VAL to VHDL Transformer
An Implementation Guide

Larry M. Augustin, Benoit A. Gennart,
Youm Huh, David C. Luckham,
Paraminder S. Sahai, Alec G. Stanculescu

Technical Report CSL-TR-89-390

Program Analysis and Verification Group Report No. 41

September 89

This work was supported by the VHSIC Program Office, Department of the Air Force (AFSC), under Contract F33615-86-C-1137, and by the US Defense Advanced Research Projects Agency, under Contract N00039-84-C0211.

VAL to VHDL Transformer
An Implementation Guide

by

Larry M. Augustin, Benoit A. Gennart,
Youm Huh, David C. Luckham,
Paraminder S. Sahai, Alec G. Stanculescu

Technical Report CSL-TR-89-390
Program Analysis and Verification Group Report No. 41
October 1989

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

This report presents one implementation of the VAL semantics. It is based on a transformation from VAL annotated VHDL to self-checking VHDL that is equivalent to the original source from the simulation semantics standpoint.

The transformation is performed as a sequence of tree to tree transformations. The report describes the semantic preserving transformations, as well as the structure of the transformer.

Key Words and Phrases: VHDL, hardware description languages, annotation languages, simulation, tree to tree transformations

Contents

List of Figures

Chapter 1

Introduction

1.1 VAL to VHDL Transformer vs VAL Compiler

In order for VAL annotations to be machine processable, they have to be transformed into some machine executable form. For simulation purposes, there are two possibilities: (1) to transform VAL annotated VHDL into pure VHDL and then use the existing VHDL compiler to transform the pure VHDL into data processable by the simulation engine, or (2) to transform directly the VAL annotated VHDL into data processable by the simulation engine.

This book presents an implementation of the first solution. Its advantage is that it supports VAL in the context of any VHDL simulator. A VAL compiler would work only in conjunction with one VHDL simulation engine.

The disadvantage of the chosen solution is that it is slower since it transforms VAL into VHDL which is in turn transformed into Simulator processable data. Also, being independent of any simulation environment, makes it more difficult to use. The user must keep track of the files that are generated by the transformer and put them manually into the design library of the available simulation environment.

1.2 VAL to VHDL Transformation Principles

The translation from VAL to VHDL is based on the following principles, supported by the VAL language:

1. Principle of Separate Compilation

   The transformation of VAL to VHDL is performed on a per compilation unit basis. Each VAL annotated VHDL compilation unit can be transformed into pure VHDL independently from one another. Note that all semantic analysis is performed after the translation, by the VHDL Analyzer.

2. Principle of Name- Transparency

   The VAL to VHDL translation should be hidden from the VAL writer. In other words, the writer of VAL annotations should not need to know about the details of the VAL to VHDL translation. VAL refers only to library names (entity, architecture, configuration names) visible in the corresponding VHDL context.

In order to support this principle, the VAL configuration annotation contains VAL declarations which state that a given compilation unit has to be used in the original VHDL form or in the generated (translated) form. This way the VAL to VHDL translator has information regarding the actual format of a referenced compilation unit; original VHDL format or translated format.

1.3 Translation Methodology

The translation methodology used consists of using a parser generator to generate a parser for VAL annotated VHDL. The parser is extended with a back end that performs the transformation into pure VHDL. The back end uses an abstract syntax tree package in order to access the parse tree, and a symbol table package in order to keep track of all declarations.

The transformation consists of consecutive tree to tree transformations that eventually lead to the desired tree. Each transformation step can be verified using a VAL/VHDL Pretty Printer that generates the ASCII VAL annotated VHDL corresponding to a given tree. Note that the Pretty Printer can print text even if the tree does not correspond to a valid VAL annotated VHDL. This allows to use the Pretty Printer for the verification of intermediate transformation steps that leave the tree in an illegal state from a VAL/VHDL language standpoint.

1.4 What to expect from the next three chapters

The following chapters describe the implementation of the VAL to VHDL Transformer. The description is top down, from an algorithmic level to the detailed level of actual procedure calls. It emphasizes the reusable packages that are part of the transformer. The use of such packages enhanced the programming productivity.

Chapter 2

Transformat ion Algorithm

The VAL Transformer runs as a preprocessor on an annotated VHDL description to generate a self-checking VHDL description. The annotated VHDL description is parsed into a tree format. The transformation algorithm consists of consecutive tree to tree transformations.

