assoc-elt)(formals-path)(actuals-path)(u1)(t)
     where
     u1 = λt.type-check-genericmap-elements
              (tl(assoc-elt*))(formals-path)(actuals-path)(u)(t))

type-check-genericmap-element(assoc-elt)(formals-path)(actuals-path)(u)(t)
= let expr1 = hd(assoc-elt)
      and expr2 = second(assoc-elt) in
   ET ⟦ expr1 ⟧ (formals-path)(k1)(t)
```

```
      where
      k1 = λ(w1,e1),t.
            RT ⟦ expr2 ⟧ (actuals-path)(k2)(t)
             where
             k2 = λ(w2,e2),t.
                   (match-types(tdesc(w1),tdesc(w2))→ u(t),
                    error
                       (cat("has type mismatch in generic map association element: ")
                        (assoc-elt)))


(PMT1) PMT ⟦ PORTMAP assoc-elt+ ⟧ (id)(context)(formals-path)(actuals-path)(u)(t)
       = let formal* = get-refs-identifiers(map-hd(assoc-elt+))
             and actual* = get-refs-identifiers(map-second(assoc-elt+))
             and formal-port* = third(t(formals-path)(*PORTS* ))
             and local-port* = third(t(actuals-path)(*PORTS* )) in
         (duplicates?(formal*)
          → error
               (cat("Duplicate formal parts in association list: ")(assoc-elt+)),
          (context = CONFIGURATION
           → check-existence-formals(id)(formal*)(formal-port*)(u1)(t),
           check-formal-local-correspondence(id)(formal*)(local-port*)(u1)(t))
           where
           u1 = λt.(context = CONFIGURATION
                     → check-coverage-locals(id)(local-port*)(actual*)(u2)(t),
                     u2(t))
            where
            u2 = λt.check-portmap-elements
                      (assoc-elt+)(formals-path)(actuals-path)(u)(t))

check-portmap-elements(assoc-elt*)(formals-path)(actuals-path)(u)(t)
= (null(assoc-elt*)→ u(t),
   let assoc-elt = hd(assoc-elt*) in
    check-portmap-element(assoc-elt)(formals-path)(actuals-path)(u1)(t)
     where
     u1 = λt.check-portmap-elements
               (tl(assoc-elt*))(formals-path)(actuals-path)(u)(t))

check-portmap-element(assoc-elt)(formals-path)(actuals-path)(u)(t)
= let expr1 = hd(assoc-elt)
      and expr2 = second(assoc-elt) in
   ET ⟦ expr1 ⟧ (formals-path)(k1)(t)
    where
    k1 = λ(w1,e1),t.
          ET ⟦ expr2 ⟧ (actuals-path)(k2)(t)
           where
           k2 = λ(w2,e2),t.
                 (¬is-sig?(w2)
                  → error
                       (cat("Non-signal actual ")(expr2)
                        ("in port map element ")(assoc-elt)),
                 read-check-portmap-element(assoc-elt)(w1)(w2)(u1)(t)
                  where
                  u1 = λt.write-check-portmap-element
                            (assoc-elt)(w1)(w2)(formals-path)
                            (actuals-path)(u2)(t)
                   where
                   u2 = λt.type-check-portmap-element
                             (assoc-elt)(tdesc(w1))(tdesc(w2))(u)(t))
```

```
read-check-portmap-element(assoc-elt)(w1)(w2)(u)(t)
= (is-readable?(w1)
   → (is-readable?(w2)→ u(t),
      error
        (cat("Non-readable actual in port map association element: ")(assoc-elt))),
   u(t))

write-check-portmap-element(assoc-elt)(formal-type)(actual-type)(formal-path)(actual-path)(u)(t)
= let actual = second(assoc-elt) in
   (is-writable?(formal-type)
    → (is-writable?(actual-type)
       → let d = lookup-desc-for-ref(actual)(actual-path)(t) in
          let sources = sources(d) in
           (formal-path ∈ sources → u(t),
            rest(formal-path)∈ map-rest(sources)
            → error(cat("Resolved signal illegal in Stage 4 VHDL: ")(namef
                                                                    (d))),
            let t1 = enter
                       (t)(path(d))(idf(d))
                       (<ε,*OBJECT* ,path(d),exported(d),type(d),value(d),
                         cons(formal-path,sources)>) in
             u(t1)),
       error
         (cat("Non-writable actual ")(actual)
          ("in port map association element: ")(assoc-elt))),
    u(t))

type-check-portmap-element(assoc-elt)(d1)(d2)(u)(t)
= (match-types(d1,d2)→ u(t),
   error(cat("Type mismatch in port map association element: ")(assoc-elt)))


check-existence-formals(id)(formal*)(compare-formal*)(u)(t)
= (null(formal*)→ u(t),
   let first-formal = hd(formal*) in
    (first-formal ∈ compare-formal*
     → check-existence-formals
          (id)(tl(formal*))(remove(first-formal)(compare-formal*))(u)(t),
     error
       (cat("Statement or configuration declaration ")(id)
        ("refers to unknown formal ")(first-formal))))

check-formal-local-correspondence(id)(formal*)(local*)(u)(t)
= (null(formal*)
   → (null(local*)→ u(t),
      error
        (cat("Statement ")(id)
         ("fails to associate actuals with the following formals: ")
         (local*))),
   let first-formal = hd(formal*) in
    (first-formal ∈ local*
     → check-formal-local-correspondence
          (id)(tl(formal*))(remove(first-formal)(local*))(u)(t),
     error
       (cat("In statement ")(id)("the formal ")(first-formal)
        ("has no corresponding local "))))
```

```
check-coverage-locals(id)(local*)(compare-local*)(u)(t)
= (null(local*)→ u(t),
   let first-local = hd(local*) in
    (first-local ∈ compare-local*
     → check-coverage-locals
          (id)(tl(local*))(remove(first-local)(compare-local*))(u)(t),
     error
        (cat("Configuration declaration ")(id)("fails to associate the local ")
         (first-local)("as an actual with some formal"))))
```

6.5.11 Declarations

(DT0) **DT** ⟦ ε ⟧ (p)(vis)(u)(t) = u(t)

(DT1) **DT** ⟦ decl decl* ⟧ (p)(vis)(u)(t)
= **DT** ⟦ decl ⟧ (p)(vis)(u_1)(t)
where u_1 = λt.**DT** ⟦ decl* ⟧ (p)(vis)(u)(t)

(DT2) **DT** ⟦ package-decl package-decl* ⟧ (p)(vis)(u)(t)
= **DT** ⟦ package-decl ⟧ (p)(vis)(u_1)(t)
where u_1 = λt.**DT** ⟦ package-decl* ⟧ (p)(vis)(u)(t)

(DT3) **DT** ⟦ package-body package-body* ⟧ (p)(vis)(u)(t)
= **DT** ⟦ package-body ⟧ (p)(vis)(u_1)(t)
where u_1 = λt.**DT** ⟦ package-body* ⟧ (p)(vis)(u)(t)

(DT4) **DT** ⟦ use-clause use-clause* ⟧ (p)(vis)(u)(t)
= **DT** ⟦ use-clause ⟧ (p)(vis)(u_1)(t)
where u_1 = λt.**DT** ⟦ 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 ⟦ DEC object-class id+ type-mark opt-expr ⟧ (p)(vis)(u)(t)
      = let q = find-progunit-env(t)(p) in
          let d = t(q)(*UNIT* ) in
           let tg = tag(d) in
            (case object-class
               (CONST ,SYSGEN ) → lookup-type(type-mark)(p)(z)(t),
               VAR
               → (case tg
                    (*PACKAGE* ,*ENTITY* ,*ARCHITECTURE* )
                    → error
                         (cat("Illegal VARIABLE declaration in ")(tg)(" context: ")
                          (decl)),
                    OTHERWISE → lookup-type(type-mark)(p)(z)(t)),
               SIG
               → (case tg
                    (*PROCESS* ,*PROCEDURE* ,*FUNCTION* )
                    → error
                         (cat("Illegal SIGNAL declaration in ")(tg)(" context: ")
                          (decl)),
                    OTHERWISE → lookup-type(type-mark)(p)(z)(t)),
               OTHERWISE → error
                                (cat("Illegal object class in declaration: ")(decl)))
        where
        z = λd.let type = (object-class = CONST → mk-type((CONST VAL) )(d),
                           mk-type(mk-tmode(object-class)(REF ))(d)) in
             process-dec(id+)(type)(opt-expr)(p)(vis)(u)(t)
```

```
find-progunit-env(t)(p)
= (t(p)(*UNIT* )≠ *UNBOUND* → p,
   (null(p)→ error("No program unit ??! "),
    find-progunit-env(t)(rest(p))))

lookup-type(id*)(p)(z)(t)
= (null(id*)→ z(void-type-desc(t)),
   name-type(id*)(ε)(p)(t)(v)
    where
    v = λw.(second(tmode(w))= TYP → 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)(ε)(hd(name)) in
      (w1 = *UNBOUND*
       → error
            (cat("Unbound identifier in auxiliary semantic function NAME-TYPE: ")
             ($(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(pars(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
       (consp(x)
        → (second(tm)= TYP
           → (null(tl(x))
              → name-type(tl(name))(((DUMMY ,VAL ),d))(p)(t)(v),
              error
                 (cat("Explicit conversion of multiple expressions to type: ")
                  (namef(d)))),
             list-type(x)(p)(t)(h)
              where
              h = λw1*,e1*.
                   ((second(tm)= OBJ ∧ is-array?(type(d)))
                                ∨ (second(tm)∈ (REF VAL) ∧ is-array-tdesc?(d))
                    → (length(x)> 1
                       → error(cat("Too many array indices for: ")(namef
                                                                   (d))),
                       (is-integer-tdesc?(get-base-type(tdesc(hd(w1*))))
                        → 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))))),
            tg ∈ (*PROCEDURE* *FUNCTION*)
            → let rtype = compatible-signatures(w1*)(signatures(d)) in
                (null(rtype)
                 → error
                      (cat("Incompatible parameter types for: ")
                       (namef(d))),
                  name-type(tl(name))(rtype)(p)(t)(v)),
                error(cat("Cannot have an argument list: ")(namef
                                                    (d))))),
        ((second(tm)= OBJ ∧ is-record?(type(d)))
              ∨ (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
              (d2 = *UNBOUND* → error(cat("Unknown record field: ")(x)),
               let tmm = (second(tm)= OBJ → tmode(type(d)), tm) in
                name-type(tl(name))(mk-type(tmm)(d2))(p)(t)(v)),
         second(tm)≠ OBJ ∨ second(tm)≠ TYP
         → let w1 = lookup-local(x)(%(path(d))(idf(d)))(p)(t) in
             (w1 = *UNBOUND*
              → error
                   (cat("Unknown identifier in function NAME-TYPE: ")
                    ($(%(path(d))(idf(d)))(x))),
              second(tmode(w1))≠ ACC → name-type(tl(name))(w1)(p)(t)(v),
              hd(tm)= PATH
              → (¬null(tl(name))∧ ¬validate-access(name)(w1)(second(tm))
                  → error(cat("Illegal access via: ")(namef(tdesc(w1)))),
                  name-type
                    (tl(name))(((PATH ,tl(second(tm))),tdesc(w1)))(p)(t)
                    (v)),
              error
                (cat("Shouldn't happen in auxiliary semantic function NAME-TYPE: ")
                 (w1))),
         error(cat("Illegal access via: ")(namef(d)))))))

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

validate-access(name)(w)(q)
= let tg = tag(tdesc(w)) in
    (tg ∈ (*PROCEDURE* *FUNCTION*)
         ∧ (¬null(tl(name))∧ ¬consp(hd(tl(name))))
     → ¬null(q)∧ hd(name)= hd(q),
     tt)

list-type(expr*)(p)(t)(vv)
```

```
= (null(expr*)→ vv(ε),
   let expr = hd(expr*) in
    ET [[ expr ]] (p)(k)(t)
     where
     k = λ(w,e),t.
          (second(tmode(w))= ACC
           → error(cat("Non-value (an access): ")(namef(tdesc(w)))(expr)),
           list-type(tl(expr*))(p)(t)(λw*.vv(cons(w,w*)))))

lookup-local(id)(definition-path)(occurrence-path)(t)
= let d = t(definition-path)(id) in
   (d = *UNBOUND* → *UNBOUND* ,
    let tg = tag(d) in
     (tg ∈ (*BLOCKNAME* *PROCESSNAME* *LOOPNAME*) → ((DUMMY ,ACC ),d),
      (prefix-path(definition-path)(occurrence-path)∨ exported(d)
       → (case tg
            (*OBJECT* ,*ENUMELT* ) → ((DUMMY ,OBJ ),d),
            (*PACKAGE* ,*COMPONENT* ,*BLOCK* ,*PROCESS* ,*PROCEDURE* ,*FUNCTION* )
            → ((DUMMY ,ACC ),d),
            OTHERWISE → ((DUMMY ,TYP ),d)),
       *UNBOUND* )))

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

compatible-par-types(actuals)(formals)
= (length(actuals)≠ length(formals)→ ff,
   length(actuals)= 0 → tt,
   let w1 = hd(actuals)
       and w2 = hd(formals) in
    (match-types(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(actuals))(tl(formals)), ff),
     ff))

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

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-expr)(p)(vis)(u)(t)
= (null(opt-expr)
   → (is-const?(w)→ error(cat("Uninitialized constant: ")($(p)(hd(id+)))),
      enter-objects(id+)(<ε,*OBJECT* ,p,vis,w,*UNDEF* ,ε>)(t)(p)(u)),
   let expr = opt-expr in
```

```
RT [[ expr ]] (p)(k)(t)
where
k = λ(w1,e),t.
      let d = tdesc(w)
          and d1 = tdesc(w1) in
       (match-types(d,d1)
        → let init-val = ((is-sysgen?(w)∨ is-const?(w))
                                              ∧ ¬(is-array?(w)∨ is-record?(w))
                          → e,
                          *UNDEF* ) in
            enter-objects(id+)(<ε,*OBJECT* ,p,vis,w,init-val,ε>)(t)(p)(u),
        error(cat("Initialization type mismatch: ")(d)(d1))))

match-types(d1,d2)
= (case tag(d1)
     (*BOOL* ,*BIT* ,*REAL* ,*TIME* ,*ENUMTYPE* ) → d1 = get-base-type(d2),
     (*INT* ,*INT_TYPE* )
     → is-integer-tdesc?(get-base-type(d2))
          ∧ match-integer-types(d1)(get-base-type(d2)),
     *SUBTYPE* → match-types(get-base-type(d1),get-base-type(d2)),
     *ARRAYTYPE*
     → tag(d2)= *ARRAYTYPE* ∧ match-array-type-names(d1,d2),
     *RECORDTYPE*
     → tag(d2)= *RECORDTYPE*
          ∧ null(set-difference(filter-components(type(d1)))(filter-components(type(d2)))),
     OTHERWISE → match-type-names(idf(d1),idf(d2)))

match-integer-types(d1,d2)
= idf(d1)= UNIVERSAL_INTEGER ∨ idf(d2)= UNIVERSAL_INTEGER

get-base-type(d) = (tag(d)= *SUBTYPE* → base-type(d), d)

match-array-type-names(d1,d2)
= let idf1 = hd(d1)
      and idf2 = hd(d2) in
   (consp(idf1)∧ consp(idf2)→ match-type-names(hd(idf1),hd(idf2)),
    consp(idf1)→ match-type-names(hd(idf1),idf2),
    consp(idf2)→ match-type-names(idf1,hd(idf2)),
    match-type-names(idf1,idf2))

match-type-names(id1,id2)
= id1 = *ANONYMOUS* ∨ id2 = *ANONYMOUS*

array-size(d)
= (ub(d)∧ lb(d)
   → let lbound = hd(tl(lb(d)))
         and ubound = hd(tl(ub(d))) in
      (ubound−lbound)+1,
   -1)

filter-components(components)
= (null(components)→ ε,
   let component = hd(components) in
    cons((hd(component),second(component)),
    filter-components(tl(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, **ref**s are essentially sequences of identifiers and expression lists; **ref**s 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) DT [[ SLCDEC object-class id+ slice-name opt-expr ]] (p)(vis)(u)(t)
      = let (type-mark,discrete-range) = slice-name in
          let q = find-progunit-env(t)(p) in
           let d = t(q)(*UNIT*) in
            let tg = tag(d) in
             (case object-class
                (CONST ,SYSGEN ) → lookup-type(type-mark)(p)(z)(t),
                VAR
                → (case tg
                     (*PACKAGE* ,*ENTITY* ,*ARCHITECTURE* )
                     → error
                          (cat("Illegal VARIABLE declaration in ")(tg)
                           (" context: ")(decl)),
                     OTHERWISE → lookup-type(type-mark)(p)(z)(t)),
                SIG
                → (case tg
                     (*PROCESS* ,*PROCEDURE* ,*FUNCTION* )
                     → error
                          (cat("Illegal SIGNAL declaration in ")(tg)(" context: ")
                           (decl)),
                     OTHERWISE → lookup-type(type-mark)(p)(z)(t)),
                OTHERWISE
```

```
                → error(cat("Illegal object class in declaration: ")(decl)))
          where
          z = λd.let type = (object-class = CONST → mk-type((CONST VAL) )(d),
                              mk-type(mk-tmode(object-class)(REF ))(d)) in
                process-slcdec(id+)(type)(discrete-range)(opt-expr)(p)(vis)(u)(t)

process-slcdec(id+)(w)(discrete-range)(opt-expr)(p)(vis)(u)(t)
= let d = tdesc(w) in
   (¬is-array?(w)→ error(cat("Can't form slice of non-array type: ")(d)),
    let (direction,expr1,expr2) = discrete-range in
     RT ⟦ expr1 ⟧ (p)(k1)(t)
      where
      k1 = λ(w1,e1),t.
            RT ⟦ expr2 ⟧ (p)(k2)(t)
             where
             k2 = λ(w2,e2),t.
                   (¬(is-integer-tdesc?(get-base-type(tdesc(w1)))
                                  ∧ is-integer-tdesc?(get-base-type(tdesc(w2))))
                    → error
                         (cat("Non-integer array bound for: ")($(p)
                                                              (hd(id+)))),
                    let field-values = tl(array-type-desc
                                          (TEMP_NAME )(ε)(p)(vis)
                                          (direction)
                                          ((direction = TO
                                            → (e1 = *UNDEF*
                                                 → second(EX ⟦ expr1 ⟧ (p)(t)),
                                                 (NUM ,e1)),
                                              (e2 = *UNDEF*
                                               → second(EX ⟦ expr2 ⟧ (p)(t)),
                                               (NUM ,e2))))
                                          ((direction = TO
                                            → (e2 = *UNDEF*
                                                 → second(EX ⟦ expr2 ⟧ (p)(t)),
                                                 (NUM ,e2)),
                                              (e1 = *UNDEF*
                                               → second(EX ⟦ expr1 ⟧ (p)(t)),
                                               (NUM ,e1))))(elty(d))) in
                     (null(opt-expr)
                      → enter-array-objects
                           (id+)(idf(d))(tmode(w))(field-values)(t)(p)(vis)
                           (u),
                      check-array-aggregate(opt-expr)(p)(v)(t)
                       where
                       v = λw3.(match-types(elty(d),tdesc(w3))
                             → enter-array-objects
                                  (id+)(idf(d))(tmode(w))
                                  (field-values)(t)(p)(vis)(u),
                             error
                                (cat("Initialization type mismatch for: ")
                                 ($(p)(hd(id+)))))))))

enter-array-objects(id*)(array-type-name)(tmode)(field-values)(t)(p)(vis)(u)
= (null(id*)→ u(t),
   let id1 = hd(id*) in
    let id2 = new-array-type-name(array-type-name) in
     let d1 = cons(id2,field-values) in
```

```
        let t1 = enter(t)(p)(id2)(field-values) in
         let new-type = mk-type(tmode)(d1) in
          (t(p)(id1)≠ *UNBOUND*
           → error(cat("Duplicate array declaration: ")($(p)(id1))),
           let d2 = <ε,*OBJECT* ,p,vis,new-type,*UNDEF* ,ε> in
            let t2 = enter(t1)(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 ≠ BITSTR ∧ tg ≠ STR
     → error(cat("Improper array initialization aggregate: ")(expr)),
     let expr1 = hd(expr+) in
      RT [[ expr1 ]] (p)(k)(t)
       where k = λ(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 = λ(w1,e1),t.
          (w1 ≠ w → "Nonuniform array aggregate ",
           check-exprs(w)(tl(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 *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 *distinct*. Each of these distinct anonymous array types is given a distinct, new, system-generated name in Phase 1 of the Stage 4 VHDL translator (via the function **new-array-type-name**), and corresponding distinct type descriptors are entered into the TSE. If present, the initialization part of the declaration is a *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 **enter-array-objects**.

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

Refer also semantic equation **DT8**, shown below.

```
(DT7) DT [[ ETDEC id id+ ]] (p)(vis)(u)(t)
      = let field-values1 = <ε,*ENUMTYPE* ,p,vis,mk-enumlit(hd(id+)),
                            mk-enumlit(last(id+)),id+> in
          (check-enum-lits(t)(p)(id)(id+)
```

```
        → enter-objects((id))(field-values1)(t)(p)(u1),
        nil)
        where
        u1 = λt1.let d = cons(id,field-values1) in
                let field-values2 = <ε,*ENUMELT* ,p,vis,
                                    mk-type((CONST VAL) )(d)> in
                  enter-objects(id+)(field-values2)(t1)(p)(u)

check-enum-lits(t)(p)(id)(id*)
= (null(id*)→ tt,
   let id1 = hd(id*) in
    (lookup(t)(p)(id1)= *UNBOUND* → check-enum-lits(t)(p)(id)(tl(id*)),
     error
       (cat("Illegal overloading for enumeration literal: ")(id1)
        (" in enumeration type: ")($(p)(id)))))
```

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.

```
(DT8) DT [[ ATDEC id discrete-range type-mark ]] (p)(vis)(u)(t)
      = 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)(ε)(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,expr1,expr2,array-type-desc,(UNIVERSAL_INTEGER) ))(p)
                    (vis)(u)(t)

attributes-low-high(id,expr1,expr2,type-desc,attribute-type-mark)(p)(vis)(u)(t)
= let decl1 = (DEC ,SYSGEN ,(mk-tick-low(id)),attribute-type-mark,expr1)
      and decl2 = (DEC ,SYSGEN ,(mk-tick-high(id)),attribute-type-mark,expr2) in
    enter-objects((id))(tl(type-desc))(t)(p)(u1)
     where u1 = λt1.DT [[ decl1 ]] (p)(vis)(u2)(t1)
     where u2 = λt2.DT [[ decl2 ]] (p)(vis)(u)(t2)

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 identifiers.

```
(DT9) DT [[ PACKAGE id decl* opt-id ]] (p)(vis)(u)(t)
      = (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 d = <ε,*PACKAGE* ,p,vis,ε> in
          let t1 = enter(t)(p)(id)(d) in
           let t2 = enter(extend(t1)(p)(id))(%(p)(id))(*UNIT* )(<ε,*PACKAGE* >) in
            let t3 = enter(t2)(%(p)(id))(*USED* )(<ε,ε>) in
             let t4 = enter(t3)(%(p)(id))(*IMPT* )(<ε,ε,ε>) 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 = %(path(d))(id) in
            let t1 = enter(t)(q)(*LAB* )(<ε,ε>) in
             let t2 = enter(t1)(p)(id)(<ε,*PACKAGE* ,path(d),exported(d),*BODY* >) in
              DT [[ decl* ]] (q)(ff)(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) DT [[ PROCEDURE id proc-par-spec* ]] (p)(vis)(u)(t)
       = (t(p)(id)≠ *UNBOUND*
          → error(cat("Duplicate procedure declaration for: ")($(p)(id))),
          let p1 = %(p)(id) in
           let t1 = enter(extend(t)(p)(id))(p1)(*UNIT* )(<ε,*PROCEDURE* >) in
            enter-formal-pars(*PROCEDURE* )(proc-par-spec*)(t1)(p1)(u1)
             where
             u1 = λt2.let formals = let id+ = collect-fids(proc-par-spec*) in
                                    collect-formal-pars(id+)(t2)(p1) in
                          let d = <ε,*PROCEDURE* ,p,vis,
                                   ((formals,
                                     mk-type((CONST VAL) )(void-type-desc(t)))),ε,ε> in
                           u(enter(t2)(p)(id)(d)))

(DT12) DT [[ FUNCTION id func-par-spec* type-mark ]] (p)(vis)(u)(t)
       = (t(p)(id)≠ *UNBOUND*
          → error(cat("Duplicate function declaration for: ")($(p)(id))),
          let p1 = %(p)(id) in
           lookup-type(type-mark)(p)(z)(t)
            where
            z = λd1.let t1 = enter
                             (extend(t)(p)(id))(p1)(*UNIT* )(<ε,*FUNCTION* >) in
                      enter-formal-pars(*FUNCTION* )(func-par-spec*)(t1)(p1)(u1)
                       where
                       u1 = λt2.let formals = let id+ = collect-fids
                                                        (func-par-spec*) in
                                              collect-formal-pars
                                                (id+)(t2)(p1) in
                                    let d = <ε,*FUNCTION* ,p,vis,
                                             ((formals,mk-type((VAR VAL) )(d1))),ε,ε> 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: ")($(p)
                                                                  (hd(id+))))),
             OTHERWISE
             → error
                  (cat("Unimplemented object class ")(object-class)
                   (" for procedure parameters: ")($(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: ")($(p)
                                                        (hd(id+)))))),
          OTHERWISE
          → error
               (cat("Unimplemented object class ")(object-class)
                (" for function parameters: ")($(p)(hd(id+)))))),
     OTHERWISE → error(cat("Illegal subprogram tag: ")(tg)))
    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 = <ε,*OBJECT* ,p,tt,type,*UNDEF* ,ε> 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*)→ ε,
   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*)→ ε,
   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 **ff**, 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) DT ⟦ SUBPROGBODY subprog-spec decl* seq-stat* opt-id ⟧ (p)(vis)(u)(t)
       = let (tg,id,par-spec*,type-mark) = subprog-spec in
          let qname = $(p)(id)
              and d = t(p)(id) in
           (d = *UNBOUND*
            → let decl = subprog-spec in
                DT ⟦ decl ⟧ (p)(vis)(u1)(t)
                where
                u1 = λt.let d = t(p)(id) in
                        process-subprog-body
                          (t)(p)(id)(d)(decl*)(seq-stat*)(u),
               ¬(tag(d)∈ (*PROCEDURE* *FUNCTION*) )
               → error(cat(qname)(" is not a subprogram specification")),
               (tg = PROCEDURE ∧ tag(d)= *FUNCTION* )
                        ∨ (tg = FUNCTION ∧ tag(d)= *PROCEDURE* )
               → error(cat("Wrong kind of subprogram body: ")(qname)),
```

```
¬null(body(d))→ error(cat("Duplicate subprogram body: ")(qname)),
¬null(opt-id)∧ opt-id ≠ id
→ error
      (cat("Subprogram body ")(qname)
       (" ended with incorrect identifier ")(opt-id)),
let formals = let id+ = collect-fids(par-spec*) in
                collect-formal-pars(id+)(t)(%(p)(id)) in
 (formals ≠ pars(hd(signatures(d)))
  → error
       (cat("Nonconforming formal parameters for subprogram: ")(qname)),
  lookup-type(type-mark)(p)(z)(t)
   where
   z = λd1.(d1 ≠ tdesc(extract-rtype(d))
         → error
              (cat("Unequal result types for subprogram: ")
               (qname)),
          process-subprog-body(t)(p)(id)(d)(decl*)(seq-stat*)(u))))
```

```
process-subprog-body(t)(p)(id)(d)(decl*)(seq-stat*)(u)
= let p1 = %(p)(id) in
    let t1 = enter(t)(p1)(*LAB* )((ε,ε)) in
     let t5 = enter(t1)(p1)(*USED* )(<ε,ε>) in
      let t6 = enter(t5)(p1)(*IMPT* )(<ε,ε,ε>) in
       let t7 = enter
                  (t6)(p)(id)(<ε,tag(d),path(d),exported(d),signatures(d),ε,ε>) in
        DT [[ decl* ]] (p1)(tt)(u1)(t7)
         where u1 = λt2.SST [[ seq-stat* ]] (p1)(u2)(t2)
         where
         u2 = λt3.let t4 = enter
                             (t3)(p)(id)
                             (<ε,tag(d),path(d),exported(d),signatures(d),
                               (DX [[ decl* ]] (p1)(t3),SSX [[ seq-stat* ]] (p1)(t3)),ε>) in
                u(t4)
```

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 4 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).

```
(DT14) DT [[ USE dotted-name+ ]] (p)(vis)(u)(t)
       = let pkgs-used-here = tl(dotted-name+)∪ {hd(dotted-name+)} in
           process-use-clause(pkgs-used-here)(p)(vis)(u)(t)
```

```
process-use-clause(dotted-name+)(p)(vis)(u)(t)
= check-pkg-names(dotted-name+)(ε)(p)(vis)(j)(t)
    where
    j = λpkg-qualified-names.
         let pkg-qnames = remove-enclosing-pkgs(p)(t)(pkg-qualified-names) in
          let local-pkgs-used = third(t(p)(*USED* )) in
           let t1 = enter
                      (t)(p)(*USED* )
                      ((ε,pkg-qnames ∪ local-pkgs-used)) in
            let t2 = let d = t(p)(*IMPT* ) in
                      let qname-list = third(d)
                          and id-list = fourth(d) in
                       import-qualified-names
                          (pkg-qnames)(qname-list)(id-list)(p)(t1) in
             u(t2)

check-pkg-names(dotted-name*)(pkg-qualified-names)(p)(vis)(j)(t)
= (null(dotted-name*)→ j(pkg-qualified-names),
    let dn = hd(dotted-name*) in
     let suffix = last(dn) in
      (suffix ≠ ALL
       → error(cat("Selected name in USE clause must end with suffix ALL: ")(dn)),
       name-type(rest(dn))(ε)(p)(t)(v)
        where
        v = λw.let d = tdesc(w) in
              (tag(d)≠ *PACKAGE*
               → error(cat("Non-package name in USE clause: ")(namef
                                                                  (d))),
               check-pkg-names
                  (tl(dotted-name*))(cons(%(path(d))(idf(d)),pkg-qualified-names))
                  (p)(vis)(j)(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 = ε
    → enter(t)(p)(*IMPT* )((ε,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)(ε) 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(tl(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**.

```
(DT15) DT [[ STDEC id type-mark opt-discrete-range ]] (p)(vis)(u)(t)
       = lookup-type(type-mark)(p)(z)(t)
          where
          z = λd.let base-type-desc = get-base-type(d) in
                 (null(opt-discrete-range)
                  → let field-values = <ε,*SUBTYPE* ,p,vis,type-tick-low(d),
                                        type-tick-high(d),base-type-desc> in
                    attributes((id,ε,ε,d,field-values))(p)(vis)(u)(t),
                  let (direction,expr1,expr2) = opt-discrete-range in
                   RT [[ expr1 ]] (p)(k1)(t)
                    where
                    k1 = λ(w1,e1),t.
                           RT [[ expr2 ]] (p)(k2)(t)
                            where
                            k2 = λ(w2,e2),t.
                                   (match-types(tdesc(w1),base-type-desc)
                                   ∧ match-types(tdesc(w2),base-type-desc)
                                     → let field-values = <ε,*SUBTYPE* ,p,vis,
                                                           (direction = TO
                                                            → (e1 = *UNDEF*
                                                               → second
                                                                    (EX [[ expr1 ]]
                                                                        (p)(t)),
                                                               (NUM ,e1)),
                                                           (e2 = *UNDEF*
```

```
                                        → second
                                              (EX ⟦ expr2 ⟧
                                                   (p)(t)),
                                         (NUM ,e2))),
                                        (direction = TO
                                         → (e2 = *UNDEF*
                                            → second
                                                  (EX ⟦ expr2 ⟧
                                                       (p)(t)),
                                            (NUM ,e2)),
                                          (e1 = *UNDEF*
                                           → second
                                                 (EX ⟦ expr1 ⟧
                                                      (p)(t)),
                                           (NUM ,e1))),base-type-desc> in
                attributes
                   ((id,
                     (direction = TO → expr1,
                      expr2),
                     (direction = TO → expr2,
                      expr1),d,field-values))(p)
                   (vis)(u)(t),
             error
                (cat("Range constraint for subtype incompatible with base type: ")
                 (base-type-desc)(tdesc(w1))
                 (tdesc(w2))(decl))))

attributes(id,lower-bound,upper-bound,d,field-values)(p)(vis)(u)(t)
= let decl1 = (DEC ,SYSGEN ,(mk-tick-low(id)),(idf(d)),lower-bound)
      and decl2 = (DEC ,SYSGEN ,(mk-tick-high(id)),(idf(d)),upper-bound) in
   enter-objects((id))(field-values)(t)(p)(u1)
    where u1 = λt1.DT ⟦ decl1 ⟧ (p)(vis)(u2)(t1)
    where u2 = λt2.DT ⟦ decl2 ⟧ (p)(vis)(u)(t2)
```

Static semantic analysis of a subtype declaration involves making certain that the lower and upper bounds of the range constraint are compatible with the subtype's base type; declaring the `'low` and `'high` attributes (representing these bounds) as system-generated identifiers; and entering a subtype descriptor in the TSE.

```
(DT16) DT ⟦ ITDEC id discrete-range ⟧ (p)(vis)(u)(t)
       = let parent-type-desc = univint-type-desc(t) in
          let (direction,expr1,expr2) = discrete-range in
           RT ⟦ expr1 ⟧ (p)(k1)(t)
            where
            k1 = λ(w1,e1),t.
                  RT ⟦ expr2 ⟧ (p)(k2)(t)
                   where
                   k2 = λ(w2,e2),t.
                         (e1 = *UNDEF* ∨ e2 = *UNDEF*
                          → error
                               (cat("Non-static bound in range constraint: ")
                                (decl)),
                          (match-types(tdesc(w1),parent-type-desc)
                                              ∧ match-types(tdesc(w2),parent-type-desc)
                           → let field-values = <ε,*INT_TYPE* ,p,vis,
```

```
                                    (direction = TO
                                     → (NUM ,e1),
                                     (NUM ,e2)),
                                    (direction = TO
                                     → (NUM ,e2),
                                     (NUM ,e1)),parent-type-desc> in
          attributes
             ((id,(direction = TO → expr1, expr2),
               (direction = TO → expr2, expr1),parent-type-desc,field-values))
             (p)(vis)(u)(t),
       error
          (cat("Incompatible range constraint for integer type: ")
           (tdesc(w1))(tdesc(w2))(decl))))
```

Static semantic analysis of an integer definition type involves making certain that the lower and upper bounds of the range constraint are static expressions compatible with the integer type's parent type (`UNIVERSAL_INTEGER`); declaring the `'low` and `'high` attributes (representing these bounds) as system-generated identifiers; and entering an integer definition type descriptor in the TSE.

```
(DT17) DT ⟦ COMPONENT id generic-decl* port-decl* ⟧ (p)(vis)(u)(t)
       = let t1 = enter(t)(p)(id)(<ε,*COMPONENT* ,p,ff>) in
           let p1 = %(p)(id) in
            let t2 = enter(extend(t1)(p)(id))(p1)(*UNIT* )(<ε,*COMPONENT* >) in
             let t3 = enter(t2)(p1)(*LAB* )(<ε,ε>) in
              let t4 = enter(t3)(p1)(*USED* )(<ε,ε>) in
               let t5 = enter(t4)(p1)(*IMPT* )(<ε,ε,ε>) in
                let t6 = enter(t4)(p1)(*GENERICS* )(<ε,ε>) in
                 let t7 = enter(t4)(p1)(*PORTS* )(<ε,ε>) in
                  GDT ⟦ generic-decl* ⟧ (p1)(vis)(u1)(t7)
                    where u1 = λt.PDT ⟦ port-decl* ⟧ (p1)(vis)(u)(t)
```

6.5.12 Concurrent Statements

```
(CST0) CST ⟦ ε ⟧ (using-configuration)(p)(u)(t) = u(t)

(CST1) CST ⟦ conc-stat conc-stat* ⟧ (using-configuration)(p)(u)(t)
       = CST ⟦ conc-stat ⟧ (using-configuration)(p)(u1)(t)
           where u1 = λt.CST ⟦ conc-stat* ⟧ (using-configuration)(p)(u)(t)
```

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

```
(CST2) CST ⟦ BLOCK id block-header decl* conc-stat* opt-id ⟧ (using-configuration)(p)(u)(t)
       = let q = find-progunit-env(t)(p) in
           let labels = third(t(q)(*LAB* )) in
            (id ∈ labels
             → error(cat("Duplicate concurrent statement label: ")($(q)(id))),
             (¬null(opt-id)∧ opt-id ≠ id
              → error
                   (cat("BLOCK statement ")(id)
                    (" ended with incorrect identifier: ")(opt-id)),
```

```
                let t1 = enter(t)(q)(*LAB* )((ε,cons(id,labels))) in
                 let t2 = enter(t1)(q)(id)(<ε,*BLOCKNAME* ,p,ff>) in
                  let p1 = %(p)(id) in
                   let t3 = enter(extend(t2)(p)(id))(p1)(*UNIT* )(<ε,*BLOCK* >) in
                    let t4 = enter(t3)(p1)(*LAB* )(<ε,ε>) in
                     let t5 = enter(t4)(p1)(*USED* )(<ε,ε>) in
                      let t6 = enter(t5)(p1)(*IMPT* )(<ε,ε,ε>) in
                       let t7 = enter(t6)(p1)(*GENERICS* )(<ε,ε>) in
                        let t8 = enter(t7)(p1)(*PORTS* )(<ε,ε>) in
                         process-block-header(block-header)(id)(p1)(u2)(t8)
                          where u2 = λt.DT [[ decl* ]] (p1)(tt)(u1)(t)
                          where
                           u1 = λt.CST [[ conc-stat* ]] (using-configuration)(p1)(u)(t)))

process-block-header(block-header)(id)(p)(u)(t)
= let generic-part = hd(block-header)
      and port-part = second(block-header) in
   process-generic-part(generic-part)(id)(p)(u1)(t)
    where u1 = λt.process-port-part(port-part)(id)(p)(u)(t)

process-generic-part(generic-part)(id)(p)(u)(t)
= (null(generic-part)→ u(t),
   let generic-decl* = hd(generic-part)
       and generic-map-aspect = second(generic-part) in
    GDT [[ generic-decl* ]] (p)(tt)(u1)(t)
     where
     u1 = λt.(null(generic-map-aspect)→ u(t),
           GMT [[ generic-map-aspect ]] (id)(BLOCK )(p)(p)(u)(t)))

process-port-part(port-part)(id)(p)(u)(t)
= (null(port-part)→ u(t),
   let port-decl* = hd(port-part)
       and port-map-aspect = second(port-part) in
    PDT [[ port-decl* ]] (p)(tt)(u1)(t)
     where
     u1 = λt.(null(port-map-aspect)→ u(t),
           PMT [[ port-map-aspect ]] (id)(BLOCK )(p)(p)(u)(t)))


(CST3) CST [[ PROCESS id ref* decl* seq-stat* opt-id ]] (using-configuration)(p)(u)(t)
       = let labels = third(t(p)(*LAB* )) in
          (id ∈ labels
           → error(cat("Duplicate concurrent statement label: ")($(p)(id))),
           let t1 = enter(t)(p)(*LAB* )((ε,cons(id,labels))) in
            (¬null(opt-id)∧ opt-id ≠ id
             → error
                  (cat("PROCESS statement ")(id)
                   (" ended with incorrect identifier: ")(opt-id)),
            let t2 = enter(t1)(p)(id)(<ε,*PROCESSNAME* ,p,ff,ref*>) in
             let p1 = %(p)(id) in
              let t3 = enter(extend(t2)(p)(id))(p1)(*UNIT* )(<ε,*PROCESS* >) in
               let t4 = enter(t3)(p1)(*LAB* )(<ε,ε>) in
                let t5 = enter(t4)(p1)(*USED* )(<ε,ε>) in
                 let t6 = enter(t5)(p1)(*IMPT* )(<ε,ε,ε>) in
                  let t7 = enter(t6)(p1)(*SENS* )(<ε,ε>) in
                   SLT [[ ref* ]] (p1)(u2)(t7)
                    where u2 = λt.DT [[ decl* ]] (p1)(tt)(u1)(t)
                    where u1 = λt.SST [[ seq-stat* ]] (p1)(u)(t)))
```

```
find-architecture-env(t)(p)
= (null(p)∨ tag(t(p)(*UNIT* ))= *ARCHITECTURE* → p,
   find-architecture-env(t)(rest(p)))


(CST4) CST [[ SEL-SIGASSN atmark delay-type id expr ref selected-waveform+ ]]
       (using-configuration)(p)(u)(t)
       = let expr* = cons(expr,
                          collect-expressions-from-selected-waveforms
                            (selected-waveform+)) in
           let ref* = delete-duplicates
                        (collect-signals-from-expr-list(expr*)(t)(p)(ε)) in
             let case-alt+ = construct-case-alternatives
                               (ref)(delay-type)(selected-waveform+) in
               let case-stat = (CASE ,atmark,expr,case-alt+) in
                 let process-stat = (PROCESS ,id,ref*,ε,(case-stat),id) in
                   CST [[ process-stat ]] (using-configuration)(p)(u)(t)

collect-expressions-from-selected-waveforms(selected-waveform*)
= (null(selected-waveform*)→ ε,
   let selected-waveform = hd(selected-waveform*) in
    let waveform = second(selected-waveform)
        and discrete-range+ = third(selected-waveform) in
      let transaction-exprs = collect-transaction-expressions(second(waveform)) in
       nconc
          (transaction-exprs,
          cons(second(discrete-range+),
          cons(third(discrete-range+),
          collect-expressions-from-selected-waveforms
             (tl(selected-waveform*))))))

collect-transaction-expressions(trans*)
= (null(trans*)→ ε,
   let transaction = hd(trans*) in
    cons(second(transaction),collect-transaction-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)
= (¬consp(expr)→ signal-refs,
   is-ref?(expr)
   → let d = lookup-desc-for-ref(expr)(p)(t) in
       (tag(d)= *OBJECT* ∧ is-sig?(type(d))
        → cons(expr,
           (consp(second(expr))
            → collect-signals-from-expr-list(second(expr))(t)(p)(signal-refs),
            collect-signals-from-expr(second(expr))(t)(p)(signal-refs))),
         (consp(second(expr))
          → collect-signals-from-expr-list(second(expr))(t)(p)(signal-refs),
          collect-signals-from-expr(second(expr))(t)(p)(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)(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)),
    collect-signals-from-expr-list(expr)(t)(p)(signal-refs))


lookup-desc-for-ref(ref)(p)(t)
= let name = second(ref) in
    let id+ = (consp(last(name))→ rest(name), name) in
     let q = access(rest(id+))(t)(p) in
      lookup-desc-on-path(t)(q)(last(id+))


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


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


construct-case-alternatives(ref)(delay-type)(selected-waveform*)
= (null(selected-waveform*)→ ε,
   let selected-waveform = hd(selected-waveform*) in
    let waveform = second(selected-waveform)
        and discrete-range+ = third(selected-waveform) in
     let sig-assn-stat = (SIGASSN ,(AT ,mk-atmark()),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*))))


(CST5) CST ⟦ COND-SIGASSN atmark delay-type id ref cond-waveform* waveform ⟧
       (using-configuration)(p)(u)(t)
       = let expr* = nconc
                       (collect-expressions-from-conditional-waveforms
                           (cond-waveform*),
                       collect-transaction-expressions(second(waveform))) in
           let ref* = delete-duplicates
                       (collect-signals-from-expr-list(expr*)(t)(p)(ε)) in
            (null(cond-waveform*)
             → let sig-assn-stat = (SIGASSN ,(AT ,mk-atmark()),delay-type,ref,waveform) in
                 let process-stat = (PROCESS ,id,ref*,ε,(sig-assn-stat),id) in
                  CST ⟦ process-stat ⟧ (using-configuration)(p)(u)(t),
              let cond-part+ = construct-cond-parts
                                 (ref)(delay-type)(cond-waveform*)
                  and else-part = ((SIGASSN ,(AT ,mk-atmark()),delay-type,ref,waveform)) in
               let if-stat = (IF ,atmark,cond-part+,else-part) in
                let process-stat = (PROCESS ,id,ref*,ε,(if-stat),id) in
                 CST ⟦ process-stat ⟧ (using-configuration)(p)(u)(t))
```

```
collect-expressions-from-conditional-waveforms(cond-waveform*)
= (null(cond-waveform*)→ ε,
   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*)→ ε,
   let cond-waveform = hd(cond-waveform*) in
    let waveform = second(cond-waveform)
        and condition = third(cond-waveform) in
     let sig-assn-stat = (SIGASSN ,(AT ,mk-atmark()),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*))))


(CST6) CST ⟦ COMPINST id ref opt-generic-map-aspect opt-port-map-aspect ⟧
       (using-configuration)(p)(u)(t)
       = let d = lookup-desc-for-ref(ref)(p)(t) in
          (d = *UNBOUND* ∨ tag(d)≠ *COMPONENT*
           → error
                (cat("No component declaration ")("for component instance ")(id)),
           record-equivalent-nested-block-stat
              (conc-stat)(using-configuration)(p)(u)(t))
```

6.5.13 Sensitivity Lists

```
(SLT0) SLT ⟦ ε ⟧ (p)(u)(t) = u(t)

(SLT1) SLT ⟦ ref ref* ⟧ (p)(u)(t)
       = SLT ⟦ ref ⟧ (p)(u1)(t)
          where u1 = λt.SLT ⟦ ref* ⟧ (p)(u)(t)
```

The **ref**s 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 = λ(w,e),t.
                let d = tdesc(w) in
                 (¬is-sig?(w)
                  → error
                       (cat("Non-signal in process sensitivity list: ")(ref)),
                  let d1 = lookup(t)(p)(*SENS* ) in
                   let t1 = enter
                              (t)(p)(*SENS* )
                              (<ε,(cons(SLX ⟦ ref ⟧ (p)(t),sensitivity(d1)))>) in
                    u(t1))
```

6.5.14 Sequential Statements

(SST0) $\underline{\mathbf{SST}}$ ⟦ ϵ ⟧ (p)(c)(t) = c(t)

(SST1) $\underline{\mathbf{SST}}$ ⟦ seq-stat seq-stat* ⟧ (p)(c)(t)
= $\underline{\mathbf{SST}}$ ⟦ seq-stat ⟧ (p)(c_1)(t)
where c_1 = λt.$\underline{\mathbf{SST}}$ ⟦ seq-stat* ⟧ (p)(c)(t)

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

(SST2) $\underline{\mathbf{SST}}$ ⟦ **NULL** atmark ⟧ (p)(c)(t) = c(t)

NULL statements require no checking.

(SST3) $\underline{\mathbf{SST}}$ ⟦ **VARASSN** atmark ref expr ⟧ (p)(c)(t)
= **let** $expr_0$ = ref **in**
$\underline{\mathbf{ET}}$ ⟦ $expr_0$ ⟧ (p)(k)(t)
where
k = λ(w,e),t.
let d = tdesc(w) **in**
(¬is-var?(w)
→ error
(cat("**Illegal target in variable assignment statement:** ")
(seq-stat)),
¬is-writable?(w)
→ error(cat("**Read-only variable:** ")(namef(d))),
$\underline{\mathbf{RT}}$ ⟦ expr ⟧ (p)(k_1)(t)
where
k_1 = λ(w_1,e_1),t.
let d_1 = tdesc(w_1) **in**
(match-types(d,d_1)→ c(t),
error(cat("**Assignment type mismatch:** ")(d)(d_1))))

find-process-env(t)(p)
= (null(p)∨ tag(t(p)(***UNIT***))= ***PROCESS*** → p, find-process-env(t)(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) $\underline{\mathbf{SST}}$ ⟦ **SIGASSN** atmark delay-type ref waveform ⟧ (p)(c)(t)
= **let** expr = ref **in**
$\underline{\mathbf{ET}}$ ⟦ expr ⟧ (p)(k)(t)
where
k = λ(w,e),t.
let d = tdesc(w)
and q = find-process-env(t)(p) **in**
(¬is-sig?(w)
→ error
(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-desc-for-ref(ref)(p)(t) in
  let sources = sources(d1) in
   (q ∈ sources → c1(t),
    rest(q)∈ map-rest(sources)
    → error
         (cat("Resolved signal illegal in Stage 4 VHDL: ")
          (namef(d1))),
    let t1 = enter
             (t)(path(d1))(idf(d1))
             (<ε,*OBJECT* ,path(d1),exported(d1),type(d1),
               value(d1),cons(q,sources)>) in
     c1(t1))
  where
  c1 = λt1.WT [[ waveform ]] (p)(k1)(t)
       where
       k1 = λ(w1,e1),t.
            let d1 = tdesc(w1) in
             (match-types(d,d1)→ c(t),
              error
                (cat("Assignment type mismatch: ")
                 (d)(d1))))
```

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

A Stage 4 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.

```
(SST6) SST ⟦ CASE atmark expr case-alt+ ⟧ (p)(c)(t)
       = RT ⟦ expr ⟧ (p)(k)(t)
          where
          k = λ(w,e),t1.
                let d = get-base-type(tdesc(w)) in
                 AT ⟦ case-alt+ ⟧ (d)(p)(y)(t1)
                  where
                  y = λh,t2.
                        (¬case-type-ok(d)
                         → error
                               (cat("Illegal CASE selector type: ")(namef(d))
                                (seq-stat)),
                         ¬case-coverage(d)(h)
                         → error
                               (cat("Incomplete CASE coverage for type: ")
                                (namef(d))(seq-stat)),
                         c(t2))

case-type-ok(d)
= is-boolean-tdesc?(d)∨ is-bit-tdesc?(d)

case-coverage(d)(h)
= (is-boolean-tdesc?(d)∧ set-card(h)= 2)
   ∨ (is-bit-tdesc?(d)∧ set-card(h)= 2)

set-card(x) = length(x)
```

A Stage 4 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 4 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 4 VHDL parser) these expressions are converted into equivalent one-element discrete ranges.

A CASE statement must be checked for the following:

- The basic type of *all* the case selection sets (and thus of the expressions that define the discrete ranges) must be the same, and must match that of the selector expression. In Stage 4 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) SST [[ LOOP atmark id seq-stat* opt-id ]] (p)(c)(t)
       = let q = find-looplabel-env(t)(p) in
           let labels = third(t(q)(*LAB* )) in
            (id ∈ labels → error(cat("Duplicate loop label: ")($(q)(id))),
             let t1 = enter(t)(q)(*LAB* )((ε,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)(id)(<ε,*LOOPNAME* ,p>) in
                let p1 = %(p)(id) in
                 let t3 = enter(extend(t2)(p)(id))(p1)(*UNIT* )(<ε,*LOOP* >) in
                  let t4 = enter(t3)(p1)(*LAB* )(<ε,ε>) in
                   let t5 = enter(t4)(p)(id)(<ε,*LOOPNAME* ,p>) in
                    let c1 = λt.SST [[ seq-stat* ]] (p1)(c)(t) in
                     c1(t5)))

(SST8) SST [[ WHILE atmark id expr seq-stat* opt-id ]] (p)(c)(t)
       = let q = find-looplabel-env(t)(p) in
           let labels = third(t(q)(*LAB* )) in
            (id ∈ labels → error(cat("Duplicate loop label: ")($(q)(id))),
             let t1 = enter(t)(q)(*LAB* )((ε,cons(id,labels))) in
              (opt-id ≠ ε ∧ opt-id ≠ id
               → error
                    (cat("Loop ")(id)(" ended with incorrect identifier: ")(opt-id)),
               let t2 = enter(t1)(q)(id)(<ε,*LOOPNAME* ,p>) in
                let p1 = %(p)(id) in
                 let t3 = enter(extend(t2)(p)(id))(p1)(*UNIT* )(<ε,*LOOP* >) in
                  let t4 = enter(t3)(p1)(*LAB* )(<ε,ε>) in
                   let t5 = enter(t4)(p)(id)(<ε,*LOOPNAME* ,p>) in
                    let c1 = λt.SST [[ seq-stat* ]] (p1)(c)(t) in
                     RT [[ expr ]] (p1)(k)(t5)
                      where
                      k = λ(w,e),t.
                            (is-boolean?(w)→ c1(t),
                             error
                               (cat("Non-boolean condition in WHILE statement: ")
                                (tdesc(w))))))

(SST9) SST [[ FOR atmark id ref discrete-range seq-stat* opt-id ]] (p)(c)(t)
       = let q = find-looplabel-env(t)(p) in
           let labels = third(t(q)(*LAB* )) in
```

```
(id ∈ labels → error(cat("Duplicate loop label: ")($(q)(id))),
 let t1 = enter(t)(q)(*LAB* )((ε,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)(id)(<ε,*LOOPNAME* ,p>) in
    let p1 = %(p)(id) in
     let t3 = enter(extend(t2)(p)(id))(p1)(*UNIT* )(<ε,*LOOP* >) in
      let t4 = enter(t3)(p1)(*LAB* )(<ε,ε>) in
       let t5 = enter(t4)(p)(id)(<ε,*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(w2,e2),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)(t5),
                          error
                            (cat("Bounds type mismatch in FOR statement: ")
                             (seq-stat)))
          where
          u = λt6.c1(t6)
                where c1 = λt7.SST [[ seq-stat* ]] (p1)(c)(t7)))

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

In Stage 4 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 by **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 ⟦ EXIT atmark opt-dotted-name opt-expr ⟧ (p)(c)(t)
        = (null(find-loop-env(t)(p))
           → error(cat("EXIT statement not in a loop: ")(seq-stat)),
           (null(opt-dotted-name)→ c1(t),
            name-type(opt-dotted-name)(ε)(p)(t)(v)
             where
             v = λw.(tag(tdesc(w))≠ *LOOPNAME*
                   → error(cat("Not a loop name: ")(namef(tdesc(w)))),
                   c1(t)))
             where
             c1 = λt.(null(opt-expr)→ c(t),
                   let expr = opt-expr in
                    RT ⟦ expr ⟧ (p)(k)(t)
                    where
                    k = λ(w,e),t.
                          (is-boolean?(w)→ c(t),
                           error
                             (cat("Non-boolean condition in EXIT statement: ")
                              (tdesc(w))))))
```

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 **BOOLEAN**, an error is raised.

```
(SST11) SST ⟦ CALL atmark ref ⟧ (p)(c)(t)
        = let expr = ref in
           ET ⟦ expr ⟧ (p)(k)(t)
           where
           k = λ(w,e),t.
                (tag(tdesc(w))= *VOID* → c(t),
                 error(cat("Invalid procedure call: ")(seq-stat)))
```

A procedure call statement boils down to an expression that is a Stage 4 VHDL name. This expression is checked by **ET**, and must have a **VOID** basic type.

```
(SST12) SST ⟦ RETURN atmark opt-expr ⟧ (p)(c)(t)
        = let d = context(t)(p) in
           let tg = tag(d)
               and cname = namef(d) in
            (null(opt-expr)
             → (tg ≠ *PROCEDURE*
                → error
                     (cat("RETURN without expression in context of non-procedure: ")
                      (cname)(seq-stat)),
                c(t)),
             (tg ≠ *FUNCTION*
              → error
                   (cat("RETURN with expression in context of non-function: ")
                    (cname)(seq-stat)),
             let expr = opt-expr in
              RT ⟦ expr ⟧ (p)(k)(t)
              where
```

```
            k = λ(w,e),t.
                  (map-match-types(tdesc(w))(extract-rtypes(signatures(d)))
                   → c(t),
                   error
                     (cat("Incorrect return expression type in function: ")
                      (cname)(seq-stat)))))

context(t)(path)
= let d = t(path)(*UNIT* ) in
    (d = *UNBOUND* → context(t)(rest(path)),
     (case tag(d)
        (*PROCEDURE* ,*FUNCTION* ,*PACKAGE* ) → t(rest(path))(last(path)),
        OTHERWISE → context(t)(rest(path))))

extract-rtypes(signatures)
= (null(signatures)→ ε,
   cons(tdesc(rtype(hd(signatures))),extract-rtypes(tl(signatures))))
```

RETURN statements have two forms, depending on the PROCEDURE or FUNCTION context in which they can appear. Auxiliary semantic function **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 RETURN statement is in the proper context. If so, then if the 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.

```
(SST13) SST ⟦ WAIT atmark ref* opt-expr1 opt-expr2 ⟧ (p)(c)(t)
        = let c1 = λt.let d = lookup(t)(p)(*SENS* ) in
                     (¬null(sensitivity(d))
                      → error
                           (cat("WAIT statement ")(seq-stat)
                            (" illegal in process with sensitivity list: ")
                            (last(p))),
                      let c2 = λt.(null(opt-expr2)→ c(t),
                                   let expr2 = opt-expr2 in
                                    RT ⟦ expr2 ⟧ (p)(k2)(t)
                                     where
                                     k2 = λ(w2,e2),t2.
                                           (is-time?(w2)→ c(t2),
                                            error
                                              (cat("Ill-typed timeout clause in WAIT statement: ")
                                               (seq-stat)))) in
                       (null(opt-expr1)→ c2(t),
                        let expr1 = opt-expr1 in
                         RT ⟦ expr1 ⟧ (p)(k1)(t)
                          where
                          k1 = λ(w1,e1),t1.
                                 (is-boolean?(w1)→ c2(t1),
                                  error
                                    (cat("Non-boolean condition clause in WAIT statement: ")
                                     (seq-stat))))) in
            check-wait-refs(seq-stat)(ref*)(p)(c1)(t)

check-wait-refs(seq-stat)(ref*)(p)(c)(t)
= (null( ⟦ ref* ⟧ )→ c(t),
   let ref = hd(ref*)
       and c1 = λt.check-wait-refs(seq-stat)(tl(ref*))(p)(c)(t) in
    check-wait-ref(seq-stat)(ref)(p)(c1)(t))
```

```
check-wait-ref(seq-stat)(ref)(p)(c)(t)
= let expr = ref in
    ET [[ expr ]] (p)(k)(t)
    where
    k = λ(w,e),t.
        let d = tdesc(w) in
         (d = *UNBOUND* → error(cat("Unbound identifier: ")(namef
                                                              (d))),
          (is-sig?(w)→ c(t),
           error
              (cat("Non-signal ")(ref)
               (" in sensitivity clause of WAIT statement: ")
               (seq-stat))))
```

Semantic equation **SST13** specifies the static semantics of the WAIT statement. which consists of a sensitivity list **ref***, an optional condition **opt-expr**$_1$, and an optional timeout expression **opt-expr**$_2$. First, auxiliary semantic function **check-wait-refs** recursively traverses the sensitivity list, checking that each **ref** denotes a declared signal. Next, a descriptor for the special identifier ***SENS*** is looked up, and if its *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 **BOOLEAN**. Finally, if present, the timeout expression is checked to have basic type **TIME**.