Each transformation step can be verified using a VAL/VHDL Pretty Printer that generates the ASCII VAL annotated VHDL corresponding to a given tree. Note that the Pretty Printer can print text even if the tree does not correspond to a valid VAL annotated VHDL. This allows to use the Pretty Printer for the verification of intermediate transformation steps that leave the tree in an illegal state from a VAL/VHDL language standpoint.

In the transformation process, the Transformer must keep track of all VAL and VHDL declarations. This is done by using the symbol table mechanism described in section 3.3.

VAL can annotate three kinds of compilation units: entities, architectures, and configurations. For each compilation unit, there are different kinds of VHDL compilation units being generated. Otherwise, the transformation paradigms for VAL entity annotations and VAL architecture annotations are similar. Some transformations are independent of VHDL (in terms of VAL only), and some are in terms of VHDL.

As a result of these characteristics of the transformation algorithm, the VAL to VHDL transformation will be presented in five parts:

1. Generation of Transformed Skeleton (section 2.1), describing what VHDL compilation units are generated for each VAL annotated VHDL unit.

2. Transformation of VAL Entity Annotations into Core VAL Entity Annotations, described in section 2.2.

3. Transformation of Core VAL Entity Annotations into VHDL, described in section 2.3.

4. Transformation of VAL Body Annotation, described in section 2.5.

5. Transformation of VAL Configuration Annotation, described in section 2.6.

2.1 Generation of Translation Skeleton

The translation skeleton is designed to implement the scoping and visibility rules of VAL. Consider the problem of observing the operation of a chip on a circuit board. One way of monitoring the chip is to remove it from its socket, plug a specially constructed adapter into the socket, and then plug the chip into the adapter. The adapter senses the signals traveling between the circuit board and the chip's pins. The signals can then be monitored to verify the behavior (and use) of the chip.

For simple cases, the VAL translation algorithm works in just such a manner. The Transformer generates an additional architecture called the *Monitor* that contains an instantiation of the component (architecture) under test. The Monitor has the same pins [1] as the component, and contains a *socket* for the component. The Monitor is plugged into a circuit in place of the component. The component is in turn plugged into the socket in the Monitor.

The Monitor body has visibility over all signals traveling between the actual architecture and the other components in the simulation. In addition, the Monitor contains logic to verify the VAL assertions made about the component under test. This includes maintaining its own separate state (the *VAL state model)* (figure 2.1).

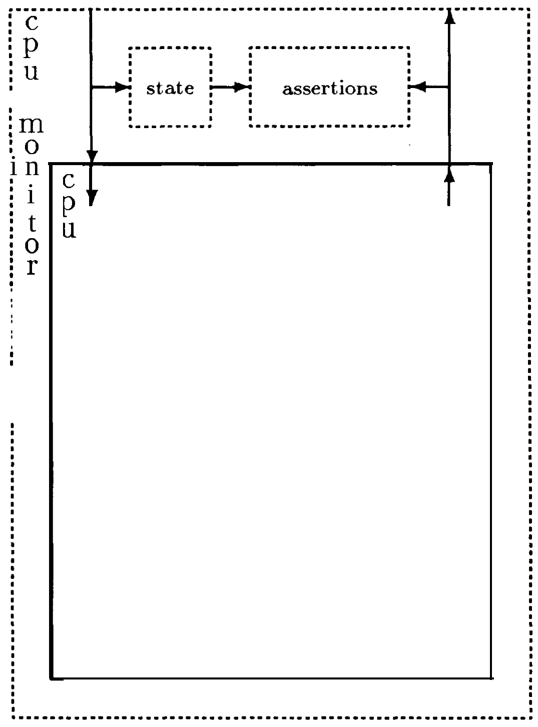

Figure 2.1: Entity annotation

One advantage of this approach (as opposed to simply monitoring the signals that the pins are connected to) is that VAL assertions can separate the value on out ports of the component from

[1] Almost. It may have an additional output pin, as described later, to allow other assertions to probe the monitored architecture's internal state.

the value on the signals that those ports drive [2]. This allows the user to make assertions about the value placed on the port by the entity. [3]

Consider now an architecture containing several components. If a component is annotated, then a monitor can be generated for that component. The mapping annotations in the architecture have visibility over the internal state of the monitor of the component. This allows annotations within the architecture that "map" the architecture's state into the states of its components. The needed visibility over the internal state of the component is provided through an additional out port on the component that carries the component's state.