6.5.15 Case Alternatives

```
(AT0) AT [[ ε ]] (d)(p)(y)(t) = y(emptyset)(t)

(AT1) AT [[ case-alt* case-alt ]] (d)(p)(y)(t)
      = AT [[ case-alt* ]] (d)(p)(y1)(t)
        where
        y1 = λh1,t1.
               AT [[ case-alt ]] (d)(p)(y2)(t1)
                 where
                 y2 = λh2,t2.
                        (case-overlap(d)((h1,h2))
                         → error
                               (cat("Overlapping case alternatives for type: ")
                                (namef(d))),
                         y(case-union(d)((h1,h2)))(t2))

(AT2) AT [[ CASECHOICE discrete-range+ seq-stat* ]] (d)(p)(y)(t)
      = DRT [[ discrete-range+ ]] (d)(p)(y1)(t)
        where
        y1 = λh,t1.
               SST [[ seq-stat* ]] (p)(c)(t1)
                 where c = λt2.y(h)(t2)

(AT3) AT [[ CASEOTHERS seq-stat* ]] (d)(p)(y)(t)
      = SST [[ seq-stat* ]] (p)(c)(t)
        where
        c = λt1.y((is-boolean-tdesc?(d)→ {FALSE ,TRUE },
                   is-bit-tdesc?(d)→ {0,1},
                   is-integer-tdesc?(d)→ INT ,
```

```
                is-enumeration-tdesc?(d)→ ENUM ,
                error
                   (cat("Illegal CASE selector type: ")(namef(d))(case-alt))))
             (t1)


case-overlap(d)(x,y)
= ((is-integer-tdesc?(d)∧ (x = INT ∨ y = INT ))
      ∨ (is-enumeration-tdesc?(d)∧ (x = ENUM ∨ y = ENUM ))
   → ff,
   x ∩ y ≠ emptyset)

case-union(d)(x,y)
= (is-integer-tdesc?(d)∧ (x = INT ∨ y = INT ) → INT ,
   is-enumeration-tdesc?(d)∧ (x = ENUM ∨ y = ENUM ) → ENUM ,
   x ∪ y)
```

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.

6.5.16 Discrete Ranges

```
(DRT0) DRT [[ ε ]] (d)(p)(y)(t) = y(emptyset)(t)

(DRT1) DRT [[ discrete-range discrete-range* ]] (d)(p)(y)(t)
       = DRT [[ discrete-range ]] (d)(p)(y1)(t)
          where
          y1 = λh1,t1.
                 DRT [[ discrete-range* ]] (d)(p)(y2)(t1)
                   where y2 = λh2,t2.y(h1 ∪ h2)(t2)
```

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

```
(DRT2) DRT [[ discrete-range ]] (d)(p)(y)(t)
       = let (direction,expr1,expr2) = discrete-range in
          RT [[ expr1 ]] (p)(k1)(t)
           where
           k1 = λ(w1,e1),t1.
                  (¬match-types(d,tdesc(w1))
                   → error(cat("CASE type mismatch: ")(d)(tdesc(w1))),
```

Appropriate constraints make sure that all **simulated_signal_sources** occur in the same simulation cycle as the containing **simulated_explicit_signal_state** and that they are legitimate sources of the **simulated_signal** according to the design source description.

```
                    e1 = *UNDEF*
                    → error(cat("Non-static CASE expression: ") [[ expr1 ]] ),
                    RT [[ expr2 ]] (p)(k2)(t1)
                     where
                     k2 = λ(w2,e2),t2.
                           (¬match-types(d,tdesc(w2))
                            → error
                                 (cat("CASE type mismatch: ")(d)(tdesc
                                                                 (w2))),
                            e2 = *UNDEF*
                            → error
                                 (cat("Non-static CASE expression: ")
                                  [[ expr2 ]] ),
                            y(mk-set(d)((direction,e1,e2)))(t2)))

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* ,*INT_TYPE* )
     → (direction = TO
        → (e1 ≤ e2 → {e1} ∪ mk-set(d)((direction,(e1+1),e2)), emptyset),
        (e1 ≥ e2 → {e1} ∪ mk-set(d)((direction,(e1−1),e2)), 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 → ε,
    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)→ ff, (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.

6.5.17 Waveforms and Transactions

(WT1) $\underline{\mathbf{WT}}$ ⟦ **WAVE** transaction$^+$ ⟧ (p)(k)(t) = $\underline{\mathbf{TRT}}$ ⟦ transaction$^+$ ⟧ (p)(k)(t)

(TRT1) $\underline{\mathbf{TRT}}$ ⟦ transaction transaction* ⟧ (p)(k)(t)
= $\underline{\mathbf{TRT}}$ ⟦ transaction ⟧ (p)(k_1)(t)
where
$k_1 = \lambda(w_1,e_1),t_1.$
let d_1 = tdesc(w_1) **in**
(null(transaction*)→ k((w_1,e_1))(t_1),
let transaction$_1^+$ = transaction* **in**
$\underline{\mathbf{TRT}}$ ⟦ transaction$_1^+$ ⟧ (p)(k_2)(t_1)
where
$k_2 = \lambda(w_2,e_2),t_2.$
let d_2 = tdesc(w_2) **in**
(¬match-types(d_1,d_2)
→ error
(cat("Type mismatch for waveform transactions: ")
(transaction)(hd(transaction$_1^+$))),
$e_1 \neq$ ***UNDEF*** $\wedge$ $e_2 \neq$ ***UNDEF***
→ ($e_1 \geq e_2$
→ error
(cat("Nonascending times for waveform transactions: ")
(transaction)(hd(transaction$_1^+$))),
k((w_2,e_2))(t_2)),
k((w_2,e_1))(t_2)))

(TRT2) $\underline{\mathbf{TRT}}$ ⟦ **TRANS** expr opt-expr ⟧ (p)(k)(t)
= $\underline{\mathbf{RT}}$ ⟦ expr ⟧ (p)(k_1)(t)
where
$k_1 = \lambda(w_1,e_1),t_1.$
(null(opt-expr)→ k((w_1,0))(t_1),
let $expr_2$ = opt-expr **in**
$\underline{\mathbf{RT}}$ ⟦ $expr_2$ ⟧ (p)(k_2)(t_1)
where
$k_2 = \lambda(w_2,e_2),t_2.$
(¬is-time?(w_2)
→ error
(cat("Transaction has ill-typed time expression: ")
(tdesc(w_2))),
$e_2 \neq$ ***UNDEF***
→ ($e_2 < 0$
→ error
(cat("Transaction has negative time expression: ")
(e_2)),
k((w_1,e_2))(t_2)),
k((w_1,e_2))(t_2)))

6.5.18 Expressions

(ET0) $\underline{\mathbf{ET}}$ ⟦ ε ⟧ (p)(k)(t) = k((ε,ε))(t)

(ET1) $\underline{\mathbf{ET}}$ ⟦ **FALSE** ⟧ (p)(k)(t) = k((mk-type((**CONST VAL**))(bool-type-desc(t)),**FALSE**))(t)

(ET2) $\underline{\mathbf{ET}}$ ⟦ **TRUE** ⟧ (p)(k)(t) = k((mk-type((**CONST VAL**))(bool-type-desc(t)),**TRUE**))(t)

```
(ET3) ET ⟦ BIT bitlit ⟧ (p)(k)(t)
      = k((mk-type((CONST VAL) )(bit-type-desc(t)),B ⟦ bitlit ⟧ ))(t)

(ET4) ET ⟦ NUM constant ⟧ (p)(k)(t)
      = k((mk-type((CONST VAL) )(int-type-desc(t)),N ⟦ constant ⟧ ))(t)

(ET5) ET ⟦ TIME constant time-unit ⟧ (p)(k)(t)
      = let normalized-constant = (case time-unit
                                    FS → N ⟦ constant ⟧ ,
                                    PS → 1000×N ⟦ constant ⟧ ,
                                    NS → 1000000×N ⟦ constant ⟧ ,
                                    US → 1000000000×N ⟦ constant ⟧ ,
                                    MS → 1000000000000×N ⟦ constant ⟧ ,
                                    SEC → 1000000000000000×N ⟦ constant ⟧ ,
                                    MIN → 60×(1000000000000000×N ⟦ constant ⟧ ),
                                    HR → 3600×(1000000000000000×N ⟦ constant ⟧ ),
                                    OTHERWISE
                                    → error
                                        (cat("Illegal unit name for physical type TIME: ")
                                         (time-unit))) in
        k((mk-type((CONST VAL) )(time-type-desc(t)),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)(ε)(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)(ε)(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))(ε)(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)(ε)(p)(t)(v)
where
v = λw.**let** d = tdesc(w) **in**
(second(tmode(w))= **TYP**
→ error(cat("Wrong context for a type: ")(namef(d))(expr)),
tag(d)= ***OBJECT*** → k((type(d),value(d)))(t),
tag(d)= ***ENUMELT*** → k((type(d),idf(d)))(t),
k((w,***UNDEF***))(t))

(ET10) **ET** ⟦ **PAGGR** $expr^*$ ⟧ (p)(k)(t)
= (length($expr^*$)= 1
→ **let** expr = hd($expr^*$) **in**
ET ⟦ expr ⟧ (p)(k)(t),
list-type($expr^*$)(p)(t)(vv)
where vv = λw^*.array-type(***ANONYMOUS***)($expr^*$)(w^*)(t)(p)(k))

(ET11) **ET** ⟦ unary-op expr ⟧ (p)(k)(t)
= **RT** ⟦ expr ⟧ (p)(k_1)(t)
where k_1 = λ(w,e),t.**OT1** ⟦ unary-op ⟧ (k)((w,e))(t)

(ET12) **ET** ⟦ binary-op $expr_1$ $expr_2$ ⟧ (p)(k)(t)
= **RT** ⟦ $expr_1$ ⟧ (p)(k_1)(t)
where
k_1 = λ(w_1,e_1),t.
RT ⟦ $expr_2$ ⟧ (p)(k_2)(t)
where k_2 = λ(w_2,e_2),t.
OT2 ⟦ binary-op ⟧ (k)((w_1,e_1))((w_2,e_2))(t)

(ET13) **ET** ⟦ relational-op $expr_1$ $expr_2$ ⟧ (p)(k)(t)
= **RT** ⟦ $expr_1$ ⟧ (p)(k_1)(t)
where
k_1 = λ(w_1,e_1),t.
RT ⟦ $expr_2$ ⟧ (p)(k_2)(t)
where
k_2 = λ(w_2,e_2),t.
OT2 ⟦ relational-op ⟧ (k)((w_1,e_1))((w_2,e_2))(t)

(RT1) **RT** ⟦ expr ⟧ (p)(k)(t)
= **ET** ⟦ expr ⟧ (p)(k_1)(t)
where
k_1 = λ(w,e),t.
let tm = tmode(w)
and d = tdesc(w) **in**
(second(tm)= **ACC** → error
(cat("Non-value (an access): ")(expr)),
second(tm)= **OUT**
→ error
(cat("Cannot dereference formal OUT parameter: ")(expr)),
second(tm)= **VAL** ∧ is-void-tdesc?(d)
→ error(cat("Void value: ")(expr)),
let w_1 = ((second(tm)= **AGR** → **(DUMMY AGR)** , **(DUMMY VAL)**),tdesc(w)) **in**
k((w_1,e))(t))

```
(OT1.1) OT1 ⟦ unary-op ⟧ (k)(w,e)(t)
        = let  d = tdesc(w)  in
            (match-types(d,argtypes1(unary-op)(d))
             → k((restype1(unary-op)(d),resval1(unary-op)(e)(d)))(t),
             error
               (cat("Argument type mismatch for unary operator: ")(unary-op)(d)))


argtypes1(unary-op)(d)
= (case unary-op
     NOT
     → (is-boolean-tdesc?(d)∨ is-bit-tdesc?(d)→ d,
        argtypes1-error(unary-op)(d)),
     (PLUS ,NEG ,ABS )
     → (is-integer-tdesc?(d)∨ is-time-tdesc?(d)→ d,
        argtypes1-error(unary-op)(d)),
     OTHERWISE → error
                          (cat("Unrecognized Stage 4 VHDL unary operator: ")(unary-op)))


argtypes1-error(unary-op)(d)
= error(cat("Unary operator ")(unary-op)(" not implemented for type: ")(d))


restype1(unary-op)(d) = mk-type((DUMMY VAL) )(d)


resval1(unary-op)(e)(d)
= (e = *UNDEF* → *UNDEF* ,
   (case unary-op
      NOT
      → (is-boolean-tdesc?(d)→ ¬e,
         is-bit-tdesc?(d)→ invert-bit(e),
         *UNDEF* ),
      PLUS → e,
      NEG → −e,
      ABS → abs(e),
      OTHERWISE → *UNDEF* ))


invert-bit(bitlit) = mk-bit-simp-symbol((−bitlit)+1)


mk-bit-simp-symbol(bitlit)
= (case bitlit
     0 → (BS 0 1) ,
     1 → (BS 1 1) ,
     OTHERWISE → error(cat("Can't construct simp symbol for bit: ")(bitlit)))


(OT2.1) OT2 ⟦ binary-op ⟧ (k)(w1,e1)(w2,e2)(t)
        = let  d1 = tdesc(w1)
               and d2 = tdesc(w2)  in
            (argtypes2(binary-op)((d1,d2))
             → k((restype2(binary-op)((d1,d2))(t),
                  resval2((d1,d2))(binary-op)((e1,e2))))(t),
             error
               (cat("Argument type mismatch for binary operator: ")(binary-op)(d1)
                (d2)))
```

```
(OT2.2) OT2 [[ relational-op ]] (k)(w1,e1)(w2,e2)(t)
        = let d1 = tdesc(w1)
              and d2 = tdesc(w2) in
            (argtypes2(relational-op)((d1,d2))
             → k((mk-type((DUMMY VAL) )(bool-type-desc(t)),
                  resval2((d1,d2))(relational-op)((e1,e2))))(t),
             error
               (cat("Argument type mismatch for relational operator: ")
                (relational-op)(d1)(d2)))

argtypes2(op)(d1,d2)
= (case op
     (AND ,NAND ,OR ,NOR ,XOR )
     → (case tag(d1)
          *BOOL* → is-boolean-tdesc?(d2)∨ argtypes2-error(op)(d1)(d2),
          *BIT* → is-bit-tdesc?(d2)∨ argtypes2-error(op)(d1)(d2),
          OTHERWISE → argtypes2-error(op)(d1)(d2)),
     (ADD ,SUB )
     → (case tag(d1)
          (*INT* ,*INT_TYPE* ) → match-types(d1)(d2)∨ argtypes2-error(op)(d1)(d2),
          (*TIME* ,*REAL* ) → d1 = d2 ∨ argtypes2-error(op)(d1)(d2),
          OTHERWISE → argtypes2-error(op)(d1)(d2)),
     MUL
     → (case tag(d1)
          (*INT* ,*INT_TYPE* ,*REAL* )
          → match-types(d1)(d2)∨ is-time-tdesc?(d2),
          *TIME*
          → is-integer-tdesc?(d2)∨ is-real-tdesc?(d2),
          OTHERWISE → argtypes2-error(op)(d1)(d2)),
     DIV
     → (case tag(d1)
          (*INT* ,*INT_TYPE* ,*REAL* )
          → match-types(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 tag(d1)
          (*INT* ,*INT_TYPE* )
          → is-integer-tdesc?(d2)∨ argtypes2-error(op)(d1)(d2),
          OTHERWISE → argtypes2-error(op)(d1)(d2)),
     EXP
     → (case tag(d1)
          (*INT* ,*INT_TYPE* ,*REAL* )
          → is-integer-tdesc?(d2)∨ 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 4 VHDL operator: ")(op)))

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

restype2(binary-op)(d1,d2)(t)
= (case binary-op
      (AND ,NAND ,OR ,NOR ,XOR ,ADD ,SUB ,MOD ,REM ,EXP ) → mk-type((DUMMY VAL) )(d1),
      MUL
      → (case tag(d1)
            (*INT* ,*INT_TYPE* ,*REAL* ) → mk-type((DUMMY VAL) )(d2),
            *TIME* → mk-type((DUMMY VAL) )(d1),
            OTHERWISE → error("Shouldn't happen!")),
      DIV
      → (case tag(d1)
            (*INT* ,*INT_TYPE* ,*REAL* ) → mk-type((DUMMY VAL) )(d2),
            *TIME*
            → (case tag(d2)
                  (*INT* ,*INT_TYPE* ,*REAL* ) → mk-type((DUMMY VAL) )(d1),
                  *TIME* → mk-type((DUMMY VAL) )(univint-type-desc(t)),
                  OTHERWISE → error("Shouldn't happen!")),
            OTHERWISE → error("Shouldn't happen!")),
      CONCAT → mk-type((DUMMY VAL) )(mk-concat-tdesc(d1)(d2)(t)),
      OTHERWISE
      → error(cat("Unrecognized Stage 4 VHDL binary operator: ")(binary-op)))

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

resval2(d1,d2)(op)(e1,e2)
= (e1 = *UNDEF* ∨ e2 = *UNDEF* → *UNDEF* ,
   let tg = tag(d1) in
    (case tg
      *BOOL*
      → (case op
            AND → e1 ∧ e2,
            NAND → ¬(e1 ∧ e2),
            OR → e1 ∨ e2,
            NOR → ¬(e1 ∨ e2),
            XOR → (e1 = e2 → ff, tt),
            EQ → e1 = e2,
            NE → e1 ≠ e2,
            LT → ¬e1 ∧ e2,
            LE → ¬e1 ∨ e2,
            GT → e1 ∧ ¬e2,
            GE → e1 ∨ ¬e2,
```

```
      OTHERWISE
      → error
            (cat("Unrecognized Stage 4 VHDL 'boolean' binary operator: ")(op))),
*BIT*
→ (case op
      AND
      → (e1 = 1 ∧ e2 = 1 → mk-bit-simp-symbol(1), mk-bit-simp-symbol(0)),
      NAND
      → (e1 = 0 ∨ e2 = 0 → mk-bit-simp-symbol(1), mk-bit-simp-symbol(0)),
      OR
      → (e1 = 1 ∨ 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 ∨ e2 = 1,
      GT → e1 = 1 ∧ e2 = 0,
      GE → e1 = 1 ∨ e2 = 0,
      OTHERWISE
      → error
            (cat("Unrecognized Stage 4 VHDL 'bit' binary operator: ")(op))),
(*INT* ,*INT_TYPE* ,*TIME* )
→ (case op
      ADD → e1+e2,
      SUB → e1−e2,
      MUL → e1×e2,
      DIV → (e2 = 0 → error("Illegal division by zero!"),
               e1/e2),
      MOD → mod(e1,e2),
      REM → rem(e1,e2),
      EXP → e1 ^ e2,
      EQ → e1 = e2,
      NE → e1 ≠ e2,
      LT → e1 < e2,
      LE → e1 ≤ e2,
      GT → e1 > e2,
      GE → e1 ≥ e2,
      OTHERWISE
      → error
            (cat("Unrecognized Stage 4 VHDL 'integer' binary operator: ")(op))),
*REAL* → error(cat("Floating point operator not yet implemented: ")(op)),
*ENUMTYPE*
→ (case op
      EQ → e1 = e2,
      NE → e1 ≠ e2,
      LT → enum-lt(e1)(e2)(literals(d1)),
      LE → enum-le(e1)(e2)(literals(d1)),
      GT → enum-lt(e2)(e1)(literals(d1)),
      GE → enum-le(e2)(e1)(literals(d1)),
      OTHERWISE
      → error
            (cat("Unrecognized Stage 4 VHDL 'enumeration type' binary operator: ")
             (op))),
*ARRAYTYPE* → *UNDEF* ,
OTHERWISE
```

```
        → error(cat("Unrecognized Stage 4 VHDL binary operator type: ")(tg))))

enum-lt(e1)(e2)(enum-lits)
= let  e1pos = position(e1)(enum-lits)
       and e2pos = position(e2)(enum-lits)  in
    e1pos < e2pos

enum-le(e1)(e2)(enum-lits)
= let  e1pos = position(e1)(enum-lits)
       and e2pos = position(e2)(enum-lits)  in
    e1pos ≤ e2pos
```

6.5.19 Primitive Semantic Equations

(N1) $\underline{\mathbf{N}}$ ⟦ constant ⟧ = constant

(B1) $\underline{\mathbf{B}}$ ⟦ bitlit ⟧ = bitlit

7 Interphase Abstract Syntax Tree Transformation

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

Stage 2 VHDL was a considerably more sophisticated language subset. Consequently, it became convenient to allow Phase 2 of the VHDL translator for Stage 2 VHDL and subsequent stages, in particular Stage 4 VHDL, 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 4 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** (design files), **DUX** (design units), **CIX** (context items), **LUX** (library-units), **CFX** (configuration declarations), **BCX** (block configurations), **CMX** (component configurations), **BIX** (binding indications), **ENX** (entity declarations), **ARX** (architecture bodies), **GDX** (generic declarations), **PDX** (port declarations), **GMX** (generic maps), **PMX** (port maps), **DX** (declarations), **CSX** (concurrent statements), **SLX** (sensitivity lists), **SSX** (sequential statements), **AX** (case alternatives), **DRX** (discrete ranges), **WX** (waveforms), **TRX** (transactions), **MEX** (reference lists), and **EX** and **RX** (expressions). These are assisted by 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:

- "desugaring" of component instantiation statements into pairs of nested `BLOCK` statements

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

Listed below are the names of Common Lisp functions, not denotationally defined, that assist in the first of these tasks.

record-equivalent-nested-block-stat

construct-equivalent-nested-block-stat

update-tse-wrt-component-instantiations

update-tse-wrt-configuration

accomplish-generic-and-port-maps

7.2 Transformed Abstract Syntax of Names

An important case in point is the translation of names, e.g. **ref**s, 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 **ref**s would exhibit analogous complexity; instead, it was deemed preferable to transform the abstract syntax of **ref**s into a form more suitable for Phase 2.

Thus, the abstract syntax of **ref**s used in Phase 1 is:

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

while the abstract syntax of **ref**s used in Phase 2 is:

```
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 4 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 **ref**s 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 4 VHDL Design Files

(DFX1) **DFX** ⟦ **DESIGN-FILE** id design-unit$^+$ ⟧ (using-configuration)(t)
= **let** p = %(ϵ)(id) **in**
(**DESIGN-FILE** ,id,**DUX** ⟦ design-unit$^+$ ⟧ (using-configuration)(p)(t))

7.3.2 Design Units

(DUX0) **DUX** ⟦ ϵ ⟧ (using-configuration)(p)(t) = ϵ

(DUX1) **DUX** ⟦ design-unit design-unit* ⟧ (using-configuration)(p)(t)
= cons(**DUX** ⟦ design-unit ⟧ (using-configuration)(p)(t),
DUX ⟦ design-unit* ⟧ (using-configuration)(p)(t))

(DUX2) **DUX** ⟦ **DESIGN-UNIT** context-item* library-unit ⟧ (using-configuration)(p)(t)
= (**DESIGN-UNIT** ,**CIX** ⟦ context-item* ⟧ (p)(t),
LUX ⟦ library-unit ⟧ (using-configuration)(p)(t))

7.3.3 Contex Items

(CIX0) **CIX** ⟦ ϵ ⟧ (p)(t) = ϵ

(CIX1) **CIX** ⟦ context-item context-item* ⟧ (p)(t)
= cons(**CIX** ⟦ context-item ⟧ (p)(t),**CIX** ⟦ context-item* ⟧ (p)(t))

(CIX2) **CIX** ⟦ **USE** dotted-name$^+$ ⟧ (p)(t)
= **let** decl = context-item **in**
DX ⟦ decl ⟧ (p)(t)

7.3.4 Library Units

(LUX1) **LUX** ⟦ **CONFIGURATION** id_1 id_2 use-clause* block-config opt-id ⟧ (using-configuration)(p)(t)
= **let** configuration-decl = library-unit **in**
CFX ⟦ configuration-decl ⟧ (p)(t)

(LUX2) **LUX** ⟦ **PACKAGE** id decl* opt-id ⟧ (using-configuration)(p)(t)
= **let** decl = library-unit **in**
DX ⟦ decl ⟧ (p)(t)

(LUX3) **LUX** ⟦ **ENTITY** id generic-decl* port-decl* decl* opt-id ⟧ (using-configuration)(p)(t)
= **let** entity-decl = library-unit **in**
ENX ⟦ entity-decl ⟧ (p)(t)

(LUX4) **LUX** ⟦ **PACKAGEBODY** id decl* opt-id ⟧ (using-configuration)(p)(t)
= **let** decl = library-unit **in**
DX ⟦ decl ⟧ (p)(t)

(LUX5) **LUX** ⟦ **ARCHITECTURE** id_1 id_2 decl* conc-stat* opt-id ⟧ (using-configuration)(p)(t)
= **let** architecture-body = library-unit **in**
ARX ⟦ architecture-body ⟧ (using-configuration)(p)(t)

7.3.5 Configuration Declarations

(CFX1) **CFX** ⟦ **CONFIGURATION** id_1 id_2 use-clause* block-config opt-id ⟧ (p)(t)
= (**CONFIGURATION** ,id_1,id_2,**DX** ⟦ use-clause* ⟧ (%(p)(id_1))(t),
BCX ⟦ block-config ⟧ (%(p)(id_2))(t),opt-id)

(BCX1) **BCX** ⟦ **BLOCK-CONFIG** id use-clause* component-config* ⟧ (p)(t)
= (**BLOCK-CONFIG** ,id,**DX** ⟦ use-clause* ⟧ (p)(t),
let p_1 = %(p)(id) **in**
CMX ⟦ component-config* ⟧ (p_1)(t))

(CMX0) **CMX** ⟦ ε ⟧ (p)(t) = ε

(CMX1) **CMX** ⟦ component-config component-config* ⟧ (p)(t)
= cons(**CMX** ⟦ component-config ⟧ (p)(t),**CMX** ⟦ component-config* ⟧ (p)(t))

(CMX2) **CMX** ⟦ **COMP-CONFIG** component-spec opt-binding-indication opt-block-config ⟧ (p)(t)
= (null(opt-binding-indication)→ (**COMP-CONFIG** ,component-spec,ε,ε),
let binding-indication = opt-binding-indication
and component-name = second(component-spec) **in**
let d = lookup-desc-for-ref((**REF** ,component-name))(p)(t) **in**
let entity-aspect = second(binding-indication) **in**
let p_1 = (hd(entity-aspect)= **BOUND-ENTITY**
→ **let** entity-name = last(second(entity-aspect)) **in**
let entity-desc = lookup(t)(p)(entity-name) **in**
%(path(entity-desc))(entity-name),
let configuration-desc = lookup
(t)(p)
(last(second(entity-aspect))) **in**
let entity-name = configured-entity(configuration-desc) **in**
let entity-desc = lookup(t)(p)(entity-name) **in**
%(path(entity-desc))(entity-name))
and p_2 = %(p)(idf(d)) **in**
(**COMP-CONFIG** ,component-spec,**BIX** ⟦ binding-indication ⟧ (p_1)(p_2)(t),
(null(opt-block-config)→ ε,
let block-config = opt-block-config **in**
BCX ⟦ block-config ⟧ (p_1)(t))))

(BIX1) **BIX** ⟦ **BIND** entity-aspect opt-generic-map-aspect opt-port-map-aspect ⟧ (p_1)(p_2)(t)
= (**BIND** ,entity-aspect,
(null(opt-generic-map-aspect)→ ε,
let generic-map-aspect = opt-generic-map-aspect **in**
GMX ⟦ generic-map-aspect ⟧ (p_1)(p_2)(t)),
(null(opt-port-map-aspect)→ ε,
let port-map-aspect = opt-port-map-aspect **in**
PMX ⟦ port-map-aspect ⟧ (p_1)(p_2)(t)))

7.3.6 Entity Declarations

(ENX1) **ENX** ⟦ **ENTITY** id generic-decl* port-decl* decl* opt-id ⟧ (p)(t)
= insert-phase1-hook
((**ENTITY** ,id,**GDX** ⟦ generic-decl* ⟧ (%(p)(id))(t),
PDX ⟦ port-decl* ⟧ (%(p)(id))(t),**DX** ⟦ decl* ⟧ (%(p)(id))(t),opt-id))
(entity-decl)

7.3.7 Architecture Bodies

(ARX1) **ARX** ⟦ **ARCHITECTURE** id_1 id_2 decl* conc-stat* opt-id ⟧ (using-configuration)(p)(t)
= let $p_1 = \%(\%(p)(id_2))(id_1)$ in
(**ARCHITECTURE** ,id_1,id_2,**DX** ⟦ decl* ⟧ (p_1)(t),
CSX ⟦ conc-stat* ⟧ (using-configuration)(p_1)(t)(tt),opt-id)

7.3.8 Generic Declarations

(GDX0) **GDX** ⟦ ϵ ⟧ (p)(t) = ϵ

(GDX1) **GDX** ⟦ generic-decl generic-decl* ⟧ (p)(t)
= cons(**GDX** ⟦ generic-decl ⟧ (p)(t),**GDX** ⟦ generic-decl* ⟧ (p)(t))

(GDX2) **GDX** ⟦ **DEC GENERIC** id^+ type-mark opt-expr ⟧ (p)(t)
= (**DEC** ,**CONST** ,id^+,type-mark,
let expr = opt-expr in
second(**EX** ⟦ expr ⟧ (p)(t)))

(GDX3) **GDX** ⟦ **SLCDEC GENERIC** id^+ slice-name opt-expr ⟧ (p)(t)
= (**SLCDEC** ,**CONST** ,id^+,
let (type-mark,discrete-range) = slice-name in
(type-mark,**DRX** ⟦ discrete-range ⟧ (p)(t)),
let expr = opt-expr in
second(**EX** ⟦ expr ⟧ (p)(t)))

7.3.9 Port Declarations

(PDX0) **PDX** ⟦ ϵ ⟧ (p)(t) = ϵ

(PDX1) **PDX** ⟦ port-decl port-decl* ⟧ (p)(t)
= cons(**PDX** ⟦ port-decl ⟧ (p)(t),**PDX** ⟦ port-decl* ⟧ (p)(t))

(PDX2) **PDX** ⟦ **DEC PORT** id^+ mode type-mark opt-expr ⟧ (p)(t)
= (**DEC** ,**SIG** ,id^+,type-mark,
let expr = opt-expr in
second(**EX** ⟦ expr ⟧ (p)(t)))

(PDX3) **PDX** ⟦ **SLCDEC PORT** id^+ mode slice-name opt-expr ⟧ (p)(t)
= (**SLCDEC** ,**SIG** ,id^+,
let (type-mark,discrete-range) = slice-name in
(type-mark,**DRX** ⟦ discrete-range ⟧ (p)(t)),
let expr = opt-expr in
second(**EX** ⟦ expr ⟧ (p)(t)))

7.3.10 Generic Maps and Port Maps

(GMX1) **GMX** ⟦ **GENERICMAP** assoc-elt$^+$ ⟧ $(p_1)(p_2)(t)$
= (**GENERICMAP** ,transform-assoc-elts(assoc-elt$^+$)$(p_1)(p_2)(t)$)

(PMX1) **PMX** ⟦ **PORTMAP** assoc-elt$^+$ ⟧ $(p_1)(p_2)(t)$
= (**PORTMAP** ,transform-assoc-elts(assoc-elt$^+$)$(p_1)(p_2)(t)$)

transform-assoc-elts(assoc-elt*)$(p_1)(p_2)(t)$
= (null(assoc-elt*)→ ϵ,
let assoc-elt = hd(assoc-elt*) **in**
let $expr_1$ = hd(assoc-elt)
and $expr_2$ = second(assoc-elt) **in**
cons(cons(second(**EX** ⟦ $expr_1$ ⟧ $(p_1)(t)$),second(**EX** ⟦ $expr_2$ ⟧ $(p_2)(t)$)),
transform-assoc-elts(tl(assoc-elt*))$(p_1)(p_2)(t)$))

7.3.11 Declarations

(DX0) **DX** ⟦ ϵ ⟧ $(p)(t) = \epsilon$

(DX1) **DX** ⟦ decl decl* ⟧ $(p)(t)$ = cons(**DX** ⟦ decl ⟧ $(p)(t)$,**DX** ⟦ decl* ⟧ $(p)(t)$)

(DX2) **DX** ⟦ package-decl package-decl* ⟧ $(p)(t)$
= cons(**DX** ⟦ package-decl ⟧ $(p)(t)$,**DX** ⟦ package-decl* ⟧ $(p)(t)$)

(DX3) **DX** ⟦ package-body package-body* ⟧ $(p)(t)$
= cons(**DX** ⟦ package-body ⟧ $(p)(t)$,**DX** ⟦ package-body* ⟧ $(p)(t)$)

(DX4) **DX** ⟦ use-clause use-clause* ⟧ $(p)(t)$
= cons(**DX** ⟦ use-clause ⟧ $(p)(t)$,**DX** ⟦ use-clause* ⟧ $(p)(t)$)

(DX5) **DX** ⟦ **DEC** object-class id$^+$ type-mark opt-expr ⟧ $(p)(t)$
= (**DEC** ,object-class,id$^+$,type-mark,
let expr = opt-expr **in**
second(**EX** ⟦ expr ⟧ $(p)(t)$))

(DX6) **DX** ⟦ **SLCDEC** object-class id$^+$ slice-name opt-expr ⟧ $(p)(t)$
= (**SLCDEC** ,object-class,id$^+$,
let (type-mark,discrete-range) = slice-name **in**
(type-mark,**DRX** ⟦ discrete-range ⟧ $(p)(t)$),
let expr = opt-expr **in**
second(**EX** ⟦ expr ⟧ $(p)(t)$))

(DX7) **DX** ⟦ **ETDEC** id id$^+$ ⟧ $(p)(t)$ = (**ETDEC** ,id,id$^+$)

(DX8) **DX** ⟦ **ATDEC** id discrete-range type-mark ⟧ $(p)(t)$
= (**ATDEC** ,id,**DRX** ⟦ discrete-range ⟧ $(p)(t)$,type-mark)

(DX9) **DX** ⟦ **PACKAGE** id decl* opt-id ⟧ $(p)(t)$
= (**PACKAGE** ,id,**DX** ⟦ decl* ⟧ $(\%(p)(id))(t)$,opt-id)

(DX10) $\underline{\mathbf{DX}}$ [[**PACKAGEBODY** id decl* opt-id]] (p)(t)
= **let** d = t(p)(id) **in**
let q = %(path(d))(id) **in**
(**PACKAGEBODY** ,id,$\underline{\mathbf{DX}}$ [[decl*]] (q)(t),opt-id)

(DX11) $\underline{\mathbf{DX}}$ [[**PROCEDURE** id proc-par-spec*]] (p)(t)
= **let** d = t(p)(id) **in**
(null(body(d))→ error(cat("**Missing subprogram body:** ")(namef
(d))),
(**PROCEDURE** ,id,proc-par-spec*))

(DX12) $\underline{\mathbf{DX}}$ [[**FUNCTION** id func-par-spec* type-mark]] (p)(t)
= **let** d = t(p)(id) **in**
(null(body(d))→ error(cat("**Missing subprogram body:** ")(namef
(d))),
(**FUNCTION** ,id,func-par-spec*,type-mark))

(DX13) $\underline{\mathbf{DX}}$ [[**SUBPROGBODY** subprog-spec decl* seq-stat* opt-id]] (p)(t)
= **let** (tg,id,par-spec*,type-mark) = subprog-spec **in**
let p_1 = %(p)(id) **in**
(**SUBPROGBODY** ,
let decl = subprog-spec **in**
$\underline{\mathbf{DX}}$ [[decl]] (p)(t),$\underline{\mathbf{DX}}$ [[decl*]] (p_1)(t),$\underline{\mathbf{SSX}}$ [[seq-stat*]] (p_1)(t),opt-id)

(DX14) $\underline{\mathbf{DX}}$ [[**USE** dotted-name+]] (p)(t) = (**USE** ,dotted-name+)

(DX15) $\underline{\mathbf{DX}}$ [[**STDEC** id type-mark opt-discrete-range]] (p)(t)
= (**STDEC** ,id,type-mark,
(null(opt-discrete-range)→ ε,
let (direction,$expr_1$,$expr_2$) = opt-discrete-range **in**
(direction,second($\underline{\mathbf{EX}}$ [[$expr_1$]] (p)(t)),second($\underline{\mathbf{EX}}$ [[$expr_2$]] (p)(t)))))

(DX16) $\underline{\mathbf{DX}}$ [[**ITDEC** id discrete-range]] (p)(t)
= (**ITDEC** ,id,
let (direction,$expr_1$,$expr_2$) = discrete-range **in**
(direction,second($\underline{\mathbf{EX}}$ [[$expr_1$]] (p)(t)),second($\underline{\mathbf{EX}}$ [[$expr_2$]] (p)(t))))

(DX17) $\underline{\mathbf{DX}}$ [[**COMPONENT** id generic-decl* port-decl*]] (p)(t)
= insert-phase1-hook
((**COMPONENT** ,id,$\underline{\mathbf{GDX}}$ [[generic-decl*]] (p)(t),$\underline{\mathbf{PDX}}$ [[port-decl*]] (p)(t)))
(decl)

7.3.12 Concurrent Statements

(CSX0) $\underline{\mathbf{CSX}}$ [[ε]] (using-configuration)(p)(t)(phase1-hook?) = ε

(CSX1) $\underline{\mathbf{CSX}}$ [[conc-stat conc-stat*]] (using-configuration)(p)(t)(phase1-hook?)
= cons($\underline{\mathbf{CSX}}$ [[conc-stat]] (using-configuration)(p)(t)(phase1-hook?),
$\underline{\mathbf{CSX}}$ [[conc-stat*]] (using-configuration)(p)(t)(phase1-hook?))

(CSX2) **CSX** ⟦ **BLOCK** id block-header decl* conc-stat* opt-id ⟧ (using-configuration)(p)(t)(phase1-hook?)
= let p_1 = %(p)(id)
and generic-part = hd(block-header)
and port-part = second(block-header) in
let generic-decl* = (null(generic-part)→ ε, hd(generic-part))
and generic-map-aspect = (null(generic-part)→ ε,
second(generic-part))
and port-decl* = (null(port-part)→ ε, hd(port-part))
and port-map-aspect = (null(port-part)→ ε, second(port-part)) in
let transformed-generic-map = (null(generic-map-aspect)→ ε,
GMX ⟦ generic-map-aspect ⟧ (p_1)(p_1)(t))
and transformed-port-map = (null(port-map-aspect)→ ε,
PMX ⟦ port-map-aspect ⟧ (p_1)(p_1)(t)) in
(phase1-hook? = tt
→ insert-phase1-hook
(accomplish-generic-and-port-maps
(transformed-generic-map)(transformed-port-map)
((**BLOCK** ,id,**DX** ⟦ decl* ⟧ (p_1)(t),
CSX ⟦ conc-stat* ⟧ (using-configuration)(p_1)(t)(phase1-hook?),opt-id)))
(conc-stat),
accomplish-generic-and-port-maps
(transformed-generic-map)(transformed-port-map)
((**BLOCK** ,id,**DX** ⟦ decl* ⟧ (p_1)(t),
CSX ⟦ conc-stat* ⟧ (using-configuration)(p_1)(t)(phase1-hook?),opt-id)))

(CSX3) **CSX** ⟦ **PROCESS** id ref* decl* seq-stat* opt-id ⟧ (using-configuration)(p)(t)(phase1-hook?)
= let p_1 = %(p)(id) in
(phase1-hook? = tt
→ insert-phase1-hook
((**PROCESS** ,id,**DX** ⟦ decl* ⟧ (p_1)(t),
let $seq\text{-}stat_1^*$ = (null(seq-stat*)
→ ((**WAIT** ,(**AT** ,mk-atmark()),ref*,ε,ε)),
(null(ref*)→ seq-stat*,
append
(seq-stat*,
((**WAIT** ,(**AT** ,mk-atmark()),ref*,ε,ε))))) in
SSX ⟦ $seq\text{-}stat_1^*$ ⟧ (p_1)(t),opt-id))(conc-stat),
(**PROCESS** ,id,**DX** ⟦ decl* ⟧ (p_1)(t),
let $seq\text{-}stat_1^*$ = (null(seq-stat*)
→ ((**WAIT** ,(**AT** ,mk-atmark()),ref*,ε,ε)),
(null(ref*)→ seq-stat*,
append
(seq-stat*,
((**WAIT** ,(**AT** ,mk-atmark()),ref*,ε,ε))))) in
SSX ⟦ $seq\text{-}stat_1^*$ ⟧ (p_1)(t),opt-id))

(CSX4) **CSX** ⟦ **SEL-SIGASSN** atmark delay-type id expr ref selected-waveform$^+$ ⟧
(using-configuration)(p)(t)(phase1-hook?)
= let expr* = cons(expr,
collect-expressions-from-selected-waveforms
(selected-waveform$^+$)) in
let ref* = delete-duplicates
(collect-signals-from-expr-list(expr*)(t)(p)(ε)) in
let case-alt$^+$ = construct-case-alternatives
(ref)(delay-type)(selected-waveform$^+$) in
let case-stat = (**CASE** ,atmark,expr,case-alt$^+$) in
let process-stat = (**PROCESS** ,id,ref*,ε,(case-stat),id) in

insert-phase1-hook
(**CSX** ⟦ process-stat ⟧ (using-configuration)(p)(t)(ff))(conc-stat)

(CSX5) **CSX** ⟦ **COND-SIGASSN** atmark delay-type id ref cond-waveform* waveform ⟧
(using-configuration)(p)(t)(phase1-hook?)
= **let** expr* = nconc
(collect-expressions-from-conditional-waveforms
(cond-waveform*),
collect-transaction-expressions(second(waveform))) **in**
let ref* = delete-duplicates
(collect-signals-from-expr-list(expr*)(t)(p)(ε)) **in**
(null(cond-waveform*)
→ **let** sig-assn-stat = (**SIGASSN** ,atmark,delay-type,ref,waveform) **in**
let process-stat = (**PROCESS** ,id,ref*,ε,(sig-assn-stat),id) **in**
insert-phase1-hook
(**CSX** ⟦ process-stat ⟧ (using-configuration)(p)(t)(ff))
(conc-stat),
let cond-part+ = construct-cond-parts
(ref)(delay-type)(cond-waveform*)
and else-part = ((**SIGASSN** ,(**AT** ,mk-atmark()),delay-type,ref,waveform)) **in**
let if-stat = (**IF** ,atmark,cond-part+,else-part) **in**
let process-stat = (**PROCESS** ,id,ref*,ε,(if-stat),id) **in**
insert-phase1-hook
(**CSX** ⟦ process-stat ⟧ (using-configuration)(p)(t)(ff))(conc-stat))

(CSX6) **CSX** ⟦ **COMPINST** id ref opt-generic-map-aspect opt-port-map-aspect ⟧
(using-configuration)(p)(t)(phase1-hook?)
= **let** block-stat = construct-equivalent-nested-block-stat
(conc-stat)(using-configuration)(last(rest(p)))
(last(p)) **in**
(hd(block-stat)= **UNCONFIGURED-COMPONENT**
→ insert-phase1-hook(block-stat)(conc-stat),
insert-phase1-hook
(**CSX** ⟦ block-stat ⟧ (using-configuration)(p)(t)(ff))(conc-stat))

7.3.13 Sensitivity Lists

(SLX0) **SLX** ⟦ ε ⟧ (p)(t) = ε

(SLX1) **SLX** ⟦ ref ref* ⟧ (p)(t) = cons(**SLX** ⟦ ref ⟧ (p)(t),**SLX** ⟦ ref* ⟧ (p)(t))

(SLX2) **SLX** ⟦ **REF** name ⟧ (p)(t)
= **let** expr = ref **in**
second(**EX** ⟦ expr ⟧ (p)(t))

7.3.14 Sequential Statements

(SSX1) **SSX** ⟦ seq-stat seq-stat* ⟧ (p)(t)
= cons(**SSX** ⟦ seq-stat ⟧ (p)(t),**SSX** ⟦ seq-stat* ⟧ (p)(t))

(SSX2) **SSX** ⟦ **NULL** atmark ⟧ (p)(t) = (**NULL** ,atmark)

(SSX3) $\underline{\mathbf{SSX}}$ [[**VARASSN** atmark ref expr]] (p)(t)
= (**VARASSN** ,atmark,
let expr_0 = ref **in**
second($\underline{\mathbf{EX}}$ [[expr_0]] (p)(t)),second($\underline{\mathbf{EX}}$ [[expr]] (p)(t)))

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

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

transform-if(cond-part*)(p)(t)
= (null(cond-part*)$\rightarrow \varepsilon$,
let (expr,seq-stat*) = hd(cond-part*) **in**
cons((second($\underline{\mathbf{EX}}$ [[expr]] (p)(t)),$\underline{\mathbf{SSX}}$ [[seq-stat*]] (p)(t)),
transform-if(tl(cond-part*))(p)(t)))

(SSX6) $\underline{\mathbf{SSX}}$ [[**CASE** atmark expr case-alt$^+$]] (p)(t)
= (**CASE** ,atmark,second($\underline{\mathbf{EX}}$ [[expr]] (p)(t)),$\underline{\mathbf{AX}}$ [[case-alt$^+$]] (p)(t))

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

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

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

(SSX10) $\underline{\mathbf{SSX}}$ [[**EXIT** atmark opt-dotted-name opt-expr]] (p)(t)
= (**EXIT** ,atmark,opt-dotted-name,
let expr = opt-expr **in**
second($\underline{\mathbf{EX}}$ [[expr]] (p)(t)))

(SSX11) $\underline{\mathbf{SSX}}$ [[**CALL** atmark ref]] (p)(t)
= (**CALL** ,atmark,
let expr = ref **in**
second($\underline{\mathbf{EX}}$ [[expr]] (p)(t)))

(SSX12) $\underline{\mathbf{SSX}}$ [[**RETURN** atmark opt-expr]] (p)(t)
= (**RETURN** ,atmark,
let expr = opt-expr **in**
second($\underline{\mathbf{EX}}$ [[expr]] (p)(t)))

(SSX13) $\underline{\mathbf{SSX}}$ [[**WAIT** atmark ref* opt-expr_1 opt-expr_2]] (p)(t)
= **let** expr_1 = opt-expr_1
and expr_2 = opt-expr_2 **in**
(**WAIT** ,atmark,$\underline{\mathbf{MEX}}$ [[ref*]] (p)(t),second($\underline{\mathbf{EX}}$ [[expr_1]] (p)(t)),
second($\underline{\mathbf{EX}}$ [[expr_2]] (p)(t)))

7.3.15 Case Alternatives

(AX0) $\underline{\mathbf{AX}}$ ⟦ ε ⟧ (p)(t) = ε

(AX1) $\underline{\mathbf{AX}}$ ⟦ case-alt case-alt* ⟧ (p)(t)
= cons($\underline{\mathbf{AX}}$ ⟦ case-alt ⟧ (p)(t),$\underline{\mathbf{AX}}$ ⟦ case-alt* ⟧ (p)(t))

(AX2) $\underline{\mathbf{AX}}$ ⟦ **CASECHOICE** discrete-range$^+$ seq-stat* ⟧ (p)(t)
= (**CASECHOICE** ,$\underline{\mathbf{DRX}}$ ⟦ discrete-range$^+$ ⟧ (p)(t),$\underline{\mathbf{SSX}}$ ⟦ seq-stat* ⟧ (p)(t))

(AX3) $\underline{\mathbf{AX}}$ ⟦ **CASEOTHERS** seq-stat* ⟧ (p)(t) = (**CASEOTHERS** ,$\underline{\mathbf{SSX}}$ ⟦ seq-stat* ⟧ (p)(t))

7.3.16 Discrete Ranges

(DRX0) $\underline{\mathbf{DRX}}$ ⟦ ε ⟧ (p)(t) = ε

(DRX1) $\underline{\mathbf{DRX}}$ ⟦ discrete-range discrete-range* ⟧ (p)(t)
= cons($\underline{\mathbf{DRX}}$ ⟦ discrete-range ⟧ (p)(t),$\underline{\mathbf{DRX}}$ ⟦ discrete-range* ⟧ (p)(t))

(DRX2) $\underline{\mathbf{DRX}}$ ⟦ discrete-range ⟧ (p)(t)
= **let** (direction,expr_1 ,expr_2) = discrete-range **in**
(direction,second($\underline{\mathbf{EX}}$ ⟦ expr_1 ⟧ (p)(t)),second($\underline{\mathbf{EX}}$ ⟦ expr_2 ⟧ (p)(t)))

7.3.17 Waveforms and Transactions

(WX1) $\underline{\mathbf{WX}}$ ⟦ **WAVE** transaction$^+$ ⟧ (p)(t) = (**WAVE** ,$\underline{\mathbf{TRX}}$ ⟦ transaction$^+$ ⟧ (p)(t))

(TRX1) $\underline{\mathbf{TRX}}$ ⟦ transaction transaction* ⟧ (p)(t)
= (null(transaction*)→ ($\underline{\mathbf{TRX}}$ ⟦ transaction ⟧ (p)(t)),
let transaction_1^+ = transaction* **in**
cons($\underline{\mathbf{TRX}}$ ⟦ transaction ⟧ (p)(t),$\underline{\mathbf{TRX}}$ ⟦ transaction_1^+ ⟧ (p)(t)))

(TRX2) $\underline{\mathbf{TRX}}$ ⟦ **TRANS** expr opt-expr ⟧ (p)(t)
= (**TRANS** ,second($\underline{\mathbf{EX}}$ ⟦ expr ⟧ (p)(t)),
let expr_1 = opt-expr **in**
second($\underline{\mathbf{EX}}$ ⟦ expr_1 ⟧ (p)(t)))

7.3.18 Expressions

(MEX0) $\underline{\mathbf{MEX}}$ ⟦ ε ⟧ (p)(t) = ε

(MEX1) $\underline{\mathbf{MEX}}$ ⟦ ref ref* ⟧ (p)(t) = cons(second($\underline{\mathbf{EX}}$ ⟦ ref ⟧ (p)(t)),$\underline{\mathbf{MEX}}$ ⟦ ref* ⟧ (p)(t))

(EX0) $\underline{\mathbf{EX}}$ ⟦ ε ⟧ (p)(t) = (void-type-desc(t),ε)

(EX1) $\underline{\mathbf{EX}}$ ⟦ **FALSE** ⟧ (p)(t) = (bool-type-desc(t),(**FALSE**))

(EX2) $\underline{\mathbf{EX}}$ ⟦ **TRUE** ⟧ (p)(t) = (bool-type-desc(t),(**TRUE**))

(EX3) $\underline{\mathbf{EX}}$ ⟦ **BIT** bitlit ⟧ (p)(t) = (bit-type-desc(t),(**BIT** ,bitlit))

(EX4) $\underline{\mathbf{EX}}$ ⟦ **NUM** constant ⟧ (p)(t) = (int-type-desc(t),(**NUM** ,constant))

```
(EX5) EX [[ TIME constant time-unit ]] (p)(t)
      = let normalized-constant = (case time-unit
                                     FS → N [[ constant ]],
                                     PS → 1000×N [[ constant ]],
                                     NS → 1000000×N [[ constant ]],
                                     US → 1000000000×N [[ constant ]],
                                     MS → 1000000000000×N [[ constant ]],
                                     SEC → 1000000000000000×N [[ constant ]],
                                     MIN → 60×(1000000000000000×N [[ constant ]]),
                                     HR → 3600×(1000000000000000×N [[ constant ]]),
                                     OTHERWISE
                                     → error
                                          (cat("Illegal unit name for physical type TIME: ")
                                           (time-unit))) in
        (time-type-desc(t),(TIME ,normalized-constant,FS ))

(EX6) EX [[ CHAR constant ]] (p)(t)
      = let d = lookup(t)((STANDARD) )(expr) in
        (type(d),(CHAR ,constant))

(EX7) EX [[ BITSTR bit-lit* ]] (p)(t) = (ε,(BITSTR ,bit-lit*))

(EX8) EX [[ STR char-lit* ]] (p)(t) = (ε,(STR ,char-lit*))

(EX9) EX [[ REF name ]] (p)(t) = transform-name(name)(ε)(ε)(p)(t)

transform-name(name)(w)(ast0*)(p)(t)
= (null(w)
   → let w1 = lookup2(t)(p)(ε)(hd(name)) in
     (w1 = *UNBOUND*
      → error
           (cat("Unbound identifier in auxiliary semantic function TRANSFORM-NAME: ")
            ($(p)(hd(name)))),
      (second(tmode(w1))= TYP → transform-name(tl(name))(w1)(ε)(p)(t),
       transform-name
         (tl(name))(w1)(((SREF ,path(tdesc(w1)),idf(tdesc(w1)))))(p)(t))),
   let d = tdesc(w)
       and tm = tmode(w) in
    let tg = tag(d) in
     (null(name)
      → (second(tm)= TYP → transform-name-aux(*CONVERSION* )(d)(ast0*),
         transform-name-aux(tg)(d)(ast0*)),
      let x = hd(name) in
       (consp(x)
        → let ast1* = transform-list(x)(p)(t) in
            (second(tm)= TYP
             → transform-name
                  (tl(name))(w)((TYPECONV ,hd(ast1*),%(path(d))(idf(d))))(p)(t),
             second(tm)= OBJ ∧ is-array-tdesc?(d)
             → transform-name
                  (tl(name))(mk-type(tm)(elty(d)))
                  (nconc(ast0*,((INDEX ,hd(ast1*)))))(p)(t),
             (second(tm)= OBJ ∧ is-array?(type(d)))
                      ∨ (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*0,((INDEX ,hd(ast*1)))))(p)(t),
            transform-name
              (tl(name))(extract-rtype(d))
              (nconc(ast*0,((PARLIST ,ast*1))))(p)(t)),
        ((second(tm)= OBJ ∧ is-record?(type(d)))
            ∨ (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
             transform-name
               (tl(name))(mk-type(tm)(d2))(nconc(ast*0,((SELECTOR ,x))))
               (p)(t),
        second(tm)= OBJ ∧ is-record-tdesc?(d)
        → let d2 = lookup-record-field(components(d))(x) in
            transform-name
              (tl(name))(mk-type(tm)(d2))(nconc(ast*0,((SELECTOR ,x))))
              (p)(t),
        let w1 = lookup-local(x)(%(path(d))(idf(d)))(p)(t) in
         (w1 = *UNBOUND*
          → error
              (cat("Unknown identifier in function TRANSFORM-NAME: ")
               ($(%(path(d))(idf(d)))(x))),
          transform-name
            (tl(name))(w1)(((SREF ,path(tdesc(w1)),idf(tdesc(w1)))))(p)
            (t))))))

transform-name-aux(tg)(d)(ast)
= (case tg
     *OBJECT* → (second(type(d)),(REF ,ast)),
     *ENUMELT* → (second(type(d)),(ENUMLIT ,idf(d))),
     (*PROCEDURE* ,*FUNCTION* )
     → (second(rtype(hd(signatures(d)))),
        (REF ,nconc(ast,((PARLIST ,ε))))),
     *CONVERSION* → (d,ast),
     *PACKAGE* → (d,(REF ,ast)),
     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
                (cat("Unrecognized Stage 4 VHDL unary operator: ")(unary-op)))
(EX12) EX [[ binary-op expr1 expr2 ]] (p)(t)
       = let (d1,e1) = EX [[ expr1 ]] (p)(t) in
           let (d2,e2) = EX [[ expr2 ]] (p)(t) in
            (d1,(scalar-op(binary-op)(d1),e1,e2))
(EX13) EX [[ relational-op expr1 expr2 ]] (p)(t)
       = let (d1,e1) = EX [[ expr1 ]] (p)(t) in
           let (d2,e2) = EX [[ expr2 ]] (p)(t) in
            (bool-type-desc(t),(scalar-op(relational-op)(d1),e1,e2))
scalar-op(op)(d)
= (is-bit-tdesc?(d)∨ is-bitvector-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 4 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 4 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.

8.1 Phase 2 Semantic Domains 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 4 VHDL language constructs to which they assign meanings) are: **DF** (design files), **DU** (design units), **CI** (context items), **LU** (library-units), **CF** (configuration declarations), **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 state deltas. Moreover, **E** and **R** also return a translated expression and list of guard formulas. Guard formulas are inserted in the precondition of generated state deltas to ensure that certain conditions are met in the proof in which the state deltas 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 state delta/assertion lists correspond to non-error final answers.

When state deltas 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 state deltas. In the absence of errors, the item appended represents a list of state deltas. Such a continuation is evaluated and applied only when the state delta 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 state deltas is applied determines which continuation is evaluated and applied to an execution state, and therefore which additional state deltas are subsequently generated.

8.1.1 Phase 2 Semantic Domains

The semantic domains and function types for Phase 2 of the Stage 4 VHDL translator are as follows.

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**)
ast : ASyn	abstract syntax trees
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* + ({CONST, VAR, SIG, DUMMY} × {VAL, OUT, REF, OBJ, ACC, TYP})	type modes
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

8.1.2 Phase 2 Semantic Functions

The semantic functions for Phase 2 of the Stage 4 VHDL translator are as follows.

DF :	**Design → TEnv → Id → Ans**	design file dynamic semantics
DU :	**DUnit* → Asyn → TEnv → Path → Dc → Dc**	design unit dynamic semantics

CI :	**CItem* → TEnv → Path → Dc → Dc**	context item dynamic semantics
LU :	**LUnit → Asyn → TEnv → Path → Dc → Dc**	library unit dynamic semantics
CF :	**Config → TEnv → Path → Dc → Dc**	configuration declaration dynamic semantics
EN :	**Ent → TEnv → Path → Dc → Dc**	entity declaration dynamic semantics
AR :	**Arch → TEnv → Path → Dc → Dc**	architecture body dynamic semantics
D :	**Dec* → TEnv → Path → Dc → Dc**	declaration dynamic semantics
CS :	**CStat* → TEnv → Path → Dc → Dc**	concurrent statement dynamic semantics
SS :	**SStat* → TEnv → Path → Sc → Sc**	sequential statement dynamic semantics
W :	**Wave → TEnv → Path → Wc → Sc**	waveform dynamic semantics
TRM :	**Trans* → TEnv → Path → Wc → Sc**	transaction list dynamic semantics
TR :	**Trans → TEnv → Path → Tc → Sc**	transaction dynamic semantics
ME :	**Expr* → TEnv → Path → Mc → Sc**	expression list dynamic semantics (*l-values*)
MR :	**Expr* → TEnv → Path → Mc → Sc**	expression list dynamic semantics (*r-values*)
E :	**Expr → TEnv → Path → Ec → Sc**	expression dynamic semantics (*l-values*)
R :	**Expr → TEnv → Path → Ec → Sc**	expression dynamic semantics (*r-values*)
T :	**Expr → TEnv → Path → Desc**	expression types
B :	**BitLit → Bit**	bit values of bit literals (primitive)
N :	**NumLit → N**	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 4 VHDL translator certainly do not have a scoping structure that naturally parallels the scopes of their corresponding Stage 4 VHDL hardware description. Furthermore, even if there were such a correspondence between source (Stage 4 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 4 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 4 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 4 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\u\n**, where **z** is the name of the main program and **n** is the current universe's ordinal number (**n** = 1, 2, ...).

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 4 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 4 VHDL hardware description.