The design units involved in the translation are shown in Figure 4.1. Assume an entity A exists cont aining VAL annotations. Three design units are generated; two entity declarations and an architecture. The architecture (named STATEMONITOR) contains the VHDL translation of the VAL annotations that appeared in the entity declaration. This includes the annotations which maintain the entity's state model. The ports of architecture STATEMONITOR are the same as for entity A with the addition of an out port of the same type as the entity's state model. This out port is used to provide visibility over the state of components of type A to any annotations within any architecture that instantiates a component of type A. The generated entity VALOUT_A declares the entity for STATEMONITOR.

Architecture STATEMONITOR contains a component SOCKET having the same ports as entity A with the addition of an in port of the same type as the entity's state model. A translated version of the original architecture body T of A is plugged into this socket. Because the entity's state is passed into the SOCKET through a port, it is visible to annotations within the architectural body. The translated version of T, VAL_T, contains a translation of the VAL annotations appearing in the architecture into VHDL. Its entity interface is described by VALIN_A.

[2]The value placed on a port by an entity does not necessarily equal the value on a signal connected to that port because bus resolution may come into play.

[3]In VHDL, a port of mode out, is not readable within the architecture. Therefore assertions about out mode ports cannot be made in VHDL.

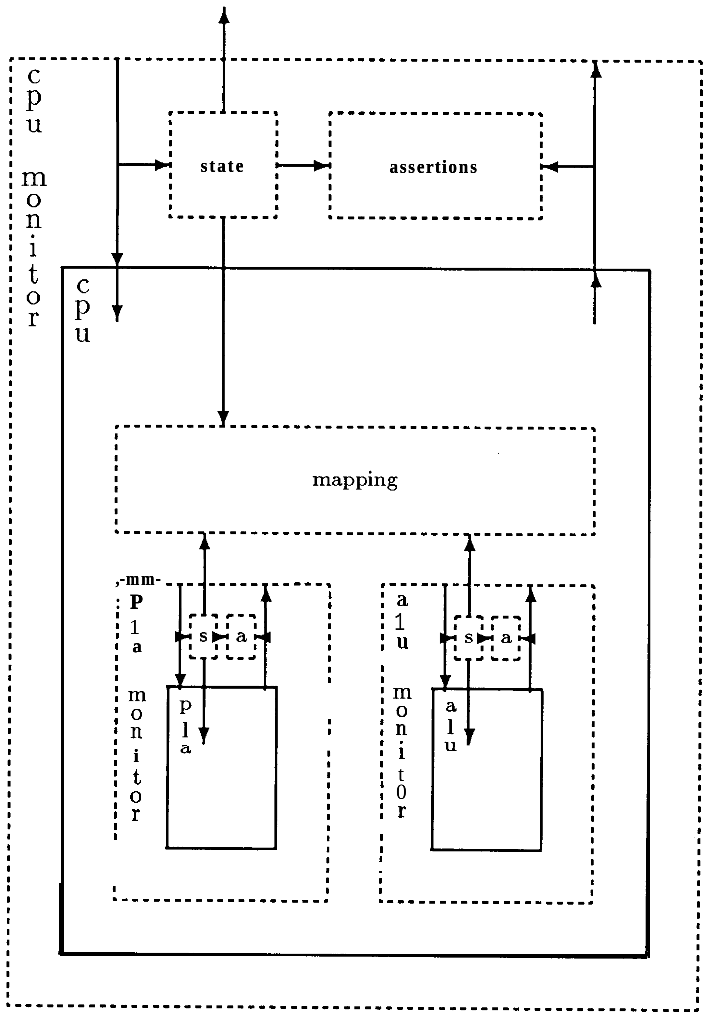

Figure 2.2: Mapping

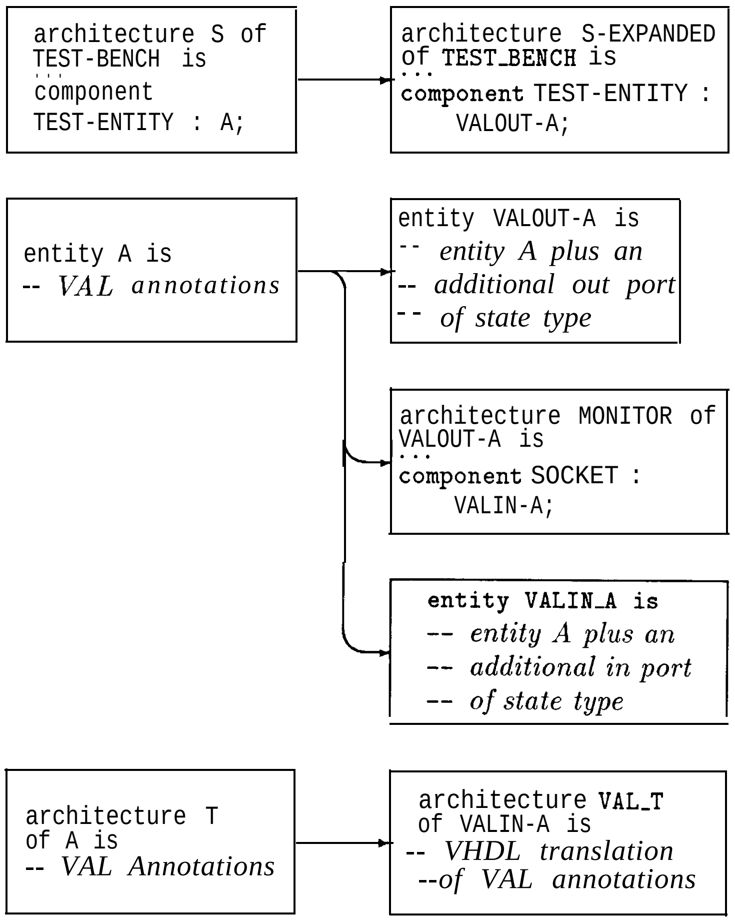

Figure 2.3: Relationship Between Design Units

2.2 Transforming VAL Annotations to Core VAL Annotations

Once the compilation units to be generated are produced, as described in section 2.1, the monitors and the expanded architectures must be filled with code corresponding to the VAL annotations. The first step is to transform the annotations to Core VAL annotations.

The core VAL annotations are equivalent to the original ones, but use fewer VAL constructs and are easier to map to VHDL.

The transformation steps required to produce the Core VAL Annotations are:

1. Normalization of time qualified expressions.
2. Isolation of base statements.
3. Elimination of derived syntax.
4. Flattening of nested guards.
5. Rescaling of timed expressions.

The following sections describe each of these steps in more detail.

2.2.1 Normalization of time qualified expressions