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

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))))
```

```
push-var(v)
= let n = next-var(second(v)) in
    (hd(v),n,cons(n,third(v)))

next-var(n)
= (numberp(n)→ n+1,
   (symbolp(n)→ mk-exp2(ADD ,n,1),
    let m = third(n) in
     (numberp(m)→ mk-exp2(ADD ,second(n),m+1),
      mk-exp2(ADD ,second(n),mk-exp2(ADD ,m,1)))))

init-var(suqn) = (suqn,1,(1))

pop-universe(v)(suqn*)
= let ustack = tl(universe-stack(v)) in
    let uname = hd(ustack) in
     make-universe-data
       (uname,universe-counter(v),ustack,
       pop-universe-vars(suqn*)(universe-vars(v)))

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

pop-var(v) = (hd(v),second(v),tl(third(v)))

get-qualified-ids(suqn*)(v)
= (null(suqn*)→ ε,
   cons(qualified-id(hd(suqn*))(v),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)))
```

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 :

$$< \textbf{*STKBOTTOM*}, \textbf{id}, \epsilon, \epsilon >$$

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 4 VHDL design file.

package body exit :

$$< \textbf{*PACKAGE-BODY-EXIT*}, \textbf{id}, \textbf{p}, \textbf{u} >$$

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 :

$$< \textbf{*SUBPROGRAM-RETURN*}, \textbf{id}, \textbf{p}, \textbf{c} >$$

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 :

$$< \textbf{*LOOP-EXIT*, id, p, c} >$$

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

block exit :

$$< \textbf{*BLOCK-EXIT*, id, p, c} >$$

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

begin marker :

$$< \textbf{*BEGIN*, id, p, c} >$$

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

undeclaration :

$$< \textbf{*UNDECLARE*, id}^{+}\textbf{, p, g} >$$

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 $\textbf{id}^{+}$ names the objects to be undeclared, and **g**: $\textbf{Sc} \rightarrow \textbf{Sc}$ is a continuation transformer which, after carrying out the explicit undeclaration specified in **g** (thereby popping this ***UNDECLARE*** descriptor from the execution stack), continues execution by means of 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 4 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.

find-configuration-abstract-syntax

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

init-array-signal-to

init-array-signal-downto

mk-element-waves-aux

get-loop-enum-param-vals

eval-expr

8.3.2 Constructing State Deltas

The construction of state deltas is specified via functions **mk-sd(z)(pre, comod, mod, post)** and **mk-sd-decl(z)(pre, comod, mod, post)**, which take five arguments: the design file name **z** (if **p** is the current path, this is always **hd(p)**) and representations of the precondition, comodification list, modification list, and postcondition of the state delta to be constructed.

These functions are used to represent the construction of state deltas without specifying their exact representation, which is SDVS-dependent and not given here. The pre- and postconditions of a state delta are *lists* of formulas, each of which represents a formula that is the logical *conjunction* of the formulas in this list. If the precondition and comod list arguments of **mk-sd** and **mk-sd-decl** are ϵ, then the precondition and comod list of the constructed state delta are **(TRUE)** and **(ALL)**, respectively. Otherwise, the given arguments are used directly in the state delta. The postcondition may contain a state delta, which is usually represented as a statement continuation applied to an execution stack.

mk-sd and **mk-sd-decl** are almost the same, the only difference being that a state delta created by **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* state delta is **(ALL)** and the mod list of *every* state delta must be *nonempty*. To ensure that a state delta's mod list is never empty, **mk-sd(z)(...)** will *always* prefix **z\pc** to its mod list argument, where **z\pc** is a unique place (represented by a system identifier) in which **z** is the name of the Stage 4 VHDL hardware description being translated. This unique place is the name of a *program counter* whose value implicitly changes when *any* state delta is applied. This program counter place does not make any other kind of appearance in a translated Stage 4 VHDL hardware description.

The notation of state deltas requires that certain symbols sometimes be prefixed to uniquely qualified names: the dot (.) and pound (#) symbols. The functions **dot** and **pound**, applied to uniquely qualified names, accomplish this.

dot(placename) = (**DOT** ,placename)
pound(placename) = (**POUND** ,placename)

Finally, the two functions **fixed-characterized-sds** and **subst-vars** are employed by the Phase 2 semantics of procedure calls to implement the SDVS *offline characterization* mechanism [20, 21], which will be incorporated in Stage 4 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 4 VHDL translator — *state delta generation* — is specified denotationally.

8.4.1 Stage 4 VHDL Design Files

```
(DF1) DF ⟦ DESIGN-FILE id design-unit+ ⟧ (t)(using-configuration)
      = let p0 = %(ε)(id)
            and configuration-ast = (null(using-configuration)→ ε,
                                     find-configuration-abstract-syntax
                                       (design-unit+)(using-configuration)) in
        let v = mk-initial-universe(id)
            and stk = (<*STKBOTTOM* ,id,ε,ε>) in
          (mk-disjoint(id,(dot(id))),
           mk-cover(dot(id),(catenate(id,"\pc"),VHDLTIME ,
                    VHDLTIME_PREVIOUS )),
           mk-scalar-decl(VHDLTIME ,(TYPE VHDLTIME) ),
           mk-scalar-decl(VHDLTIME_PREVIOUS ,(TYPE VHDLTIME) ),
           mk-rel(vhdltime-type-desc(t))((EQ ,dot(VHDLTIME ),mk-vhdltime(0)(0))),
           mk-rel
             (vhdltime-type-desc(t))((EQ ,dot(VHDLTIME_PREVIOUS ),mk-vhdltime(0)(0))),
           mk-decl-sd(id)(ε)(ε)(ε)(u1(v)(stk)))
           where
           u1 = λv,stk.
                  DU ⟦ design-unit+ ⟧ (configuration-ast)(t)(p0)(u2)(v)(stk)
           where u2 = λv,stk.block-exit(v)(stk)

mk-disjoint(id,lst) = cons(ALLDISJOINT ,cons(id,lst))

mk-cover(id,lst) = cons(COVERING ,cons(id,lst))

mk-scalar-decl(placename,place-type) = (DECLARE ,placename,place-type)

vhdltime-type-desc(t) = t((STANDARD) )(VHDLTIME )

mk-rel(d)(op,e1,e2)
= let tg = tag(d) in
    (case tg
       (*BOOL* ,*BIT* ,*INT* ,*REAL* ,*TIME* ,*VHDLTIME* ,*ENUMTYPE* ,*VOID* ,*POLY* )
       → mk-scalar-rel(tg)((op,e1,e2)),
       *SUBTYPE* → mk-scalar-rel(tag(base-type(d)))((op,e1,e2)),
       *INT_TYPE* → mk-scalar-rel(tag(parent-type(d)))((op,e1,e2)),
       *WAVE* → (EQ ,e1,e2),
       *ARRAYTYPE*
       → (is-bitvector-tdesc?(d)
          → (case op
               EQ
               → (is-constant-bitvector?(e1)∧ is-constant-bitvector?(e2)
                  → (EQ ,cons(USCONC ,e1),cons(USCONC ,e2)),
                  is-constant-bitvector?(e2)→ (EQ ,e1,cons(USCONC ,e2)),
                  is-constant-bitvector?(e1)→ (EQ ,cons(USCONC ,e1),e2),
                  (EQ ,e1,e2)),
               NE
```

```
        → (is-constant-bitvector?(e1)∧ is-constant-bitvector?(e2)
            → (NEQ ,cons(USCONC ,e1),cons(USCONC ,e2)),
            is-constant-bitvector?(e2)→ (NEQ ,e1,cons(USCONC ,e2)),
            is-constant-bitvector?(e1)→ (NEQ ,cons(USCONC ,e1),e2),
            (NEQ ,e1,e2)),
        LT
        → (EQ ,(BS ,1,1),
            (is-constant-bitvector?(e1)∧ is-constant-bitvector?(e2)
             → (USLSS ,cons(USCONC ,e1),cons(USCONC ,e2)),
             is-constant-bitvector?(e2)→ (USLSS ,e1,cons(USCONC ,e2)),
             is-constant-bitvector?(e1)→ (USLSS ,cons(USCONC ,e1),e2),
             (USLSS ,e1,e2))),
        LE
        → (EQ ,(BS ,1,1),
            (is-constant-bitvector?(e1)∧ is-constant-bitvector?(e2)
             → (USLEQ ,cons(USCONC ,e1),cons(USCONC ,e2)),
             is-constant-bitvector?(e2)→ (USLEQ ,e1,cons(USCONC ,e2)),
             is-constant-bitvector?(e1)→ (USLEQ ,cons(USCONC ,e1),e2),
             (USLEQ ,e1,e2))),
        GT
        → (EQ ,(BS ,1,1),
            (is-constant-bitvector?(e1)∧ is-constant-bitvector?(e2)
             → (USGTR ,cons(USCONC ,e1),cons(USCONC ,e2)),
             is-constant-bitvector?(e2)→ (USGTR ,e1,cons(USCONC ,e2)),
             is-constant-bitvector?(e1)→ (USGTR ,cons(USCONC ,e1),e2),
             (USGTR ,e1,e2))),
        GE
        → (EQ ,(BS ,1,1),
            (is-constant-bitvector?(e1)∧ is-constant-bitvector?(e2)
             → (USGEQ ,cons(USCONC ,e1),cons(USCONC ,e2)),
             is-constant-bitvector?(e2)→ (USGEQ ,e1,cons(USCONC ,e2)),
             is-constant-bitvector?(e1)→ (USGEQ ,cons(USCONC ,e1),e2),
             (USGEQ ,e1,e2))),
        OTHERWISE → impl-error("Shouldn't happen!")),
    is-string-tdesc?(d)
    → (case op
        EQ
        → (is-constant-string?(e1)∧ is-constant-string?(e2)
            → (EQ ,cons(ACONC ,e1),cons(ACONC ,e2)),
            is-constant-string?(e2)→ (EQ ,e1,cons(ACONC ,e2)),
            is-constant-string?(e1)→ (EQ ,cons(ACONC ,e1),e2),
            (EQ ,e1,e2)),
        NE
        → (is-constant-string?(e1)∧ is-constant-string?(e2)
            → (NEQ ,cons(ACONC ,e1),cons(ACONC ,e2)),
            is-constant-string?(e2)→ (NEQ ,e1,cons(ACONC ,e2)),
            is-constant-string?(e1)→ (NEQ ,cons(ACONC ,e1),e2),
            (NEQ ,e1,e2)),
        OTHERWISE → impl-error("Shouldn't happen!")),
    (case op
      EQ
      → (dotted-expr-p(e2)→ (EQ ,e1,e2), impl-error("Shouldn't happen!")),
      NE
      → (dotted-expr-p(e2)→ (NEQ ,e1,e2),
          impl-error("Shouldn't happen!")),
      OTHERWISE → impl-error("Shouldn't happen!"))),
*RECORDTYPE*
```

```
      → (dotted-expr-p(e2)→ (EQ ,e1,e2), impl-error("Shouldn't happen!")),
      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 )

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 ,(EQ ,e1,TRUE ),(EQ ,e2,FALSE )),
          GE → (IMPLIES ,e2,e1),
          OTHERWISE
          → impl-error
               ("Unrecognized Stage 4 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 4 VHDL BIT relational operator: ~a",
               relational-op)),
     (*INT* ,*TIME* )
     → (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 4 VHDL INTEGER 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 4 VHDL VHDLTIME 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 4 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 4 VHDL ENUMERATION relational operator: ~a",
            relational-op)),
    *VOID*
    → (case relational-op
        EQ → (EQ ,e1,e2),
        NE → (NEQ ,e1,e2),
        OTHERWISE
        → impl-error
            ("Unrecognized Stage 4 VHDL VOID relational operator: ~a",
            relational-op)),
    *POLY*
    → (case relational-op
        EQ → (EQ ,e1,e2),
        NE → (NEQ ,e1,e2),
        OTHERWISE
        → impl-error
            ("Unrecognized Stage 4 VHDL POLYMORPHIC relational operator: ~a",
            relational-op)),
    OTHERWISE → impl-error("Unsupported Stage 4 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 )),
      (COND ,e1,e2,(NOT ,e2))),
   (NEQ ,e1,e2))

simple-term(term)
= let operators = (DOT POUND) in
   ¬consp(term)∨ hd(term)∈ operators

mk-vhdltime(global)(delta) = (VHDLTIME ,global,delta)

block-exit(v)(stk)
= let <tg,qname,p,g> = hd(stk) in
   (case tg
      *STKBOTTOM* → model-execution-complete(qname),
      *UNDECLARE* → g(λvv,s.block-exit(vv)(s))(v)(stk),
      (*BLOCK-EXIT* ,*SUBPROGRAM-RETURN* ) → g(v)(stk-pop(stk)),
      (*BEGIN* ,*LOOP-EXIT* ,*PACKAGE-BODY-EXIT* ) → block-exit(v)(stk-pop(stk)),
      OTHERWISE
      → impl-error("Unknown execution stack descriptor with tag: ~a",tg))

model-execution-complete(id)
= (mk-sd(id)(ε)(ε)(ε)(((VHDL_MODEL_EXECUTION_COMPLETE ,id))))
```

A Stage 4 VHDL design file has a name **id** — supplied as an argument to the SDVS command *vhdltr* — and consists of a nonempty sequence of design units.

The semantics of the design file has two semantic arguments: the TSE **t** constructed by Phase 1, and an identifier **using-configuration** supplied to the *vhdltr* command that specifies the configuration declaration to be used in configuring the design entity (in the absence of such a configuration, this identifier is expected to be **none**).

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 **mk-disjoint(place-list)** generates an SDVS assertion stating that the places in **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 **id\pc** as well as the places **vhdltime** and **vhdltime_previous**.

The function **mk-cover(place, place-list)**[2] generates an SDVS *covering* assertion that **place** covers all the places in **place-list** and that all of the places in **place-list** are mutually disjoint.

- *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(0,0)** 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 a state delta that defines the execution of the hardware description is generated. The application of this state delta leads to further usable state deltas, 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 state deltas symbolically elaborate the design file's design units.

8.4.2 Design Units

(DU0) $\underline{\mathbf{DU}}$ ⟦ ϵ ⟧ (configuration-ast)(t)(p)(u)(v)(stk) = u(v)(stk)

(DU1) $\underline{\mathbf{DU}}$ ⟦ design-unit design-unit* ⟧ (configuration-ast)(t)(p)(u)(v)(stk)
= $\underline{\mathbf{DU}}$ ⟦ design-unit ⟧ (configuration-ast)(t)(p)(u_1)(v)(stk)
where
$u_1 = \lambda v_1, stk_1.$
$\underline{\mathbf{DU}}$ ⟦ design-unit* ⟧ (configuration-ast)(t)(p)(u)(v_1)(stk_1)

(DU2) $\underline{\mathbf{DU}}$ ⟦ **DESIGN-UNIT** context-item* library-unit ⟧ (configuration-ast)(t)(p)(u)(v)(stk)
= $\underline{\mathbf{CI}}$ ⟦ context-item* ⟧ (t)(rest(p))(u_1)(v)(stk)
where
$u_1 = \lambda v_1, stk_1.$
$\underline{\mathbf{LU}}$ ⟦ library-unit ⟧ (configuration-ast)(t)(p)(u)(v_1)(stk_1)

8.4.3 Contex Items

(CI0) $\underline{\mathbf{CI}}$ ⟦ ϵ ⟧ (t)(p)(u)(v)(stk) = u(v)(stk)

(CI1) $\underline{\mathbf{CI}}$ ⟦ context-item context-item* ⟧ (t)(p)(u)(v)(stk)
= $\underline{\mathbf{CI}}$ ⟦ context-item ⟧ (t)(p)(u_1)(v)(stk)
where $u_1 = \lambda v_1, stk_1.\underline{\mathbf{CI}}$ ⟦ context-item* ⟧ (t)(p)(u)(v_1)(stk_1)

(CI2) $\underline{\mathbf{CI}}$ ⟦ **USE** dotted-name$^+$ ⟧ (t)(p)(u)(v)(stk)
= **let** decl = context-item **in**
$\underline{\mathbf{D}}$ ⟦ decl ⟧ (t)(p)(u)(v)(stk)

[2]The function **mk-cover** has in some instances been superseded by **mk-cover-already**; it implements an experimental new naming scheme for VHDL variables. The scheme is available only when the SDVS function **new-declarations** is defined to return non-NIL. In SDVS Version 12, this new scheme is not available, so we will not discuss the actions of this function here.

8.4.4 Library Units

(LU1) $\underline{\text{LU}}$ ⟦ **CONFIGURATION** id_1 id_2 use-clause* block-config opt-id ⟧ (configuration-ast)(t)(p)(u)(v)(stk)
= **let** configuration-decl = library-unit **in**
$\underline{\text{CF}}$ ⟦ configuration-decl ⟧ (t)(p)(u)(v)(stk)

(LU2) $\underline{\text{LU}}$ ⟦ **PACKAGE** id decl* opt-id ⟧ (configuration-ast)(t)(p)(u)(v)(stk)
= **let** decl = library-unit **in**
$\underline{\text{D}}$ ⟦ decl ⟧ (t)(p)(u)(v)(stk)

(LU3) $\underline{\text{LU}}$ ⟦ **ENTITY** id $decl_1^*$ $decl_2^*$ $decl_3^*$ opt-id phase1-hook ⟧ (configuration-ast)(t)(p)(u)(v)(stk)
= (null(configuration-ast)
→ **let** entity-decl = library-unit **in**
$\underline{\text{EN}}$ ⟦ entity-decl ⟧ (t)(p)(u)(v)(stk),
let configuration-entity-id = get-configuration-entity-id
(configuration-ast) **in**
(id = configuration-entity-id
→ **let** entity-decl = library-unit **in**
$\underline{\text{EN}}$ ⟦ entity-decl ⟧ (t)(p)(u)(v)(stk),
u(v)(stk)))

(LU4) $\underline{\text{LU}}$ ⟦ **PACKAGEBODY** id decl* opt-id ⟧ (configuration-ast)(t)(p)(u)(v)(stk)
= **let** decl = library-unit **in**
$\underline{\text{D}}$ ⟦ decl ⟧ (t)(p)(u)(v)(stk)

(LU5) $\underline{\text{LU}}$ ⟦ **ARCHITECTURE** id_1 id_2 decl* conc-stat* opt-id ⟧ (configuration-ast)(t)(p)(u)(v)(stk)
= (null(configuration-ast)
→ **let** architecture-body = library-unit **in**
$\underline{\text{AR}}$ ⟦ architecture-body ⟧ (t)(p)(u)(v)(stk),
let configuration-entity-id = get-configuration-entity-id
(configuration-ast)
and configuration-architecture-id = get-configuration-architecture-id
(configuration-ast) **in**
(id_2 = configuration-entity-id ∧ id_1 = configuration-architecture-id
→ **let** architecture-body = library-unit **in**
$\underline{\text{AR}}$ ⟦ architecture-body ⟧ (t)(p)(u)(v)(stk),
u(v)(stk)))

get-configuration-entity-id(configuration-ast) = hd(tl(tl(configuration-ast)))

get-configuration-architecture-id(configuration-ast)
= hd(tl(hd(tl(tl(tl(tl(tl(configuration-ast))))))))

8.4.5 Configuration Declarations

(CF1) $\underline{\text{CF}}$ ⟦ **CONFIGURATION** id_1 id_2 use-clause* block-config opt-id ⟧ (t)(p)(u)(v)(stk)
= u(v)(stk)

8.4.6 Entity Declarations

(EN1) $\underline{\mathbf{EN}}$ ⟦ **ENTITY** id decl_1^* decl_2^* decl_3^* opt-id phase1-hook ⟧ (t)(p)(u)(v)(stk)
= **let** $p_1 = \%(p)(id)$ **in**
$\underline{\mathbf{D}}$ ⟦ decl_1^* ⟧ (t)(p_1)(u_1)(v)(stk)
where
$u_1 = \lambda v_1, stk_1.$
$\underline{\mathbf{D}}$ ⟦ decl_2^* ⟧ (t)(p_1)(u_2)(v_1)(stk_1)
where $u_2 = \lambda v_2, stk_2.\underline{\mathbf{D}}$ ⟦ decl_3^* ⟧ (t)(p_1)(u)(v_2)(stk_2)

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**.

8.4.7 Architecture Bodies

(AR1) $\underline{\mathbf{AR}}$ ⟦ **ARCHITECTURE** id_1 id_2 decl^* conc-stat^* opt-id ⟧ (t)(p)(u)(v)(stk)
= **let** $p_1 = \%(\%(p)(id_2))(id_1)$ **in**
$\underline{\mathbf{D}}$ ⟦ decl^* ⟧ (t)(p_1)(u_1)(v)(stk)
where
$u_1 = \lambda v_1, stk_1.$
$\underline{\mathbf{CS}}$ ⟦ conc-stat^* ⟧ (t)(p_1)(u_2)(v_1)(stk_1)
where
$u_2 = \lambda v_2, stk_2.$
cons((**VHDL_MODEL_ELABORATION_COMPLETE** ,hd(p)),
(mk-sd
(hd(p))(ε)(ε)(ε)
((make-vhdl-begin-model-execution
(hd(p))(u)(t)(v_2)(stk_2)))))

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 `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 Simplifier 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.

8.4.8 Declarations

(D0) $\underline{\mathbf{D}}$ ⟦ ε ⟧ (t)(p)(u)(v)(stk) = u(v)(stk)

(D1) $\underline{\mathbf{D}}$ ⟦ decl decl^* ⟧ (t)(p)(u)(v)(stk)
= $\underline{\mathbf{D}}$ ⟦ decl ⟧ (t)(p)(u_1)(v)(stk)
where $u_1 = \lambda v_1, stk_1.\underline{\mathbf{D}}$ ⟦ decl^* ⟧ (t)(p)(u)(v_1)(stk_1)

(D2) $\underline{\mathbf{D}}$ ⟦ package-decl package-decl* ⟧ (t)(p)(u)(v)(stk)
= $\underline{\mathbf{D}}$ ⟦ package-decl ⟧ (t)(p)(u_1)(v)(stk)
where $u_1 = \lambda v_1, stk_1.\underline{\mathbf{D}}$ ⟦ package-decl* ⟧ (t)(p)(u)(v_1)(stk_1)

(D3) $\underline{\mathbf{D}}$ ⟦ package-body package-body* ⟧ (t)(p)(u)(v)(stk)
= $\underline{\mathbf{D}}$ ⟦ package-body ⟧ (t)(p)(u_1)(v)(stk)
where $u_1 = \lambda v_1, stk_1.\underline{\mathbf{D}}$ ⟦ package-body* ⟧ (t)(p)(u)(v_1)(stk_1)

(D4) $\underline{\mathbf{D}}$ ⟦ use-clause use-clause* ⟧ (t)(p)(u)(v)(stk)
= $\underline{\mathbf{D}}$ ⟦ use-clause ⟧ (t)(p)(u_1)(v)(stk)
where $u_1 = \lambda v_1, stk_1.\underline{\mathbf{D}}$ ⟦ use-clause* ⟧ (t)(p)(u)(v_1)(stk_1)

The Phase 2 processing of declarations proceeds sequentially, from first to last.

(D5) $\underline{\mathbf{D}}$ ⟦ **DEC** object-class id^+ type-mark opt-expr ⟧ (t)(p)(u)(v)(stk)
= **let** d = lookup-desc(type-mark)(t)(p) **in**
(case tag(d)
(***BOOL*** ,***BIT*** ,***INT*** ,***REAL*** ,***TIME*** ,***ENUMTYPE*** ,***SUBTYPE*** ,***INT_TYPE***)
→ gen-scalar-decl
(decl)(object-class)(id^+)(d)(opt-expr)(t)(p)(u)(v)(stk),
ARRAYTYPE
→ gen-array-decl
(decl)(object-class)(id^+)(d)(direction(d))(real-lb(d))
(real-ub(d))(elty(d))(opt-expr)(t)(p)(u)(v)(stk),
RECORDTYPE
→ gen-record-decl
(decl)(object-class)(id^+)(d)(opt-expr)(t)(p)(u)(v)(stk),
OTHERWISE → u(v)(stk))

(D6) $\underline{\mathbf{D}}$ ⟦ **SLCDEC** object-class id^+ slice-name opt-expr ⟧ (t)(p)(u)(v)(stk)
= **let** d = lookup(t)(p)(hd(id^+)) **in**
let anon-array-type-desc = second(type(d)) **in**
gen-array-decl
(decl)(object-class)(id^+)(anon-array-type-desc)
(direction(anon-array-type-desc))(lb(anon-array-type-desc))
(ub(anon-array-type-desc))(elty(anon-array-type-desc))(opt-expr)(t)(p)
(u)(v)(stk)

lookup-desc(id^*)(t)(p)
= (null(id^*)→ void-type-desc(t),
let q = access(rest(id^*))(t)(p) **in**
lookup-desc-on-path(t)(q)(last(id^*)))

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

access(id^*)(t)(p)
= (null(id^*)→ p,
let d = lookup(t)(p)(hd(id^*)) **in**
access(tl(id^*))(t)(%(path(d))(idf(d))))

```
gen-scalar-decl(decl)(object-class)(id+)(d)(expr)(t)(p)(u)(v)(stk)
= (null(expr)
   → gen-scalar-decl-id+(decl)(object-class)(id+)(d)(expr)(t)(p)(u)(v)(stk),
   gen-scalar-decl-id*(decl)(object-class)(id+)(d)(expr)(t)(p)(u)(v)(stk))

gen-scalar-decl-id+(decl)(object-class)(id+)(d)(expr)(t)(p)(u)(v)(stk)
= (object-class = SIG
   → gen-scalar-signal-decl-id+(decl)(id+)(d)(expr)(t)(p)(u)(v)(stk),
   gen-scalar-nonsignal-decl-id+(decl)(id+)(d)(expr)(t)(p)(u)(v)(stk))

gen-scalar-decl-id*(decl)(object-class)(id*)(d)(expr)(t)(p)(u)(v)(stk)
= (null(id*)→ u(v)(stk),
   let id+ = (hd(id*)) in
    gen-scalar-decl-id+(decl)(object-class)(id+)(d)(expr)(t)(p)(u1)(v)(stk)
     where
     u1 = λv1,stk1.
          gen-scalar-decl-id*
             (decl)(object-class)(tl(id*))(d)(expr)(t)(p)(u)(v1)(stk1))


gen-scalar-nonsignal-decl-id+(decl)(id+)(d)(expr)(t)(p)(u)(v)(stk)
= R [[ expr ]] (t)(p)(k)(v)(stk)
    where
    k = λ(e,f),v1,stk1.
         let z = hd(p)
             and suqn+ = get-qids(id+)(t)(p) in
          let v2 = push-universe(v1)(z)(suqn+) in
           let duqn+ = get-qualified-ids(suqn+)(v2) in
            (mk-decl-sd
               (z)(f)(ε)((z))
               (nconc
                  (mk-qual-id-coverings(suqn+)(duqn+)(z)(v)(t),
                  mk-scalar-nonsignal-dec-post
                     (decl)((duqn+,e,d))(t)(p)(u)(v2)(stk))))

get-qids(id*)(t)(p)
= (null(id*)→ ε, cons(qid(t(p)(hd(id*))),get-qids(tl(id*))(t)(p)))

get-qualified-ids(suqn*)(v)
= (null(suqn*)→ ε,
   cons(qualified-id(hd(suqn*))(v),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*)→ ε,
   let suqn = hd(suqn*) in
    cons((DECLARE ,suqn,(TYPE ,PLACEARRAY )),qualified-id-decls(tl(suqn*))))

mk-qual-id-coverings(suqn+)(duqn+)(z)(v)(t)
= (new-declarations()
   → (already-qualified-id(hd(suqn+))(v)
      → (mk-rel(univint-type-desc(t))((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+))),
   (mk-disjoint(z,cons(dot(z),duqn+)),mk-cover(pound(z),cons(dot(z),duqn+))))

mk-scalar-nonsignal-dec-post(decl)(duqn*,e,d)(t)(p)(u)(v)(stk)
= 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)),
    let precondition = mk-constraint-guards((e))((d))(t)(p)(v)(stk) in
     nconc
       (mk-scalar-nonsignal-dec-post-declare(duqn*)(type-spec),
       u1(v)(stk))
      where
      u1 = λv1,stk1.
            (mk-decl-sd
               (hd(p))(precondition)(ε)(duqn*)
               (nconc
                  (mk-scalar-nonsignal-dec-post-init(duqn*)(e)(d),
                  u(v1)(stk1)))))

mk-type-spec(d)(t)(p)
= (case tag(d)
    *BOOL* → (TYPE BOOLEAN) ,
    *BIT* → (TYPE BIT) ,
    (*INT* ,*INT_TYPE* ,*TIME* ) → (TYPE INTEGER) ,
    *REAL* → (TYPE FLOAT) ,
    *VHDLTIME* → (TYPE VHDLTIME) ,
    *ENUMTYPE*
    → (idf(d)= CHARACTER → (TYPE CHARACTER) ,
       cons(TYPE ,cons(ENUMERATION ,literals(d)))),
    *SUBTYPE* → mk-type-spec(base-type(d))(t)(p),
    *VOID* → (TYPE VOID) ,
    *POLY* → (TYPE POLYMORPHIC) ,
    *RECORDTYPE* → cons(TYPE ,cons(RECORD ,record-to-type(components(d))(t)(p))),
    *ARRAYTYPE*
    → let expr1 = lb(d) in
       R ⟦ expr1 ⟧ (t)(p)(k1)(ε)(ε)
       where
       k1 = λ(e1,f1),v1,stk1.
             let expr2 = ub(d) in
              R ⟦ expr2 ⟧ (t)(p)(k2)(v1)(stk1)
              where
              k2 = λ(e2,f2),v2,stk2.
                    cons(TYPE ,
                    (ARRAY ,e1,e2,mk-type-spec(elty(d))(t)(p))),
    *WAVE* → (TYPE ,WAVEFORM ,mk-type-spec(hd(type(d)))(t)(p)),
    OTHERWISE → impl-error("Unrecognized Stage 4 VHDL type: ~a",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* ,*TIME* ,*VHDLTIME* ,*ENUMTYPE* ,*WAVE* ,*VOID* ,
     *POLY* )
    → (mk-rel(d)((EQ ,pound(target),value))),
    *SUBTYPE* → assign(base-type(d))((target,value)),
    *INT_TYPE* → assign(parent-type(d))((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)))),
       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)))),
        (dotted-expr-p(value)→ (mk-rel(d)((EQ ,pound(target),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))))),
     *RECORDTYPE*
     → (dotted-expr-p(value)→ assign-record(d)((target,value)),
        assign-record-fields(components(d))((target,value))),
     OTHERWISE → impl-error("Unrecognized Stage 4 VHDL type tag: ~a",tag(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 )

dotted-expr-p(expr) = consp(expr)∧ hd(expr)= DOT

assign-array-to(target)(aggregate)(element-type-desc)(start-index)(m)
= (null(aggregate)→ ε,
   nconc
      (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)→ ε,
   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 4 VHDL binary operator: ~a",binary-op))

assign-record(d)(target-record,dotted-source-record)
= cons(mk-rel(d)((EQ ,pound(target-record),dotted-source-record)),
   assign-record-aux
      (components(d))((target-record,second(dotted-source-record))))

assign-record-aux(comp*)(target-record,source-record-name)
= (null(comp*)→ ε,
    let (id,d) = hd(comp*) in
     nconc
        (assign
            (d)
            ((mk-recelt(target-record,id),dot(mk-recelt(source-record-name,id)))),
        assign-record-aux(tl(comp*))((target-record,source-record-name))))

assign-record-fields(comp*)(target-record,source-fields)
= (null(comp*)→ ε,
    let (id,d) = hd(comp*) in
     nconc
        (assign(d)((mk-recelt(target-record,id),second(assoc(id,source-fields)))),
        assign-record-fields(tl(comp*))((target-record,source-fields))))

mk-recelt(e)(id) = (RECORD ,e,id)


gen-scalar-signal-decl-id+(decl)(id+)(d)(expr)(t)(p)(u)(v)(stk)
= 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
             let v2 = push-universe(v1)(z)(suqn+) in
              let signal-duqn+ = get-qualified-ids(signal-suqn+)(v2)
                  and driver-duqn+ = get-qualified-ids(driver-suqn+)(v2) in
               let duqn+ = append(signal-duqn+,driver-duqn+) in
                (mk-decl-sd
                    (z)(f)(ε)((z))
                    (nconc
                        (mk-qual-id-coverings(suqn+)(duqn+)(z)(v)(t),
                        mk-scalar-signal-dec-post
                           (decl)((duqn+,signal-duqn+,driver-duqn+,e,d))(t)(p)(u)
                           (v2)(stk))))
```