The normalization of time qualified expressions consists of three steps :

1. Generalize defaults – Default time references are added to every expression. There are two cases:

   1. el **during** T1 becomes el **during**[–T1,0]

   2. e1 becomes e1[0]

2. Set upper bounds – The upper bound of all time qualified expressions is shifted to zero. The expression

   ```
   S during[T1 , T2]
   ```

   becomes,

   ```
   S[T2] during[T1 –T2,0]
   ```

3. Push time references – Time references are "pushed" through each expression until they are associated with signals. A timed expression (e1[T1])[T2] is rewritten as (e1[T1 + T2]).

   The interval in time qualified expressions is not affected by pushing time references. For example,

```
(el [T1] during[T2,0])[T2]
```

becomes,

```
e1 [T1 + T2] during[T2, 0]
```

This eliminates all expressions of the form (e)[T]. Timed references are now only associated with signals.

2.2.2 Isolation of Base Statements

The base statements can be either assertions or drive statements. Each statement or group of statements may be guarded by hierarchical guard statements that contain general boolean expressions (including time qualified boolean expressions). For the purpose of transforming time references, each VAL assertion or drive statement is processed separately.

Each base statement is treated as if it had its own copy of the guards guarding it. Due to the transformation algorithm that is explained below, the same guard may be transformed differently for different guarded statements.

For example,

```
when e1 then s1; s2; end when;
```

becomes,

```
when e1 then s1; end when;
when e1 then s2; end when;
```

Where sl and s2 may be drive, assertion, or guarded statements.

2.2.3 Elimination of Derived Syntax

Derived syntax refers to language constructs that can be rewritten in terms of syntactically simpler language constructs. The rules that specify how to eliminate a language construct are known as *rewrite* rules since they specify how to rewrite one construct in terms of another.

The following VAL constructs are eliminated by this step:

- Select – The **select** process activates one of a set of child processes based on the value of a selection expression. It can be re-written as a set of **when** processes. For example,

```
select < expr >
  c1 | c2 => s1;
  c3 | c4 => s2;
  else s3;
end select;
```

becomes ,[4]

```
when (c1 = <expr>) or (c2 = <expr>) then s1; end when;
when (c3 = <expr>) or (c4 = <expr>) then s2; end when;
when (c1 /= <expr>) and (c2 /= <expr>) and
    (c3 /= <expr>) and (c4 /= <expr>) then s3; end when;
```