```
name-drivers(signal-names)
= (null(signal-names)→ ε,
   cons(name-driver(hd(signal-names)),name-drivers(tl(signal-names))))


mk-scalar-signal-dec-post(decl)(duqn*,signal-duqn*,driver-duqn*,e,d)(t)(p)(u)(v)(stk)
= let sigtype-spec = mk-sigtype-spec(d)(t)(p)
      and waveform-type-spec = (TYPE ,WAVEFORM ,mk-type-spec(d)(t)(p))
      and precondition = mk-constraint-guards((e))((d))(t)(p)(v)(stk) in
    nconc
      (mk-scalar-signal-dec-post-declare
         (signal-duqn*)(driver-duqn*)(sigtype-spec)(waveform-type-spec),
      u1(v)(stk))
     where
     u1 = λv1,stk1.
           (mk-decl-sd
              (hd(p))(precondition)(ε)(duqn*)
              (nconc
                 (mk-scalar-signal-dec-post-init
                    (signal-duqn*)(driver-duqn*)(e)(d)(waveform-type-desc(d)),
                 u(v1)(stk1))))


mk-scalar-signal-dec-post-declare(signal-duqn*)(driver-duqn*)(sigtype-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))((sigtype-spec,waveform-type-spec)),
      mk-scalar-signal-dec-post-declare
        (tl(signal-duqn*))(tl(driver-duqn*))(sigtype-spec)(waveform-type-spec)))


mk-scalar-signal-decl(signal-name,driver-name)(sigtype-spec,waveform-type-spec)
= (mk-scalar-decl(signal-name,sigtype-spec),
   mk-scalar-decl(driver-name,waveform-type-spec))


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) = <WAVEFORM ,ε,*WAVE* ,(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)))
```

```
gen-array-decl(decl)
              (object-class)(id+)(type-desc)(direction)(lower-bound)(upper-bound)(element-type-desc)(expr)
              (t)(p)(u)(v)(stk)
= (null(expr)
   → gen-array-decl-id+
        (decl)(object-class)(id+)(type-desc)(direction)(lower-bound)(upper-bound)
        (element-type-desc)(expr)(t)(p)(u)(v)(stk),
   gen-array-decl-id*
      (decl)(object-class)(id+)(type-desc)(direction)(lower-bound)(upper-bound)
      (element-type-desc)(expr)(t)(p)(u)(v)(stk))

real-lb(d)
= let bound = lb(d) in
    (is-num-lit?(bound)→ bound,
     (REF ,((SREF ,path(d),mk-tick-low(idf(d))))))

real-ub(d)
= (path(d)= (STANDARD) ∧ idf(d)∈ (STRING BIT_VECTOR) → ε,
   let bound = ub(d) in
     (is-num-lit?(bound)→ bound,
      (REF ,((SREF ,path(d),mk-tick-high(idf(d)))))))

mk-tick-low(id) = catenate(id,"'LOW")

mk-tick-high(id) = catenate(id,"'HIGH")

gen-array-decl-id+(decl)
                  (object-class)(id+)(type-desc)(direction)(lower-bound)(upper-bound)(element-type-desc)(expr)
                  (t)(p)(u)(v)(stk)
= (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+
      (decl)(id+)(direction)(lower-bound)(upper-bound)(element-type-desc)(expr)
      (t)(p)(u)(v)(stk))

gen-array-decl-id*(decl)
                  (object-class)(id*)(type-desc)(direction)(lower-bound)(upper-bound)(element-type-desc)(expr)
                  (t)(p)(u)(v)(stk)
= (null(id*)→ u(v,stk),
   let id+ = (hd(id*)) in
     gen-array-decl-id+
        (decl)(object-class)(id+)(type-desc)(direction)(lower-bound)(upper-bound)
        (element-type-desc)(expr)(t)(p)(u1)(v)(stk)
      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)(v1)(stk1))


gen-array-nonsignal-decl-id+(decl)
                           (id+)(direction)(expr1)(expr2)(element-type-desc)(expr)
                           (t)(p)(u)(v)(stk)
= R [[ expr ]] (t)(p)(k)(v)(stk)
    where
```

```
k = λ(e,f),v1,stk1.
      R [[ expr1 ]] (t)(p)(k1)(v1)(stk1)
       where
       k1 = λ(e1,f1),v2,stk2.
              R [[ expr2 ]] (t)(p)(k2)(v2)(stk2)
               where
               k2 = λ(e2,f2),v3,stk3.
                      let z = hd(p)
                          and len = length-expr(expr)
                          and suqn+ = get-qids(id+)(t)(p) in
                       let v4 = push-universe(v3)(z)(suqn+) in
                        let duqn+ = get-qualified-ids(suqn+)(v4) in
                         let g1 = (e1 ∧ e2
                                    → mk-rel
                                         (univint-type-desc(t))
                                         ((LE ,e1,e2)),
                                    TRUE )
                             and g2 = (e1 ∧ e2
                                         → mk-rel
                                              (univint-type-desc(t))
                                              ((GE ,
                                                mk-exp2
                                                   (ADD ,mk-exp2(SUB ,e2,e1),
                                                   1),len)),
                                         TRUE ) in
                          (mk-decl-sd
                             (z)
                             (nconc
                                (f1,f2,(g1),
                                (len = 0 → f, nconc((g2),f))))(ε)((z))
                             (nconc
                                (mk-qual-id-coverings
                                    (suqn+)(duqn+)(z)(v)(t),
                                mk-array-nonsignal-dec-post
                                   (decl)
                                   ((duqn+,e,direction,e1,e2,element-type-desc))
                                   (t)(p)(u)(v4)(stk3))))

length-expr(expr)
= (null(expr)→ 0,
   hd(expr)∈ (BITSTR STR PAGGR) → length(second(expr)),
   1)

mk-array-nonsignal-dec-post(decl)
                           (duqn*,e,direction,lower-bound,upper-bound,element-type-desc)
                           (t)(p)(u)(v)(stk)
= let element-type-spec = mk-type-spec(element-type-desc)(t)(p) in
   (null(e)
    → nconc
         (mk-array-nonsignal-dec-post-declare
            (duqn*)(direction)(lower-bound)(upper-bound)(element-type-spec),
         u(v)(stk)),
    nconc
      (mk-array-nonsignal-dec-post-declare
          (duqn*)(direction)(lower-bound)(upper-bound)(element-type-spec),
      u1(v)(stk))
     where
```

```
    u1 = λv1,stk1.
          (mk-decl-sd
             (hd(p))(ε)(ε)(duqn*)
             (nconc
                ((direction = TO
                  → mk-array-nonsignal-dec-post-init-to
                        (duqn*)(e)(element-type-desc)(lower-bound),
                    mk-array-nonsignal-dec-post-init-downto
                       (duqn*)(e)(element-type-desc)(upper-bound)),
                 u(v1)(stk1)))))

mk-array-nonsignal-dec-post-declare(duqn*)(direction)(lower-bound)(upper-bound)(element-type-spec)
= (null(duqn*)→ ε,
   let duqn = hd(duqn*) in
    nconc
      (mk-vhdl-array-decl
         (duqn)(direction)(lower-bound)
         ((null(upper-bound)
           → (lower-bound = 1 → (RANGE ,duqn),
              mk-exp2(SUB ,mk-exp2(ADD ,(RANGE ,duqn),lower-bound),1)),
           upper-bound))(element-type-spec),
       mk-array-nonsignal-dec-post-declare
         (tl(duqn*))(direction)(lower-bound)(upper-bound)(element-type-spec)))

mk-vhdl-array-decl(id)(direction)(lower-bound)(upper-bound)(element-type-spec)
= (case second(element-type-spec)
     BIT
     → (mk-array-decl(id)(lower-bound)(upper-bound)(element-type-spec),
        mk-bitvec-fn-decl(id)(direction)(lower-bound)(upper-bound)),
     CHARACTER
     → (mk-array-decl(id)(lower-bound)(upper-bound)(element-type-spec),
        mk-string-fn-decl(id)(direction)(lower-bound)(upper-bound)),
     OTHERWISE
     → (mk-array-decl(id)(lower-bound)(upper-bound)(element-type-spec)))

mk-array-decl(id)(lower-bound)(upper-bound)(element-type-spec)
= (DECLARE ,id,(TYPE ,ARRAY ,lower-bound,upper-bound,element-type-spec))

mk-bitvec-fn-decl(bitvec-name)(direction)(lower-bound)(upper-bound)
= let bitvec-elt-names = (direction = TO
                          → mk-slice-elt-names-to
                               (bitvec-name)(lower-bound)(upper-bound),
                          mk-slice-elt-names-downto
                             (bitvec-name)(lower-bound)(upper-bound)) in
   (DECLARE ,bitvec-name,(TYPE ,FN ,concatenate-bits(bitvec-elt-names)))

mk-string-fn-decl(string-name)(direction)(lower-bound)(upper-bound)
= let string-elt-names = (direction = TO
                          → mk-slice-elt-names-to
                               (string-name)(lower-bound)(upper-bound),
                          mk-slice-elt-names-downto
                             (string-name)(lower-bound)(upper-bound)) in
   (DECLARE ,string-name,(TYPE ,FN ,concatenate-characters(string-elt-names)))

mk-slice-elt-names-to(slice-name)(lower-bound)(upper-bound)
= (lower-bound > upper-bound → ε,
   cons(mk-array-elt(slice-name)(lower-bound),
   mk-slice-elt-names-to(slice-name)(lower-bound+1)(upper-bound)))
```

```
mk-slice-elt-names-downto(slice-name)(lower-bound)(upper-bound)
= (upper-bound < lower-bound → ε,
    cons(mk-array-elt(slice-name)(upper-bound),
    mk-slice-elt-names-downto(slice-name)(lower-bound)(upper-bound−1)))

mk-array-elt(id)(e) = (ELEMENT ,id,e)

concatenate-bits(bit-names) = cons(USCONC ,mk-dotted-names(bit-names))

concatenate-characters(char-names) = cons(ACONC ,mk-dotted-names(char-names))

mk-dotted-names(names)
= (null(names)→ ε, cons(dot(hd(names)),mk-dotted-names(tl(names))))

mk-array-nonsignal-dec-post-init-to(duqn*)(e)(element-type-desc)(lower-bound)
= (null(duqn*)→ ε,
    nconc
      (assign-array-to(hd(duqn*))(e)(element-type-desc)(lower-bound)(0),
      mk-array-nonsignal-dec-post-init-to
        (tl(duqn*))(e)(element-type-desc)(lower-bound)))

mk-array-nonsignal-dec-post-init-downto(duqn*)(e)(element-type-desc)(upper-bound)
= (null(duqn*)→ ε,
    nconc
      (assign-array-downto(hd(duqn*))(e)(element-type-desc)(upper-bound)(0),
      mk-array-nonsignal-dec-post-init-downto
        (tl(duqn*))(e)(element-type-desc)(upper-bound)))


gen-array-signal-decl-id+(decl)
                         (id+)(type-desc)(direction)(expr1)(expr2)(element-type-desc)(expr)
                         (t)(p)(u)(v)(stk)
= R [[ expr ]] (t)(p)(k)(v)(stk)
    where
    k = λ(e,f),v1,stk1.
          R [[ expr1 ]] (t)(p)(k1)(v1)(stk1)
            where
            k1 = λ(e1,f1),v2,stk2.
                   R [[ expr2 ]] (t)(p)(k2)(v2)(stk2)
                     where
                     k2 = λ(e2,f2),v3,stk3.
                            let z = hd(p)
                                and len = length-expr(expr)
                                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
                                 let v4 = push-universe(v3)(z)(suqn+) in
                                  let signal-duqn+ = get-qualified-ids
                                                        (signal-suqn+)(v4)
                                      and driver-duqn+ = get-qualified-ids
                                                            (driver-suqn+)(v4) in
                                    let duqn+ = append
                                                  (signal-duqn+,driver-duqn+) in
                                     let g1 = (e1 ∧ e2
                                                → mk-rel
                                                    (univint-type-desc(t))
                                                    ((LE ,e1,e2)),
```

```
                    TRUE )
              and g2 = (e1 ∧ e2
                        → mk-rel
                            (univint-type-desc(t))
                            ((GE ,
                              mk-exp2
                                (ADD ,
                                 mk-exp2(SUB ,e2,e1),1),len)),
                           TRUE ) in
        (mk-decl-sd
            (z)
            (nconc
                (f1,f2,(g1),
                (len = 0 → f, nconc(f,(g2)))))(ε)((z))
            (nconc
                (mk-qual-id-coverings
                    (suqn+)(duqn+)(z)(v)(t),
                mk-array-signal-dec-post
                  (decl)
                  ((duqn+,signal-duqn+,driver-duqn+,e,type-desc,direction,
                    e1,e2,element-type-desc))(t)(p)(u)
                  (v4)(stk3))))

mk-array-signal-dec-post(decl)
                        (duqn*,signal-duqn*,driver-duqn*,e,type-desc,
                         direction,lower-bound,upper-bound,element-type-desc)
                        (t)(p)(u)(v)(stk)
= let element-sigtype-spec = mk-sigtype-spec(element-type-desc)(t)(p)
      and element-waveform-type-spec = mk-waveform-type-spec
                                          (mk-type-spec(element-type-desc)(t)(p)) in
    nconc
      (mk-array-signal-dec-post-declare
          (signal-duqn*)(driver-duqn*)(direction)(lower-bound)(upper-bound)
          (element-sigtype-spec)(element-waveform-type-spec)(tt)(t)(p)(v)(stk),
       u1(v)(stk))
      where
      u1 = λv1,stk1.
            (mk-decl-sd
                (hd(p))(ε)(ε)(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*)(direction)(lower-bound)(upper-bound)
                                (element-sigtype-spec)(element-waveform-type-spec)(fn-decls?)
                                (t)(p)(v)(stk)
= (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)(direction)(lower-bound)(upper-bound)
            (element-sigtype-spec)(element-waveform-type-spec)(fn-decls?)(t)(p)(v)
            (stk),
         mk-array-signal-dec-post-declare
           (tl(signal-duqn*))(tl(driver-duqn*))(direction)(lower-bound)
           (upper-bound)(element-sigtype-spec)(element-waveform-type-spec)
           (fn-decls?)(t)(p)(v)(stk)))

mk-array-signal-decl(signal-name)(driver-name)(direction)(lower-bound)(upper-bound)
                    (element-sigtype-spec)(element-waveform-type-spec)(fn-decls?)
                    (t)(p)(v)(stk)
= nconc
      (mk-vhdl-sigarray-decl
          (signal-name)(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))(element-sigtype-spec)(fn-decls?),
      (mk-array-decl
          (driver-name)(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))(element-waveform-type-spec)))

mk-array-signal-elt-fn-decls(signal-duqn)(driver-duqn)(element-type-desc)(lower-bound)(upper-bound)
                            (t)(p)(v)(stk)
= (is-array-tdesc?(element-type-desc)
   → 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
       let expr1 = real-lb(element-type-desc) in
        R [[ expr1 ]] (t)(p)(k1)(v)(stk)
        where
        k1 = λ(e1,f1),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-elt-fn-decls-aux
                       (signal-elts)(driver-elts)(elty(element-type-desc))
                       (e1)(e2)(t)(p)(v2)(stk2),
   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-fn-decls(scalar-signal-elts,scalar-driver-elts))

mk-array-signal-elt-fn-decls-aux(signal-duqn*)(driver-duqn*)
                                (element-type-desc)(lower-bound)(upper-bound)
                                (t)(p)(v)(stk)
= (null(signal-duqn*)→ ε,
   let signal-duqn = hd(signal-duqn*)
       and driver-duqn = hd(driver-duqn*) in
```

```
    nconc
      (mk-array-signal-elt-fn-decls
          (signal-duqn)(driver-duqn)(element-type-desc)(lower-bound)(upper-bound)
          (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)→ ε,
   cons(mk-scalar-signal-fn-decl(hd(signal-names),hd(driver-names)),
   mk-scalar-signal-fn-decls(tl(signal-names),tl(driver-names))))

mk-array-signal-dec-post-init(signal-duqn*)(driver-duqn*)(e)
                             (type-desc)(direction)(lower-bound)(upper-bound)
                             (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(e1,f1),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))

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)
= (is-array-tdesc?(element-type-desc)
   → let expr1 = real-lb(element-type-desc) in
     R ⟦ expr1 ⟧ (t)(p)(k1)(v)(stk)
     where
     k1 = λ(e1,f1),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))

mk-array-signal-dec-post-init-elt-arrays-to(signal-duqn*)(driver-duqn*)(e)
                                           (type-desc)(lower-bound)(upper-bound)
                                           (elt-type-desc)(elt-direction)(elt-lower-bound)(elt-upper-bound)
                                           (t)(p)(v)(stk)
= (null(signal-duqn*)→ ε,
   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)(elty(elt-type-desc))
            (waveform-type-desc(elty(elt-type-desc)))(t)(p)(v)(stk),
        mk-array-signal-dec-post-init-elt-arrays-to
          (tl(signal-duqn*))(tl(driver-duqn*))(tl(e))(type-desc)(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-elt-arrays-downto(signal-duqn*)(driver-duqn*)(e)
                                               (type-desc)(lower-bound)(upper-bound)
                                               (elt-type-desc)(elt-direction)(elt-lower-bound)(elt-upper-bound)
                                               (t)(p)(v)(stk)
= (null(signal-duqn*)→ ε,
   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)(elty(elt-type-desc))
            (waveform-type-desc(elty(elt-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)(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)(lower-bound)(upper-bound)
                                 (element-type-desc)(element-waveform-type-desc)
                                 (t)(p)(v)(stk)
= (null(signal-duqn*)→ ε,
```

```
    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)(lower-bound)(upper-bound)
                                            (element-type-desc)(element-waveform-type-desc)
                                            (t)(p)(v)(stk)
= (null(signal-duqn*)→ ε,
    let signal-duqn = hd(signal-duqn*)
        and driver-duqn = hd(driver-duqn*) in
      let initial-waveforms = init-array-signal-to
                                (signal-duqn)(driver-duqn)(e)(type-desc)
                                (element-type-desc)(lower-bound)(upper-bound) in
        nconc
          (assign-array-to
              (driver-duqn)(initial-waveforms)(element-waveform-type-desc)
              (lower-bound)(0),
          mk-array-signal-dec-post-init-elt-scalars-to
            (tl(signal-duqn*))(tl(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-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)
= (null(signal-duqn*)→ ε,
    let signal-duqn = hd(signal-duqn*)
        and driver-duqn = hd(driver-duqn*) in
      let initial-waveforms = init-array-signal-downto
                                (signal-duqn)(driver-duqn)(e)(type-desc)
                                (element-type-desc)(lower-bound)(upper-bound) in
        nconc
          (assign-array-downto
              (driver-duqn)(initial-waveforms)(element-waveform-type-desc)
              (upper-bound)(0),
          mk-array-signal-dec-post-init-elt-scalars-downto
            (tl(signal-duqn*))(tl(driver-duqn*))(e)(type-desc)(lower-bound)
            (upper-bound)(element-type-desc)(element-waveform-type-desc)(t)(p)(v)
            (stk)))


(D7) D [[ ETDEC id id+ ]] (t)(p)(u)(v)(stk)
     = (mk-decl-sd
          (hd(p))(ε)(ε)(ε)
          (nconc(mk-etdec-post((id))(t)(p),u(v)(stk))))

mk-etdec-post(type-mark)(t)(p)
= let d = lookup-desc(type-mark)(t)(p) in
    mk-enumlit-rels(d)(literals(d))
```

```
mk-enumlit-rels(d)(id*)
= (null(tl(id*))→ ε,
   let id1 = hd(id*)
       and id2 = hd(tl(id*)) in
    cons(mk-rel(d)((PRED ,id1 ,id2)),mk-enumlit-rels(d)(tl(id*))))
```

The translation of an enumeration type declaration emits an SDVS declaration of the enumeration type.

```
(D8) D [[ ATDEC id discrete-range type-mark ]] (t)(p)(u)(v)(stk)
     = let (direction,expr1 ,expr2) = discrete-range in
         let lower-bound = (direction = TO → expr1, expr2)
             and upper-bound = (direction = TO → expr2, expr1) in
          attributes-low-high
            ((id,lower-bound,upper-bound,(UNIVERSAL_INTEGER) ))(t)(p)(u)(v)(stk)

attributes-low-high(id,lower-bound,upper-bound,attribute-type-mark)(t)(p)(u)(v)(stk)
= let decl1 = (DEC ,SYSGEN ,(mk-tick-low(id)),attribute-type-mark,lower-bound)
      and decl2 = (DEC ,SYSGEN ,(mk-tick-high(id)),attribute-type-mark,upper-bound) in
   let decl+ = (decl1 ,decl2) in
    D [[ decl+ ]] (t)(p)(u)(v)(stk)

mk-tick-low(id) = catenate(id,"'LOW")

mk-tick-high(id) = catenate(id,"'HIGH")
```

An array type declaration declares and initializes the `'low` and `'high` array type attributes.

```
(D9) D [[ PACKAGE id decl* opt-id ]] (t)(p)(u)(v)(stk)
     = D [[ decl* ]] (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) D [[ PACKAGEBODY id decl* opt-id ]] (t)(p)(u)(v)(stk)
      = let pb-exit-desc = <*PACKAGE-BODY-EXIT* ,id,p,λv,s.u(v)(s)> in
          D [[ decl* ]] (t)(%(p)(id))(u1)(v)(stk-push(pb-exit-desc)(stk))
           where u1 = λv1 ,stk1 .package-body-exit(v1)(stk1 )

package-body-exit(v)(stk)
= let <tg,qname,p,g> = hd(stk) in
   (case tg
      *STKBOTTOM* → model-execution-complete(qname),
      *UNDECLARE* → g(λvv,s.package-body-exit(vv)(s))(v)(stk),
      (*BEGIN* ) → package-body-exit(v)(stk-pop(stk)),
      (*PACKAGE-BODY-EXIT* ,*LOOP-EXIT* ,*SUBPROGRAM-RETURN* ) → g(v)(stk-pop(stk)),
      OTHERWISE
      → 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) $\underline{\mathbf{D}}$ ⟦ **PROCEDURE** id proc-par-spec* ⟧ (t)(p)(u)(v)(stk) = u(v)(stk)

(D12) $\underline{\mathbf{D}}$ ⟦ **FUNCTION** id func-par-spec* type-mark ⟧ (t)(p)(u)(v)(stk) = u(v)(stk)

(D13) $\underline{\mathbf{D}}$ ⟦ **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) $\underline{\mathbf{D}}$ ⟦ **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.

```
(D15) D ⟦ STDEC id type-mark opt-discrete-range ⟧ (t)(p)(u)(v)(stk)
      = let z = hd(p)
            and subtype-desc = lookup-desc-on-path(t)(p)(id) in
        let basetype-desc = base-type(subtype-desc) in
         let expr1 = type-tick-low(basetype-desc)
             and expr2 = type-tick-low(subtype-desc)
             and expr3 = type-tick-high(subtype-desc)
             and expr4 = type-tick-high(basetype-desc) in
          R ⟦ expr1 ⟧ (t)(p)(k1)(v)(stk)
           where
           k1 = λ(e1,f1),v1,stk1.
                  R ⟦ expr2 ⟧ (t)(p)(k2)(v1)(stk1)
                   where
                   k2 = λ(e2,f2),v2,stk2.
                          R ⟦ expr3 ⟧ (t)(p)(k3)(v2)(stk2)
                           where
                           k3 = λ(e3,f3),v3,stk3.
                                  R ⟦ expr4 ⟧ (t)(p)(k4)(v3)(stk3)
                                   where
                                   k4 = λ(e4,f4),v4,stk4.
                                          (mk-decl-sd
                                             (z)
                                             (nconc
                                                ((e1
                                                  → (mk-rel
                                                        (basetype-desc)
                                                        ((LE ,e1,e2))),
                                                  ε),
                                                 (e4
                                                  → (mk-rel
                                                        (basetype-desc)
                                                        ((LE ,e3,e4))),
                                                  ε)))(ε)(ε)
                                               (u1(v4)(stk4)))
                                   where
                                   u1 = λv5,stk5.
                                          attributes-low-high
                                             ((id,expr2,expr3,
                                               (idf(basetype-desc))))(t)(p)(u)
                                             (v5)(stk5)
```

The Phase 2 semantics of subtype declarations generates a state delta with guards in the precondition to ensure that the subtype range falls within the range of allowable values for the subtype's base type. Assuming this holds, the continuation in the state delta's postcondition performs the Phase 2 processing of declarations and initializations for the `'low` and `'high` attributes representing the subtype bounds.

(D16) $\underline{\mathbf{D}}$ ⟦ **ITDEC** id discrete-range ⟧ (t)(p)(u)(v)(stk)
= **let** z = hd(p)
and integer-type-desc = lookup-desc-on-path(t)(p)(id) **in**
let $expr_1$ = type-tick-low(integer-type-desc)
and $expr_2$ = type-tick-high(integer-type-desc) **in**
attributes-low-high
((id,$expr_1$,$expr_2$,(**UNIVERSAL_INTEGER**)))(t)(p)(u)(v)(stk)

The Phase 2 semantics of integer type declarations simply processes declarations and initializations for the `'low` and `'high` attributes representing the integer type bounds.

(D17) $\underline{\mathbf{D}}$ ⟦ **COMPONENT** id $decl_1^*$ $decl_2^*$ phase1-hook ⟧ (t)(p)(u)(v)(stk) = u(v)(stk)

8.4.9 Concurrent Statements

(CS0) $\underline{\mathbf{CS}}$ ⟦ ε ⟧ (t)(p)(u)(v)(stk) = u(v)(stk)

(CS1) $\underline{\mathbf{CS}}$ ⟦ conc-stat conc-stat* ⟧ (t)(p)(u)(v)(stk)
= $\underline{\mathbf{CS}}$ ⟦ conc-stat ⟧ (t)(p)(u_1)(v)(stk)
where u_1 = λv,stk.$\underline{\mathbf{CS}}$ ⟦ conc-stat* ⟧ (t)(p)(u)(v)(stk)

A list of concurrent statements is translated in order, from first to last.

(CS2) $\underline{\mathbf{CS}}$ ⟦ **BLOCK** id decl* conc-stat* opt-id phase1-hook ⟧ (t)(p)(u)(v)(stk)
= **let** p_1 = %(p)(id) **in**
$\underline{\mathbf{D}}$ ⟦ decl* ⟧ (t)(p_1)(u_1)(v)(stk)
where u_1 = λv_1,stk_1.$\underline{\mathbf{CS}}$ ⟦ conc-stat* ⟧ (t)(p_1)(u)(v_1)(stk_1)

(CS3) $\underline{\mathbf{CS}}$ ⟦ **PROCESS** id decl* seq-stat* opt-id phase1-hook ⟧ (t)(p)(u)(v)(stk)
= **let** process-qid = qid(lookup(t)(p)(id))
and p_1 = %(p)(id) **in**
(mk-decl-sd
(hd(p))(ε)(ε)(ε)
((make-vhdl-process-elaborate
(process-qid)(t)(p_1)(seq-stat*)(u_1)(v)(stk))))
where u_1 = λv,stk.$\underline{\mathbf{D}}$ ⟦ decl* ⟧ (t)(p_1)(u)(v)(stk)

8.4.10 Sequential Statements

(SS0) $\underline{\mathbf{SS}}$ ⟦ ε ⟧ (t)(p)(c)(v)(stk) = c(v)(stk)

(SS1) $\underline{\mathbf{SS}}$ ⟦ seq-stat seq-stat* ⟧ (t)(p)(c)(v)(stk)
= $\underline{\mathbf{SS}}$ ⟦ seq-stat ⟧ (t)(p)(c_1)(v)(stk)
where c_1 = λv,stk.$\underline{\mathbf{SS}}$ ⟦ seq-stat* ⟧ (t)(p)(c)(v)(stk)

A list of sequential statements is translated in order, from first to last.

(SS2) $\underline{\mathbf{SS}}$ ⟦ **NULL** atmark ⟧ (t)(p)(c)(v)(stk)
= ((**EQ** ,pound(catenate(hd(p),"\pc")),atmark),
mk-sd(hd(p))(ε)(ε)(ε)(c(v)(stk)))

NULL statements have no effect.

(SS3) $\underline{\mathbf{SS}}$ ⟦ **VARASSN** atmark ref expr ⟧ (t)(p)(c)(v)(stk)
= cons((**EQ** ,pound(catenate(hd(p),"\pc")),atmark),
let d = $\underline{\mathbf{T}}$ ⟦ ref ⟧ (t)(p) **in**
$\underline{\mathbf{E}}$ ⟦ ref ⟧ (t)(p)(k_1)(v)(stk)
where
k_1 = λ(e_1,f_1),v,stk.
$\underline{\mathbf{R}}$ ⟦ expr ⟧ (t)(p)(k_2)(v)(stk)
where
k_2 = λ(e_2,f_2),v,stk.
let precondition = nconc
(mk-constraint-guards
((e_2))((d))(t)
(p)(v)(stk),f_1,f_2) **in**
(mk-sd
(hd(p))(precondition)(ε)((e_1))
(nconc
(assign(d)((e_1,e_2)),
c(v)(stk)))))

assign(d)(target,value)
= (case tag(d)
(***BOOL*** ,***BIT*** ,***INT*** ,***REAL*** ,***TIME*** ,***VHDLTIME*** ,***ENUMTYPE*** ,***WAVE*** ,***VOID*** ,
POLY)
→ (mk-rel(d)((**EQ** ,pound(target),value))),
SUBTYPE → assign(base-type(d))((target,value)),
INT_TYPE → assign(parent-type(d))((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)))),
    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)))),
    (dotted-expr-p(value)→ (mk-rel(d)((EQ ,pound(target),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))))),
*RECORDTYPE*
→ (dotted-expr-p(value)→ assign-record(d)((target,value)),
    assign-record-fields(components(d))((target,value))),
OTHERWISE → impl-error("Unrecognized Stage 4 VHDL type tag: ~a",tag(d)))
```

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 state delta 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 state deltas by appropriately using the continuation **c**. Assignments in Stage 4 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) SS ⟦ SIGASSN atmark delay-type ref waveform ⟧ (t)(p)(c)(v)(stk)
    = cons((EQ ,pound(catenate(hd(p),"\pc")),atmark),
      let d = T ⟦ ref ⟧ (t)(p) in
        (case tag(d)
          (*BOOL* ,*BIT* ,*INT* ,*REAL* ,*TIME* ,*ENUMTYPE* ,*SUBTYPE* ,
           *INT_TYPE* )
          → scalar-signal-assignment
              (seq-stat)(delay-type)(ref)(waveform)(d)(t)(p)(c)(v)(stk),
          *ARRAYTYPE*
          → array-signal-assignment
```

```
            (atmark)(delay-type)(ref)(waveform)(t)(p)(c)(v)(stk),
        OTHERWISE
        → impl-error
            ("Signal assignment not implemented for object ",ref,
            " of type ",d)))

scalar-signal-assignment(seq-stat)(delay-type)(ref)(waveform)(d)(t)(p)(c)(v)(stk)
= E [[ ref ]] (t)(p)(k)(v)(stk)
  where
  k = λ(signal-name,guard),v,stk.
      let driver-name = name-driver(signal-name) in
       W [[ waveform ]] (d)(t)(p)(wave-cont)(v)(stk)
        where
        wave-cont = λ(trans*,guard*),v,stk.
                    let all-guards = nconc(guard,guard*) in
                     (delay-type = TRANSPORT
                      → (mk-sd
                           (hd(p))(all-guards)(ε)((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))(ε)((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)),
                           all-guards))(ε)((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 ,ε,*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)


array-signal-assignment(atmark)(delay-type)(ref)(waveform)(t)(p)(c)(v)(stk)
= let seq-stat+ = cascade-array-signal-assignment
                    (atmark)(delay-type)(ref)(waveform)(t)(p)(c)(v)(stk) in
    SS [[ seq-stat+ ]] (t)(p)(c)(v)(stk)

cascade-array-signal-assignment(atmark)(delay-type)(ref)(agg-wave)(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(atmark)(delay-type)(array-refs)(element-waves)

mk-scalar-signal-assignments(atmark)(delay-type)(array-refs)(element-waves)
= (null(array-refs)→ ε,
   cons((SIGASSN ,atmark,delay-type,hd(array-refs),hd(element-waves)),
   mk-scalar-signal-assignments
     (atmark)(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)(v)(stk)
      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 atmark cond-part+ else-part ⟧ (t)(p)(c)(v)(stk)
      = cons((EQ ,pound(catenate(hd(p),"\pc")),atmark),
        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-stat1*) = hd(cond-part*) in
    R ⟦ expr ⟧ (t)(p)(k)(v)(stk)
     where
     k = λ(e,f),v1,stk1.
          (mk-sd
             (hd(p))(cons(e,f))(ε)(ε)
             (let c1 = λv2,stk2.SS ⟦ seq-stat1* ⟧ (t)(p)(c)(v2)(stk2) in
               c1(v1)(stk1)),
           mk-sd
             (hd(p))(cons(mk-not(e),f))(ε)(ε)
             (let c2 = λv3,stk3.
                         gen-if
                            (tl(cond-part*))(seq-stat*)(ε)(t)(p)(c)(v3)(stk3) in
              c2(v1)(stk1))))
```

The abstract syntax of a Stage 4 VHDL `IF` statement consists of a finite, nonempty list of **cond-part**s 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* state deltas: 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-stat**s 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) SS ⟦ CASE atmark expr case-alt+ ⟧ (t)(p)(c)(v)(stk)
      = cons((EQ ,pound(catenate(hd(p),"\pc")),atmark),
        R ⟦ expr ⟧ (t)(p)(k)(v)(stk)
          where
          k = λ(e,f),v,stk.
                let d = T ⟦ expr ⟧ (t)(p) in
                  gen-case(ε)(d)((e,f))(case-alt+)(t)(p)(c)(v)(stk))

gen-case(g)(d)(e,f)(case-alt*)(t)(p)(c)(v)(stk)
= (null(case-alt*)→ ε,
   let (h,sd) = gen-alt(g)(d)((e,f))(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)))

gen-alt(g)(d)(e,f)(case-alt)(t)(p)(c)(v)(stk)
= let case-alt-tag = hd(case-alt) in
   (case-alt-tag = CASEOTHERS
    → let seq-stat* = hd(tl(case-alt)) in
        let c1 = λv1,stk1.SS ⟦ seq-stat* ⟧ (t)(p)(c)(v1)(stk1) in
          (ε,
           mk-sd
              (hd(p))(append(f,(mk-not(mk-ors(g)))))(ε)(ε)
              (c1(v)(stk))),
    let (case-set,seq-stat*) = tl(case-alt) in
     let c1 = λv1,stk1.SS ⟦ seq-stat* ⟧ (t)(p)(c)(v1)(stk1) in
      let h = append(f,gen-guard(case-set)(d)(e)(t)(p)) in
       (h,mk-sd(hd(p))(h)(ε)(ε)(c1(v)(stk))))

mk-ors(disjs)
= (case length(disjs)
     1 → hd(disjs),
     2 → mk-or(hd(disjs))(hd(tl(disjs))),
     OTHERWISE → mk-or(hd(disjs))(mk-ors(tl(disjs))))

mk-or(e1,e2)
= (null(e1)→ e2,
   null(e2)→ e1,
   consp(e1)∧ consp(e2)
   → (hd(e1)= OR
      → (hd(e2)= OR → cons(OR ,append(tl(e1),tl(e2))), append(e1,(e2))),
      hd(e2)= OR → nconc((OR ,e1),tl(e2)),
      (OR ,e1,e2)),
   (OR ,e1,e2))

gen-guard(discrete-range*)(d)(e)(t)(p)
= (null(discrete-range*)→ ε,
   let (direction,expr1,expr2) = hd(discrete-range*) in
    R ⟦ expr1 ⟧ (t)(p)(k1)(ε)(ε)
     where
     k1 = λ(e1,f1),v1,stk1.
           (expr1 = expr2
            → let h = nconc(f1,(mk-rel(d)((EQ ,e,e1)))) in
                (null(tl(discrete-range*))→ h,
                 (cons(OR ,
                  cons(hd(h),gen-guard(tl(discrete-range*))(d)(e)(t)(p))))),
           R ⟦ expr2 ⟧ (t)(p)(k2)(v1)(stk1)
            where
            k2 = λ(e2,f2),v2,stk2.
```

```
let h = nconc
            (f1,f2,
            (direction = TO
             → ((AND ,mk-rel(d)((GE ,e,e1)),
                  mk-rel(d)((LE ,e,e2)))),
              ((AND ,mk-rel(d)((LE ,e,e1)),
                mk-rel(d)((GE ,e,e2)))))) in
 (cons(OR ,
  cons(hd(h),
  gen-guard(tl(discrete-range*))(d)(e)(t)(p))))))
```

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; and
- 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 a state delta representing the execution of the corresponding sequential statement list. This formula and state delta are collected by **gen-case**; the final result returned by **gen-case** is a list of state deltas. The function **gen-guard** converts discrete range lists into formulas representing case selection sets. The function **mk-or(formula$_1$, formula$_2$)** constructs the logical disjunction of two formulas; if one of the formulas is empty, then **mk-or** ignores it and returns the nonempty one.

(SS7) SS ⟦ **LOOP** atmark id seq-stat* opt-id ⟧ (t)(p)(c)(v)(stk)

```
      = let lp-desc = <*LOOP-EXIT* ,id,p,λv,s.c(v)(s)> in
          let stk1 = stk-push(lp-desc)(stk) in
            cons((EQ ,pound(catenate(hd(p),"\pc")),atmark),
            loop-infinite(seq-stat)(id)(seq-stat*)(t)(%(p)(id))(c)(v)(stk1))

loop-infinite(seq-stat)(id)(seq-stat*)(t)(p)(c)(v)(stk)
= let c1 = λv,stk.
             SS [[ seq-stat* ]] (t)(p)(c2)(v)(stk)
              where
              c2 = λv,stk.
                     loop-infinite(seq-stat)(id)(seq-stat*)(t)(p)(c)(v)(stk) in
  (mk-sd(hd(p))(ε)(ε)(ε)(c1(v)(stk)))


(SS8) SS [[ WHILE atmark id expr seq-stat* opt-id ]] (t)(p)(c)(v)(stk)
      = let lp-desc = <*LOOP-EXIT* ,id,p,λv,s.c(v)(s)> in
          let stk0 = stk-push(lp-desc)(stk) in
            cons((EQ ,pound(catenate(hd(p),"\pc")),atmark),
            loop-while(seq-stat)(id)(expr)(seq-stat*)(t)(%(p)(id))(c)(v)(stk0))

loop-while(seq-stat)(id)(expr)(seq-stat*)(t)(p)(c)(v)(stk)
= R [[ expr ]] (t)(p)(k)(v)(stk)
   where
   k = λ(e,f),v,stk.
        let c1 = λv,stk.
                   SS [[ seq-stat* ]] (t)(p)(c2)(v)(stk)
                    where
                    c2 = λv,stk.
                           loop-while
                             (seq-stat)(id)(expr)(seq-stat*)(t)(p)(c)(v)
                             (stk) in
         (mk-sd
            (hd(p))(cons(e,f))(ε)(ε)(c1(v)(stk)),
          mk-sd
            (hd(p))(cons(mk-not(e),f))(ε)(ε)
            (c(v)(stk-pop(stk))))


(SS9) SS [[ FOR atmark id ref discrete-range seq-stat* opt-id ]] (t)(p)(c)(v)(stk)
      = let d = T [[ ref ]] (t)(p) in
          let lp-desc = <*LOOP-EXIT* ,id,p,
                          λv,s.c(v)(s)> in
           let stk0 = stk-push(lp-desc)(stk) in
            let (direction,expr1 ,expr2) = discrete-range in
             cons((EQ ,pound(catenate(hd(p),"\pc")),atmark),
             R [[ expr1 ]] (t)(p)(k1)(v)(stk0)
              where
              k1 = λ(e1,f1),v1,stk1.
                     R [[ expr2 ]] (t)(p)(k2)(v1)(stk1)
                      where
                      k2 = λ(e2,f2),v2,stk2.
                             let bk-desc = <*BLOCK-EXIT* ,id,p,λv,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)(v2)
                                 (stk-push(bk-desc)(stk2))
```

```
                    where
                    u = λv3,stk3.
                          let  bg-desc = <*BEGIN* ,id,%(p)(id),
                                          λv,s.c1(v)(s)>  in
                           (mk-sd
                              (hd(p))(nconc(f1,f2))(ε)(ε)
                              ((case tag(d)
                                  *INT*
                                  → let  final-iter-val = eval-expr
                                                          (e2)  in
                                       loop-for-int
                                          (seq-stat)(ref)(d)
                                          (direction)
                                          (final-iter-val)
                                          (seq-stat*)(t)(%(p)(id))(c1)
                                          (v3)
                                          (stk-push(bg-desc)(stk3)),
                                  *ENUMTYPE*
                                  → let  initial-iter-val = eval-expr
                                                            (e1)
                                         and final-iter-val = eval-expr
                                                              (e2)
                                         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))
                                          (c1)(v3)
                                          (stk-push
                                             (bg-desc)(stk3)),
                                  OTHERWISE
                                  → impl-error
                                       ("Illegal FOR loop parameter type: ~a",
                                       d))))
                          where
                          c1 = λv4,stk4.
                                block-exit(v4)(stk4))

loop-for-int(seq-stat)(ref)(d)(direction)(final-iter-val)(seq-stat*)(t)(p)(c)(v)(stk)
= E ⟦ ref ⟧ (t)(p)(k)(v)(stk)
    where
    k = λ(e,f),v,stk.
          R ⟦ ref ⟧ (t)(p)(k1)(v)(stk)
           where
           k1 = λ(e1,f1),v1,stk1.
                 let  c0 = λv0,stk0.
                           SS ⟦ seq-stat* ⟧ (t)(p)(c1)(v0)(stk0)
                            where
                            c1 = λv2,stk2.
```

```
                                    (mk-sd
                                        (hd(p))(ε)(ε)((e))
                                        (cons(mk-rel
                                                (d)
                                                ((EQ ,pound(e),
                                                  (direction = TO
                                                   → mk-exp2(ADD ,e1,1),
                                                   mk-exp2(SUB ,e1,1)))),
                                         loop-for-int
                                            (seq-stat)(ref)(d)(direction)
                                            (final-iter-val)(seq-stat*)(t)
                                            (p)(c)(v2)(stk2)))) in
                        (mk-sd
                            (hd(p))
                            (cons(mk-rel
                                    (d)
                                    (((direction = TO → LE , GE ),e1,final-iter-val)),f1))(ε)(ε)(c0(v1)(stk1)),
                         mk-sd
                            (hd(p))
                            (cons(mk-rel
                                    (d)
                                    (((direction = TO → GT , LT ),e1,final-iter-val)),f1))(ε)(ε)
                            (c(v1)(stk1)))

loop-for-enum(seq-stat)(ref)(d)(direction)(parameter-updates)(final-iter-val)(seq-stat*)(t)(p)(c)(v)(stk)
= E [[ ref ]] (t)(p)(k)(v)(stk)
    where
    k = λ(e,f),v,stk.
          R [[ ref ]] (t)(p)(k1)(v)(stk)
           where
           k1 = λ(e1,f1),v1,stk1.
                 let c0 = λv0,stk0.
                          SS [[ seq-stat* ]] (t)(p)(c1)(v0)(stk0)
                           where
                           c1 = λv2,stk2.
                                  (parameter-updates
                                   → (mk-sd
                                         (hd(p))(ε)(ε)((e))
                                         (cons(mk-rel
                                                 (d)
                                                 ((EQ ,pound(e),
                                                   hd(parameter-updates))),
                                          loop-for-enum
                                             (seq-stat)(ref)(d)
                                             (direction)
                                             (tl(parameter-updates))
                                             (final-iter-val)(seq-stat*)
                                             (t)(p)(c)(v2)(stk2)))),
                                      (mk-sd
                                         (hd(p))(ε)(ε)(ε)
                                         (c(v2)(stk2)))) in
                        (mk-sd
                            (hd(p))
                            (cons(mk-rel
                                    (d)
                                    (((direction = TO → LE , GE ),e1,final-iter-val)),f1))(ε)(ε)(c0(v1)(stk1)),
                         mk-sd
```

```
        (hd(p))
        (cons(mk-rel
                (d)
                (((direction = TO → GT , LT ),e1,final-iter-val)),f1))(ε)(ε)
        (c(v1)(stk1)))
```

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* state delta 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 state deltas, one of which is recursive. The recursive state delta 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 state delta represents the case where the loop is exited "naturally" by virtue of its control expression having the value **false**. The postcondition of this state delta 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.

```
(SS10) SS [[ EXIT atmark opt-dotted-name opt-expr ]] (t)(p)(c)(v)(stk)
       = cons((EQ ,pound(catenate(hd(p),"\pc")),atmark),
         let expr = opt-expr in
          R [[ expr ]] (t)(p)(k)(v)(stk)
          where
          k = λ(e,f),v1,stk1.
                let loop-name = (null(opt-dotted-name)→ ε,
                                 last(opt-dotted-name)) in
            (null(e)→ exit(loop-name)(v1)(stk),
             (mk-sd
                (hd(p))(cons(e,f))(ε)(ε)
                (c1(v1)(stk1)
                  where c1 = λv2,stk2.exit(loop-name)(v2)(stk2)),
              mk-sd
                (hd(p))(cons(mk-not(e),f))(ε)(ε)
                (c(v1)(stk1)))))

exit(loop-name)(v)(stk)
= let <tg,id,p,g> = hd(stk) in
    (case tg
       *LOOP-EXIT*
       → (¬null(loop-name)∧ id ≠ loop-name → exit(loop-name)(v)(stk-pop(stk)),
          g(v)(stk-pop(stk))),
       *UNDECLARE* → g(λvv,s.exit(loop-name)(vv)(s))(v)(stk),
```

```
(*BEGIN* ,*BLOCK-EXIT* ) → exit(loop-name)(v)(stk-pop(stk)),
OTHERWISE → execution-error("*** EXECUTION ERROR -- ILLEGAL EXIT ***"))
```

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 implict 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 state deltas 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 state delta's postcondition. The other state delta 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 state delta is not even generated.

```
(SS11) SS [[ CALL atmark ref ]] (t)(p)(c)(v)(stk)
       = cons((EQ ,pound(catenate(hd(p),"\pc")),atmark),
         let basic-ref = second(ref) in
          let (tg,q,id) = hd(basic-ref) in
           let d = t(q)(id) in
            let expr* = second(second(basic-ref)) in
             gen-call(ref)(d)(expr*)(tt)(ff)(t)(p)(c)(v)(stk))

gen-call(ref)(d)(expr*)(gen-guards?)(no-unbind?)(t)(p)(c)(v)(stk)
= let z = hd(p)
      and q = %(path(d))(idf(d)) in
   let (decl*,seq-stat*) = body(d) in
    bind-parameters(ref)(d)(expr*)(gen-guards?)(t)(p)(u)(v)(stk)
     where
     u = λv1,stk1.
          let sp-desc = <*SUBPROGRAM-RETURN* ,idf(d),p,λv,s.c(v)(s)>
              and par-desc = <*UNDECLARE* ,collect-allpars(extract-pars(d)(t)),p,
                              λc1,v4,stk4.
                               (mk-sd
                                   (z)(ε)(ε)(ε)
                                   (cons((EQ ,pound(catenate(z,"\pc")),
                                         (EXITED ,$(path(d))(idf(d)))),
                                    unbind-parameters
```

```
                                    (ref)(d)(expr*)(no-unbind?)(t)(p)(c1)
                                    (v4)(stk4))),
                                mk-sd
                                  (z)
                                  (((EQ ,dot(catenate(z,"\pc")),
                                     (EXITED ,$(path(d))(idf(d))))))(ε)(ε)
                                  (unbind-parameters
                                    (ref)(d)(expr*)(no-unbind?)(t)(p)(c1)(v4)
                                    (stk4)))> in
            let stk5 = stk-push(par-desc)(stk-push(sp-desc)(stk1)) in
             (mk-sd
                (z)(ε)(ε)(ε)
                (cons((EQ ,pound(catenate(z,"\pc")),
                      (AT ,$(path(d))(idf(d)))),
                 u2(v1)(stk5))))
              where
              u2 = λv6,stk6.
                    (null(characterizations(d))
                     → D ⟦ decl* ⟧ (t)(q)(u1)(v6)(stk6),
                     null(seq-stat*)
                     → gen-characterizations
                          (ε)(p)(characterizations(d))(c2)(v6)(stk6)
                        where
                        c2 = λv7,stk7.
                              unbind-parameters
                                (ref)(d)(expr*)(no-unbind?)(t)(p)(c3)
                                (v7)(stk7)
                        where c3 = λv8,stk8.block-exit(v8)(stk8),
                     impl-error
                       ("Offline Characterization not yet implemented
                         for procedures with nonempty bodies !"))
            where
            u1 = λv2,stk2.
                  let bg-desc = <*BEGIN* ,idf(d),q,λvv,s.c1(vv)(s)> in
                   SS ⟦ seq-stat* ⟧ (t)(q)(c1)(v2)(stk-push(bg-desc)(stk2))
                   where c1 = λv3,stk3.block-exit(v3)(stk3)

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)

bind-parameters(ref)(d)(actuals)(gen-guards?)(t)(p)(u)(v)(stk)
= let z = hd(p)
    and q = %(path(d))(idf(d))
    and par-assoc-list = extract-pars(d)(t) in
   (null(par-assoc-list)→ u(v)(stk),
```

```
    let all-formals = get-qids(collect-allpars(par-assoc-list))(t)(q)
        and to-formals = get-qids(collect-topars(par-assoc-list))(t)(q)
        and type-descriptors = collect-topars-types(par-assoc-list)
        and from-actuals = collect-fromargs(actuals)(par-assoc-list) in
     let v0 = push-universe(v)(z)(all-formals) in
      let qual-all-formals = get-qualified-ids(all-formals)(v0)
          and qual-to-formals = get-qualified-ids(to-formals)(v0) in
       (mk-decl-sd
          (z)(ε)(ε)((z))
          (nconc
             (mk-qual-id-coverings(all-formals)(qual-all-formals)(z)(v)(t),
             mk-par-decls(q)(par-assoc-list)(p)(t)(v0),
             (null(qual-to-formals)→ u(v0)(stk),
              let expr* = from-actuals in
               MR ⟦ expr* ⟧ (t)(p)(h)(v0)(stk)
               where
               h = λ(e*,f*),v1,stk1.
                     u1(v1)(stk1)
                      where
                      u1 = λv2,stk2.
                            let precondition = (gen-guards?
                                                → nconc
                                                    (mk-constraint-guards
                                                       (e*)(type-descriptors)
                                                       (t)(p)(v2)(stk2),f*),
                                                f*) in
                             (mk-decl-sd
                                (z)(precondition)(ε)(qual-to-formals)
                                (nconc
                                   (assign-multiple
                                      (qual-to-formals)(type-descriptors)(e*),
                                   u(v2)(stk2)))))))))

extract-pars(d)(t)
= let signatures = signatures(d) in
    let signature = hd(signatures) in
     (null(tl(signatures))→ pars(signature),
      extract-poly-pars(pars(signature))(t))

extract-poly-pars(par-assoc-list)(t)
= (null(par-assoc-list)→ ε,
   let par = hd(par-assoc-list) in
    cons((hd(par),(hd(second(par)),poly-type-desc(t))),
    extract-poly-pars(tl(par-assoc-list))(t)))

collect-allpars(par-assoc-list)
= (null(par-assoc-list)→ ε,
   let (id,w) = hd(par-assoc-list) in
    cons(id,collect-allpars(tl(par-assoc-list))))

collect-topars(par-assoc-list)
= (null(par-assoc-list)→ ε,
   let (id,w) = hd(par-assoc-list) in
    (ref-mode(tmode(w))∈ (REF VAL)
     → cons(id,collect-topars(tl(par-assoc-list))),
     collect-topars(tl(par-assoc-list))))
```

```
collect-fromargs(actuals)(par-assoc-list)
= (null(par-assoc-list)→ ε,
    let (id,w) = hd(par-assoc-list) in
     (ref-mode(tmode(w))∈ (REF VAL)
      → cons(hd(actuals),collect-fromargs(tl(actuals))(tl(par-assoc-list))),
      collect-fromargs(tl(actuals))(tl(par-assoc-list))))

collect-frompars(par-assoc-list)(p)
= (null(par-assoc-list)→ ε,
    let (id,w) = hd(par-assoc-list) in
     (ref-mode(tmode(w))∈ (REF OUT)
      → cons((REF ,((SREF ,p,id))),
         collect-frompars(tl(par-assoc-list))(p)),
      collect-frompars(tl(par-assoc-list))(p)))

collect-toargs(actuals-ids)(par-assoc-list)
= (null(actuals-ids)→ ε,
    let (id,w) = hd(par-assoc-list) in
     (ref-mode(tmode(w))∈ (REF OUT)
      → cons(hd(actuals-ids),collect-toargs(tl(actuals-ids))(tl(par-assoc-list))),
      collect-toargs(tl(actuals-ids))(tl(par-assoc-list))))

collect-topars-types(par-assoc-list)
= (null(par-assoc-list)→ ε,
    let (id,w) = hd(par-assoc-list) in
     (ref-mode(tmode(w))∈ (REF VAL)
      → cons(tdesc(w),collect-topars-types(tl(par-assoc-list))),
      collect-topars-types(tl(par-assoc-list))))

collect-toargs-types(actuals)(par-assoc-list)(t)(p)
= (null(actuals)→ ε,
    let (id,w) = hd(par-assoc-list) in
     (ref-mode(tmode(w))∈ (REF OUT)
      → let expr = hd(actuals) in
          cons(T ⟦ expr ⟧ (t)(p),
          collect-toargs-types(tl(actuals))(tl(par-assoc-list))(t)(p)),
      collect-toargs-types(tl(actuals))(tl(par-assoc-list))(t)(p)))

collect-guards-for-exprs(expr*)(d*)(t)(p)(v)(stk)
= MR ⟦ expr* ⟧ (t)(p)(h)(v)(stk)
   where h = λ(e*,f*),v,stk.mk-constraint-guards(e*)(d*)(t)(p)(v)(stk)

mk-constraint-guards(e*)(d*)(t)(p)(v)(stk)
= (null(e*)→ ε,
    let e = hd(e*)
        and d = hd(d*) in
     (¬(tag(d)∈ (*INT* *SUBTYPE* *INT_TYPE*) )
      → mk-constraint-guards(tl(e*))(tl(d*))(t)(p)(v)(stk),
      (tag(d)= *INT* → mk-constraint-guards(tl(e*))(tl(d*))(t)(p)(v)(stk),
       let dd = (tag(d)= *SUBTYPE* → base-type(d), parent-type(d))
           and expr1 = type-tick-low(d)
           and expr2 = type-tick-high(d) in
        R ⟦ expr1 ⟧ (t)(p)(k1)(v)(stk)
         where
         k1 = λ(e1,f1),v1,stk1.
                R ⟦ expr2 ⟧ (t)(p)(k2)(v1)(stk1)
                 where
                 k2 = λ(e2,f2),v2,stk2.
                        nconc
                           ((e1 → (mk-rel(dd)((LE ,e1,e))), ε),
                           (e2 → (mk-rel(dd)((LE ,e,e2))), ε),
                           mk-constraint-guards(tl(e*))(tl(d*))(t)(p)(v)(stk)))))
```

```
mk-par-decls(q)(par-assoc-list)(p)(t)(v)
= (null(par-assoc-list)→ ε,
   let (id,w) = hd(par-assoc-list) in
    cons((DECLARE ,qualified-id(qid(t(q)(id)))(v),mk-type-spec(tdesc(w))(t)(p)),
    mk-par-decls(q)(tl(par-assoc-list))(p)(t)(v)))

assign-multiple(duqn*)(type-descriptors)(e*)
= (null(duqn*)→ ε,
   let target = hd(duqn*)
       and d = hd(type-descriptors)
       and source = hd(e*) in
    nconc
       (assign(d)((target,source)),
       assign-multiple(tl(duqn*))(tl(type-descriptors))(tl(e*))))

unbind-parameters(ref)(d)(actuals)(no-unbind?)(t)(p)(c)(v)(stk)
= let z = hd(p)
      and q = %(path(d))(idf(d))
      and par-assoc-list = extract-pars(d)(t) in
   let all-formals = get-qids(collect-allpars(par-assoc-list))(t)(q) in
    let qual-all-formals = get-qualified-ids(all-formals)(v) in
     (null(qual-all-formals)
      → (mk-sd
            (z)(ε)(ε)(ε)
            (c(pop-universe(v)(all-formals))(stk-pop(stk)))),
      (no-unbind?
       → (mk-sd
             (z)(ε)(ε)(cons(z,qual-all-formals))
             (cons(mk-cover-already((dot(z),cons(pound(z),qual-all-formals)))(t),
              cons(mk-undeclare(qual-all-formals),
              c(pop-universe(v)(all-formals))(stk-pop(stk))))))),
       let expr1* = actuals in
        MR [[ expr1* ]] (t)(p)(h1)(v)(stk)
        where
        h1 = λ(e1*,f1*),v1,stk1.
               let to-actuals = collect-toargs(underef(e1*))(par-assoc-list) in
                let qual-to-actuals = get-qualified-ids(to-actuals)(v1) in
                 (null(qual-to-actuals)
                  → (mk-sd
                        (z)(ε)(ε)(cons(z,qual-all-formals))
                        (cons(mk-cover-already
                                 ((dot(z),cons(pound(z),qual-all-formals)))(t),
                         cons(mk-undeclare(qual-all-formals),
                         c(pop-universe(v1)(all-formals))(stk-pop(stk1))))))),
                  let from-formals = collect-frompars(par-assoc-list)(q)
                      and type-descriptors = collect-toargs-types
                                                (actuals)(par-assoc-list)(t)(p) in
                   let expr2* = from-formals in
                    MR [[ expr2* ]] (t)(q)(h2)(v1)(stk1)
                    where
                    h2 = λ(e2*,f2*),v2,stk2.
                           u1(v2)(stk2)
                            where
                            u1 = λv3,stk3.
                                   let guard* = nconc
                                                  (collect-guards-for-exprs
                                                     (from-formals)
```

```
                                (type-descriptors)(t)
                                (q)(v3)(stk3),f1*,
                            f2*) in
(mk-sd
   (z)(guard*)(ε)(qual-to-actuals)
   (nconc
      (assign-multiple
         (qual-to-actuals)
         (type-descriptors)(e2*),
      u2(v3)(stk3))))
 where
 u2 = λv4,stk4.
       (mk-sd
          (z)(ε)(ε)
          (cons(z,qual-all-formals))
          (cons(mk-cover-already
                   ((dot(z),
                     cons(pound(z),
                     qual-all-formals)))(t),
           cons(mk-undeclare
                   (qual-all-formals),
           c(pop-universe
                (v4)
                (all-formals))
           (stk-pop(stk4)))))))))
```

```
underef(actuals)
= (null(actuals)→ ε,
   let actual = hd(actuals) in
    (dotted-expr-p(actual)→ cons(second(actual),underef(tl(actuals))),
     cons(actual,underef(tl(actuals)))))
```

```
mk-cover-already(id,lst)(t)
= (new-declarations()→ mk-rel(univint-type-desc(t))((EQ ,hd(lst),id)),
   mk-cover(id,lst))
```

```
mk-undeclare(lst) = cons(UNDECLARE ,lst)
```

Procedure calls in Stage 4 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 [3].

- 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) SS [[ RETURN atmark opt-expr ]] (t)(p)(c)(v)(stk)
       = cons((EQ ,pound(catenate(hd(p),"\pc")),atmark),
         let expr = opt-expr in
          R [[ expr ]] (t)(p)(k)(v)(stk)
          where
          k = λ(e,f),v,stk.
                (null(e)→ (mk-sd(hd(p))(ε)(ε)(ε)(return(v)(stk))),
                 let d = context(t)(p) in
                  let result-d = tdesc(extract-rtype(d)) in
                   let precondition = nconc
                                        (mk-constraint-guards
                                           ((e))((result-d))(t)(p)
                                           (v)(stk),f) in
                    (mk-sd
                       (hd(p))(precondition)(ε)
                       ((qualified-id(qid(d))(v)))
                       (nconc
                          (assign(result-d)((qualified-id(qid(d))(v),e)),
                          c1(v)(stk)
                            where c1 = λv,stk.return(v)(stk))))))

return(v)(stk)
= let <tg,qname,p,g> = hd(stk) in
    (case tg
       *UNDECLARE* → g(λvv,s.return(vv)(s))(v)(stk),
       (*BLOCK-EXIT* ,*SUBPROGRAM-RETURN* ) → g(v)(stk-pop(stk)),
       (*BEGIN* ,*LOOP-EXIT* ,*PACKAGE-BODY-EXIT* ) → return(v)(stk-pop(stk)),
       OTHERWISE
       → impl-error("Bad execution stack descriptor tag in context: ~a",tg))

context(t)(path)
= let d = t(path)(*UNIT* ) in
    (d = *UNBOUND* → context(t)(rest(path)),
     (case tag(d)
        (*PROCEDURE* ,*FUNCTION* ,*PACKAGE* ) → t(rest(path))(last(path)),
        OTHERWISE → context(t)(rest(path))))
```

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

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) SS [[ WAIT atmark ref* opt-expr1 opt-expr2 ]] (t)(p)(c)(v)(stk)
       = cons((EQ ,pound(catenate(hd(p),"\pc")),atmark),
         let c1 = λv,stk.
                  (mk-sd
                     (hd(p))(ε)(ε)(ε)
                     ((make-vhdl-try-resume-next-process(hd(p))))) in
          ME [[ ref* ]] (t)(p)(h)(v)(stk)
           where
           h = λ(e*,f*),v,stk.
                 let expr1 = opt-expr1 in
                  R [[ expr1 ]] (t)(p)(k1)(v)(stk)
                  where
                  k1 = λ(e1,f1),v,stk.
                         let expr2 = opt-expr2 in
                          R [[ expr2 ]] (t)(p)(k2)(v)(stk)
                          where
                          k2 = λ(e2,f2),v,stk.
                                 let process-id = last(find-process-env
                                                        (t)(p)) in
                                  let process-qid = qid(lookup
                                                         (t)(p)
                                                         (process-id)) in
                                   (mk-sd
                                      (hd(p))(nconc(f1,f2,f*))(ε)(ε)
                                      ((make-vhdl-process-suspend
                                          (process-qid)
                                          (get-signals(e*))
                                          (e1)(e2)(c)(v)(stk)(c1(v)(stk))))))
```

```
find-process-env(t)(p)
= (null(p)∨ 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))))
```

8.4.11 Waveforms and Transactions

```
(W1) W ⟦ WAVE transaction+ ⟧ (d)(t)(p)(wave-cont)(v)(stk)
     = TRM ⟦ transaction+ ⟧ (d)(t)(p)(wave-cont)(v)(stk)

(TRM0) TRM ⟦ ε ⟧ (d)(t)(p)(wave-cont)(v)(stk) = wave-cont((ε,ε))(v)(stk)

(TRM1) TRM ⟦ transaction transaction* ⟧ (d)(t)(p)(wave-cont)(v)(stk)
       = TR ⟦ transaction ⟧ (d)(t)(p)(trans-cont)(v)(stk)
         where
         trans-cont = λ(trans,guard),v,stk.
                      TRM ⟦ transaction* ⟧ (d)(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 ⟧ (d)(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 ⟦ expr2 ⟧ (t)(p)(k1)(v)(stk)
               where
               k1 = λ(e2,f2),v,stk.
                    trans-cont
                      ((mk-transaction-for-update(e1)(e2),
                        nconc
                          (mk-constraint-guards
                            ((e1))((d))(t)(p)(v)(stk),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)
```

8.4.12 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.

```
(ME0) ME ⟦ ε ⟧ (t)(p)(h)(v)(stk) = h((ε,ε))(v)(stk)
```

```
(ME1) ME [[ ref ref* ]] (t)(p)(h)(v)(stk)
      = E [[ ref ]] (t)(p)(k)(v)(stk)
          where
          k = λ(e,f),v1,stk1.
                ME [[ ref* ]] (t)(p)(h1)(v1)(stk1)
                  where h1 = λ(e*,f*),v2,stk2.h((cons(e,e*),nconc(f,f*)))(v2)(stk2)

(MR0) MR [[ ε ]] (t)(p)(h)(v)(stk) = h((ε,ε))(v)(stk)
(MR1) MR [[ expr expr* ]] (t)(p)(h)(v)(stk)
      = R [[ expr ]] (t)(p)(k)(v)(stk)
          where
          k = λ(e,f),v1,stk1.
                MR [[ expr* ]] (t)(p)(h1)(v1)(stk1)
                  where h1 = λ(e*,f*),v2,stk2.h((cons(e,e*),nconc(f,f*)))(v2)(stk2)
```

The translation of a (possibly empty) multiple expression list yields a list of translated expressions and a corresponding list of guard formulas.

```
(E1) E [[ REF modifier+ ]] (t)(p)(k)(v)(stk)
     = let basic-ref = modifier+ in
         let (basic-name,d) = gen-basic-name(basic-ref)(t)(v) in
           gen-name(ref)(basic-name)(ε)(d)(tl(basic-ref))(t)(p)(k)(v)(stk)

gen-basic-name(basic-ref)(t)(v)
= let (tg,q,id) = hd(basic-ref) in
    let d = t(q)(id) in
      (case tag(d)
         (*PROCEDURE* ,*FUNCTION* ) → (qualified-id(qid(d))(v),d),
         OTHERWISE → (qualified-id(qid(d))(v),tdesc(type(d))))

gen-name(ref)(e)(f)(d)(ref-tail)(t)(p)(k)(v)(stk)
= (null(ref-tail)→ k((e,f))(v)(stk),
    let modifier = hd(ref-tail) in
      let (tg,isp) = modifier in
        (case tg
           INDEX → gen-array-ref(isp)(e)(f)(d)(t)(p)(c)(v)(stk),
           SELECTOR → gen-record-ref(isp)(e)(f)(d)(c)(v)(stk),
           PARLIST → gen-function-call(ref)(isp)(d)(t)(p)(c)(v)(stk),
           OTHERWISE
           → impl-error("Unrecognized Stage 4 VHDL reference modifier tag: ~a",tg))
      where
      c = λ(e1,f1,d1),v,stk.
            gen-name(ref)(e1)(f1)(d1)(tl(ref-tail))(t)(p)(k)(v)(stk))

gen-array-ref(expr)(e)(f)(d)(t)(p)(c)(v)(stk)
= R [[ expr ]] (t)(p)(k)(v)(stk)
    where
    k = λ(e0,f0),v,stk.
          c(((ELEMENT ,e,e0),
             nconc
               (f,f0,
               (null(ub(d))
                → (mk-rel(univint-type-desc(t))((GE ,e0,(ORIGIN ,e)))),
                (mk-rel(univint-type-desc(t))((GE ,e0,(ORIGIN ,e))),
                 mk-rel
                   (univint-type-desc(t))
                   ((LE ,e0,
                     mk-exp2(SUB ,mk-exp2(ADD ,(ORIGIN ,e),(RANGE ,e)),1))))))),
            elty(d)))(v)(stk)
```

```
gen-record-ref(id)(e)(f)(d)(c)(v)(stk)
= c((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)(c)(v)(stk)
= declare-function-name(d)(t)(p)(u)(v)(stk)
   where
   u = λv,stk.
        gen-call(ref)(d)(expr*)(tt)(tt)(t)(p)(c1)(v)(stk)
         where
         c1 = λv,stk.
                c((qualified-id(qid(d))(v),ε,tdesc(extract-rtype(d))))(v)(stk)

declare-function-name(d)(t)(p)(u)(v)(stk)
= let q = path(d)
      and dd = tdesc(extract-rtype(d)) in
   let z = hd(q) in
    let suqn+ = get-qids((idf(d)))(t)(q) in
     let v1 = push-universe(v)(z)(suqn+) in
      let duqn+ = get-qualified-ids(suqn+)(v1) in
       let dc-desc = <*UNDECLARE* ,idf(d),q,
                        λu1,v2,stk2.
                         undeclare-function-name
                           (suqn+)(duqn+)(z)(t)(u1)(v2)(stk2)> in
        (mk-decl-sd
            (z)(ε)(ε)((z))
            (nconc
                (mk-qual-id-coverings(suqn+)(duqn+)(z)(v)(t),
                mk-scalar-nonsignal-dec-post
                   (ε)((duqn+,ε,dd))(t)(q)(u)(v1)(stk-push(dc-desc)(stk)))))

undeclare-function-name(suqn+)(duqn+)(z)(t)(u)(v)(stk)
= (mk-sd
     (z)(ε)(ε)(cons(z,duqn+))
     (cons(mk-cover-already((dot(z),cons(pound(z),duqn+)))(t),
      cons(mk-undeclare(duqn+),
      u(pop-universe(v)(suqn+))(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) $\underline{\mathbf{R}}$ ⟦ 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) $\underline{\mathbf{R}}$ ⟦ **FALSE** ⟧ (t)(p)(k)(v)(stk) = k((**FALSE** ,ε))(v)(stk)

(R2) $\underline{\mathbf{R}}$ ⟦ **TRUE** ⟧ (t)(p)(k)(v)(stk) = k((**TRUE** ,ε))(v)(stk)

(R3) $\underline{\mathbf{R}}$ ⟦ **BIT** bitlit ⟧ (t)(p)(k)(v)(stk) = k(($\underline{\mathbf{B}}$ ⟦ bitlit ⟧ ,ε))(v)(stk)

(R4) $\underline{\mathbf{R}}$ ⟦ **NUM** constant ⟧ (t)(p)(k)(v)(stk) = k(($\underline{\mathbf{N}}$ ⟦ constant ⟧ ,ε))(v)(stk)

(R5) $\underline{\mathbf{R}}$ ⟦ **TIME** constant **FS** ⟧ (t)(p)(k)(v)(stk) = k(($\underline{\mathbf{N}}$ ⟦ constant ⟧ ,ε))(v)(stk)

(R6) $\underline{\mathbf{R}}$ ⟦ **CHAR** constant ⟧ (t)(p)(k)(v)(stk) = k((expr,ε))(v)(stk)

(R7) $\underline{\mathbf{R}}$ ⟦ **ENUMLIT** id ⟧ (t)(p)(k)(v)(stk) = k((id,ε))(v)(stk)

(R8) $\underline{\mathbf{R}}$ ⟦ **BITSTR** bit-lit* ⟧ (t)(p)(k)(v)(stk)
= **let** expr* = bit-lit* **in**
$\underline{\mathbf{MR}}$ ⟦ expr* ⟧ (t)(p)(k)(v)(stk)

(R9) $\underline{\mathbf{R}}$ ⟦ **STR** char-lit* ⟧ (t)(p)(k)(v)(stk)
= **let** expr* = char-lit* **in**
$\underline{\mathbf{MR}}$ ⟦ expr* ⟧ (t)(p)(k)(v)(stk)

(R10) $\underline{\mathbf{R}}$ ⟦ **REF** modifier$^+$ ⟧ (t)(p)(k)(v)(stk)
= **let** ref = expr **in**
$\underline{\mathbf{E}}$ ⟦ ref ⟧ (t)(p)(k_1)(v)(stk)
where k_1 = λ(e,f),v_1,stk_1.k((dot(e),f))(v_1)(stk_1)

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.

(R11) $\underline{R}$ ⟦ **PAGGR** expr* ⟧ (t)(p)(k)(v)(stk) = $\underline{MR}$ ⟦ expr* ⟧ (t)(p)(k)(v)(stk)

(R12) $\underline{R}$ ⟦ **TYPECONV** expr type-mark ⟧ (t)(p)(k)(v)(stk)
= **let** d = lookup-desc(type-mark)(t)(p) **in**
$\underline{R}$ ⟦ expr ⟧ (t)(p)(k_1)(v)(stk)
where
k_1 = λ(e,f),v,stk.
let constraint-guard* = mk-constraint-guards
((e))((d))(t)(p)(v)(stk) **in**
(null(constraint-guard*)→ k((e,f))(v)(stk),
(mk-sd(hd(p))(constraint-guard*)(ε)(ε)(k((e,f))(v)(stk))))

(R13) $\underline{R}$ ⟦ unary-op expr ⟧ (t)(p)(k)(v)(stk)
= $\underline{R}$ ⟦ expr ⟧ (t)(p)(k_1)(v)(stk)
where k_1 = λ(e,f),v_1,stk_1.k((mk-exp1(unary-op,e),f))(v_1)(stk_1)

mk-exp1(unary-op,e)
= (case unary-op
NOT → (**NOT** ,e),
BNOT → (**USNOT** ,e),
PLUS → e,
NEG → (**MINUS** ,e),
ABS → (**ABS** ,e),
(**RNEG** ,**RABS**) → (unary-op,e),
OTHERWISE
→ impl-error("**Unrecognized Stage 4 VHDL unary operator: ~a**",unary-op))

(R14) $\underline{R}$ ⟦ binary-op $expr_1$ $expr_2$ ⟧ (t)(p)(k)(v)(stk)
= $\underline{R}$ ⟦ $expr_1$ ⟧ (t)(p)(k_1)(v)(stk)
where
k_1 = λ(e_1,f_1,v_1,stk_1).
$\underline{R}$ ⟦ $expr_2$ ⟧ (t)(p)(k_2)(v_1)(stk_1)
where
k_2 = λ(e_2,f_2),v_2,stk_2.
k((mk-exp2(binary-op,e_1,e_2),nconc(f_1,f_2)))(v_2)(stk_2)

(R15) $\underline{R}$ ⟦ relational-op $expr_1$ $expr_2$ ⟧ (t)(p)(k)(v)(stk)
= $\underline{R}$ ⟦ $expr_1$ ⟧ (t)(p)(k_1)(v)(stk)
where
k_1 = λ(e_1,f_1,v_1,stk_1).
$\underline{R}$ ⟦ $expr_2$ ⟧ (t)(p)(k_2)(v_1)(stk_1)
where
k_2 = λ(e_2,f_2),v_2,stk_2.
let d = $\underline{T}$ ⟦ $expr_1$ ⟧ (t)(p) **in**
k((mk-rel(d)((relational-op,e_1,e_2)),nconc(f_1,f_2)))
(v_2)(stk_2)

8.4.13 Expression Types

The function **mk-rel** (described earlier) requires a type descriptor as its first argument; application of the semantic function **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 **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);

(T0) $\underline{\mathbf{T}}$ [[ϵ]] (t)(p) = void-type-desc(t)

(T1) $\underline{\mathbf{T}}$ [[**FALSE**]] (t)(p) = bool-type-desc(t)

(T2) $\underline{\mathbf{T}}$ [[**TRUE**]] (t)(p) = bool-type-desc(t)

(T3) $\underline{\mathbf{T}}$ [[**BIT** bitlit]] (t)(p) = bit-type-desc(t)

(T4) $\underline{\mathbf{T}}$ [[**NUM** constant]] (t)(p) = univint-type-desc(t)

(T5) $\underline{\mathbf{T}}$ [[**TIME** constant **FS**]] (t)(p) = time-type-desc(t)

(T6) $\underline{\mathbf{T}}$ [[**CHAR** constant]] (t)(p) = char-type-desc(t)

(T7) $\underline{\mathbf{T}}$ [[**ENUMLIT** id]] (t)(p)
= **let** d = lookup-desc-on-path(t)(p)(id) **in**
tdesc(type(d))

(T8) $\underline{\mathbf{T}}$ [[**BITSTR** bit-lit*]] (t)(p) = bitvector-type-desc(t)

(T9) $\underline{\mathbf{T}}$ [[**STR** char-lit*]] (t)(p) = string-type-desc(t)

(T10) $\underline{\mathbf{T}}$ [[**REF** modifier+]] (t)(p)
= **let** basic-ref = modifier+ **in**
get-type-desc(basic-ref)(t)(p)

```
get-type-desc(basic-ref)(t)(p)
= let (tg,q,id) = hd(basic-ref) in
    let d = t(q)(id) in
     (case tag(d)
        (*PROCEDURE* ,*FUNCTION* ,*PROCESS* )
        → process-ref-tail(d)(tl(basic-ref))(t)(p),
        OTHERWISE → process-ref-tail(tdesc(type(d)))(tl(basic-ref))(t)(p))
```

```
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(elty(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 4 VHDL reference modifier tag: ~a",
            hd(modifier))))
```

(T11) $\underline{\mathbf{T}}$ ⟦ **PAGGR** expr* ⟧ (t)(p) = void-type-desc(t)

(T12) $\underline{\mathbf{T}}$ ⟦ **TYPECONV** expr type-mark ⟧ (t)(p) = lookup-desc(type-mark)(t)(p)

(T13) $\underline{\mathbf{T}}$ ⟦ unary-op expr ⟧ (t)(p) = tdesc(restype1(unary-op)(t))

```
restype1(unary-op)(t)
= (case unary-op
     NOT → (VAL ,bool-type-desc(t)),
     BNOT → (VAL ,bit-type-desc(t)),
     (PLUS ,NEG ,ABS ) → (VAL ,univint-type-desc(t)),
     (RNEG ,RABS ) → (VAL ,real-type-desc(t)),
     OTHERWISE
     → impl-error("Unrecognized Stage 4 VHDL unary operator: ~a",unary-op))
```

(T14) $\underline{\mathbf{T}}$ ⟦ binary-op $expr_1$ $expr_2$ ⟧ (t)(p)
= tdesc(restype2(binary-op)(($expr_1$,$expr_2$))(t)(p))

```
restype2(binary-op)(expr1,expr2)(t)(p)
= (case binary-op
     (AND ,NAND ,OR ,NOR ,XOR ) → mk-type((DUMMY VAL) )(bool-type-desc(t)),
     (BAND ,BNAND ,BOR ,BNOR ,BXOR ) → mk-type((DUMMY VAL) )(bit-type-desc(t)),
     (ADD ,SUB ,MUL ,DIV ,MOD ,REM ,EXP ) → mk-type((DUMMY VAL) )(univint-type-desc(t)),
     (RPLUS ,RMINUS ,RTIMES ,RDIV ,REXPT ) → mk-type((DUMMY VAL) )(real-type-desc(t)),
     CONCAT
     → let d1 = T ⟦ expr1 ⟧ (t)(p)
           and d2 = T ⟦ expr2 ⟧ (t)(p) in
       mk-type((DUMMY VAL) )(mk-concat-tdesc(d1)(d2)(t)),
     OTHERWISE
     → impl-error("Unrecognized Stage 4 VHDL binary operator: ~a",binary-op))

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

(T15) $\underline{\mathbf{T}}$ ⟦ relational-op $expr_1$ $expr_2$ ⟧ (t)(p) = bool-type-desc(t)

8.4.14 Primitive Semantic Equations

The following semantic functions are primitive.

(N1) $\underline{\mathbf{N}}$ ⟦ constant ⟧ = constant

(B1) $\underline{\mathbf{B}}$ ⟦ bitlit ⟧ = mk-bit-simp-symbol(bitlit)

mk-bit-simp-symbol(bitlit)
= (case bitlit
  0 → **(BS 0 1)** ,
  1 → **(BS 1 1)** ,
  OTHERWISE → impl-error("Can't construct simp symbol for bit: ~a ",bitlit))

9 Conclusion

A precise and well-documented formal specification of the Stage 4 VHDL translator has been presented in this report. We have completed and exercised a Common Lisp implementation of both translation phases described herein. As the SDVS interface to VHDL continues to expand and mature, our confidence grows in our language translator semantic specification and implementation paradigm.

Stage 4 VHDL represents a robust subset of the VHSIC Hardware Description Language, supporting both behavioral and structural descriptions of digital devices with the inclusion of the following language constructs: design files, design units, configuration declarations, entity declarations, architecture bodies, ports, declarative parts in entity declarations, package `STANDARD` (containing predefined types `BOOLEAN`, `BIT`, `UNIVERSAL_INTEGER`, `INTEGER`, `TIME`, `CHARACTER`, `REAL`, `STRING`, and `BIT_VECTOR`), user-defined packages, `USE` clauses, generics, component declarations, generic and port maps, array type declarations, certain predefined attributes, enumeration types, subtypes of scalar types, integer type definitions, type conversions, `BLOCK` statements, `PROCESS` statements, concurrent signal assignment statements, component instantiation statements, subprograms (procedures and functions), `IF` and `CASE` statements, `WHILE` and `FOR` loops, octal and hexadecimal representations of bitstrings, and general expressions of type `TIME` in `AFTER` clauses.

Much of our work henceforth will focus on applying SDVS and the VHDL translator to the formal verification of realistic VHDL hardware descriptions. Indeed, we have already made significant steps in this direction in fiscal year 1994. We identified VHDL descriptions suitable for a Stage 4 VHDL verification exercise, developed in-house at the National Security Agency. These specify a set of commercial standard parts, the *Am7968/Am7969 TAXIchip™ (Transparent Asynchronous Xmitter-Receiver Interface) Integrated Circuits* designed by Advanced Micro Devices, Inc. (AMD). The TAXIchip Am7968 Transmitter/Am7969 Receiver chipset constitutes a general-purpose interface for high-speed serial communication between two parallel-data hosts, and is used in a prototype cryptographic device currently being built by NSA.

We wrote formal state delta specifications for simplified versions of these descriptions, as well as a specification for a combined system in which the output of the transmitter is input to the receiver, and have completed proofs that the descriptions meet several of these specifications.

In fiscal year 1995, we intend to proceed by incrementally incorporating additional features of the original TAXIchip descriptions into the simplified descriptions we have produced, and by attempting to prove successively more interesting properties of the latter. After implementing a few, relatively minor, enhancements to the VHDL translator — principally, a subset of the IEEE `STD_LOGIC_1164` multivalue logic system — not much additional effort will be required to formulate and prove specifications of the TAXIchip VHDL as originally given.

References

[1] J. V. Cook, I. V. Filippenko, B. H. Levy, L. G. Marcus, and T. K. Menas, "Formal Computer Verification in the State Delta Verification System (SDVS)," in *Proceedings of the AIAA Computing in Aerospace Conference*, (Baltimore, Maryland), pp. 77–87, American Institute of Aeronautics and Astronautics, October 1991.

[2] B. Levy, I. Filippenko, L. Marcus, and T. Menas, "Using the State Delta Verification System (SDVS) for Hardware Verification," in *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience: Nijmegen, The Netherlands* (ed. V. Stavridou, T. F. Melham, and R. T. Boute), pp. 337–360, North-Holland, June 1992.

[3] L. G. Marcus, "SDVS 13 Users' Manual," Technical Report ATR-94(4778)-5, The Aerospace Corporation, September 1994.

[4] T. K. Menas and I. V. Filippenko, "SDVS 13 Tutorial," Technical Report ATR-94(4778)-6, The Aerospace Corporation, September 1994.

[5] B. H. Levy, "Feasibility of Hardware Verification Using SDVS," Technical Report ATR-88(3778)-9, The Aerospace Corporation, September 1988.

[6] IEEE, *Standard VHDL Language Reference Manual*, 1988. IEEE Std. 1076-1987.

[7] D. F. Martin and J. V. Cook, "Adding Ada Program Verification Capability to the State Delta Verification System (SDVS)," in *Proceedings of the 11th National Computer Security Conference*, National Bureau of Standards/National Computer Security Center, October 1988.

[8] T. Aiken, I. Filippenko, B. Levy, and D. Martin, "A Formal Description of the Incremental Translation of Core VHDL into State Deltas in the State Delta Verification System (SDVS)," Technical Report ATR-89(4778)-9, The Aerospace Corporation, September 1989.

[9] I. V. Filippenko, "Example Proof of a Core VHDL Description in the State Delta Verification System (SDVS)," Technical Report ATR-90(5778)-6, The Aerospace Corporation, September 1990.

[10] I. V. Filippenko, "Some Examples of Verifying Core VHDL Hardware Descriptions Using the State Delta Verification System (SDVS)," Technical Report ATR-91(6778)-6, The Aerospace Corporation, September 1991.

[11] I. V. Filippenko, "A Formal Description of the Incremental Translation of Stage 1 VHDL into State Deltas in the State Delta Verification System (SDVS)," Technical Report ATR-91(6778)-7, The Aerospace Corporation, September 1991.

[12] I. V. Filippenko and L. G. Marcus, "Integrating Structural VHDL Hardware Descriptions into the State Delta Verification System (SDVS)," Technical Report ATR-92(8180)-1, The Aerospace Corporation, September 1992.

[13] M. J. C. Gordon, *The Denotational Description of Programming Languages: An Introduction*, (New York: Springer-Verlag, 1979).

[14] I. V. Filippenko, "A Formal Description of the Incremental Translation of Stage 2 VHDL into State Deltas in the State Delta Verification System (SDVS)," Technical Report ATR-92(2778)-4, The Aerospace Corporation, September 1992.

[15] I. V. Filippenko, "A Formal Description of the Incremental Translation of Stage 3 VHDL into State Deltas in SDVS," Technical Report ATR-93(3778)-2, The Aerospace Corporation, September 1993.

[16] J. V. Cook, "The Language for DENOTE (Denotational Semantics Translation Environment)," Technical Report TR-0090(5920-07)-2, The Aerospace Corporation, September 1990.

[17] L. Marcus and B. H. Levy, "Specifying and Proving Core VHDL Descriptions in the State Delta Verification System (SDVS)," Technical Report ATR-89(4778)-5, The Aerospace Corporation, September 1989.

[18] L. Marcus, T. Redmond, and S. Shelah, "Completeness of State Deltas," Technical Report ATR-86(8454)-2, The Aerospace Corporation, September 1986.

[19] T. K. Menas, "The Relation of the Temporal Logic of the State Delta Verification System (SDVS) to Classical First-Order Temporal Logic," Technical Report ATR-90(5778)-10, The Aerospace Corporation, September 1990.

[20] J. E. Doner and J. V. Cook, "Offline Characterization of Procedures in the State Delta Verification System (SDVS)," Technical Report ATR-90(8590)-5, The Aerospace Corporation, September 1990.

[21] J. V. Cook and J. E. Doner, "Example Proofs Using Offline Characterization of Procedures in the State Delta Verification System (SDVS)," Technical Report TR-0090(5920-07)-3, The Aerospace Corporation, September 1990.

Index