- Macro – A macro is a name for a list of parameterized statements. For every occurrence of the macro, the statements associated with the macro are copied, and the actual parameters substituted for the formal parameters. the rewrite rules must be performed recursively on the result of the expansion.
- Generate – The generate statement receives a label, if it has not one already (it is optional in VAL, but mandatory in VHDL). The generate statement is then translated as a VHDL generate statement.
- Else and elsewhen – The syntactically more complex forms of the guarded processes are rewritten into simpler forms. For example,

```
when e1 then s1;
  elsewhen e2 then s3;
  else s4;
end when;
```

becomes,

```
when e1 then s1; end when;
when not e1 and e2 then s3; end when;
when not e1 and not e2 then s4 end when;
```

[4]This, and the other rewrite rules, neglect semantic checking. If compile time semantic checks can guarantee no semantic errors, then the behavior of the rewritten expression is correct. Otherwise the transformation rule can be extended to include run-time semantic checking.

Once the rewrite rules have been performed recursively on the VAL description, only simple **when** processes (with no **else** parts), drive processes, and assertion processes remain.

2.2.4 Flattening of Nested Guards

Next, nested guarded statements are flattened. The VAL process

```
when e 1 then
  when e2 then
    s1;
  end when;
end when;
```

becomes,

```
when e1 and e2 then
  s l ;
end when;
```

The VAL description now consists of a list of simple (no else clauses) guarded statements, each guarding a single drive or assertion process.

2.2.5 Rescaling of Timed Expressions

Since time in VAL is relative, the reference point of the entire description can be shifted in time such that all references are to the past. This facilitates the translation to VHDL since only the past value of a signal can be referenced in VHDL.

The rules for manipulating VAL expressions are described theoretically in [1].

The rescaling of timed expressions consist of two steps:

1. Compute furthest future reference – For each guarded process, the time of the furthest forward reference of all expressions in that process is computed. This includes expressions in the drive or assertion process in the then part of the guarded process. This time point will be referred to as T_{max}. For an expression e, MAX(e) = T_{max} is:

   - MAX(e) = maximum of all the subexpressions of e
   - MAX(e during[T1,0]) = MAX(e)
   - MAX(s[T]) = T
   - MAX(T1,T2,T3,...) $= T_i$ if $T_i \geq T_j \ \forall j \neq i$

   T_{max} for a guarded expression is the MAX of all the expressions within the guarded statement and all of its children. A different T_{max} is computed for each guarded expression.

2. Rescale time – Each timed expression, with the exception of time in qualified expressions, has T_{max} subtracted from it. All time references should now be less than or equal to zero.

   Qualified expressions are not affected since they are normalized already.

Note: from the standpoint of preserving semantics, substracting a value from all time references in a condition that must hold over an interval is equivalent to adding the same value to the bounds of the interval and to leave the time references in the condition as they are.

For example, consider:

```
when e1 [T1] during [T2, T3] then
  s [T4] <- e2[T5];
end when;
```

The upper bound of the **during** is first shifted to zero, during the normalization of time qualified expressions..

```
when e1 [T1-T3] during[T2-T3,0] then
  s [T4] <- e2 [T5] ;
end when;
```

The furthest future reference time is given by:

$$TMAX = MAX(T4, T5, T1 - T3)$$

The rescaled code fragment becomes:

```
when e1[T1-T3-TMAX] during[T2-T3,0] then
  s [T4-TMAX] <- e2 [T5-TMAX] ;
end when;
```

Note that causality requires that $T4 - TMAX = 0$. If $TMAX > T4$, then the drive statement depends on a future value ($T5 > T4$ from the definition of TMAX) and the behavior is non-causal. Accordingly, a run time check for this condition is generated in the form of an assertion.

2.3 Code Generation

Once the preceding transformations have been applied to the VAL description, the code is in a canonical form characterized by:

- Only simple guarded processes with no nesting or else clauses.
- References rescaled relative to zero.
- Upper bound of time qualified expressions set to zero.
- One statement per guarded process.

There are two kinds of processes that can appear within a guarded process: a drive process or a flavor of assertion. In addition to these two cases of processes, timed expressions and time qualified expressions must also be translated into the corresponding VHDL.

The transformation of the Core VAL Annotations into VHDL are preformed in four steps:

1. Translation of timed expressions.
2. Translation of time qualified expressions.
3. Translation of drive processes.
4. Translation of assertion processes.

The following sections describe the translation of each of these language constructs.

2.3.1 Translation of timed expressions

Recall that in an earlier translation step all time references were rescaled relative to the constant T_{max}. Therefore all timed expressions must be less than zero; i.e. all timed expressions are delays. This can be modeled in VHDL by a signal assignment statement using *transport* delay.[5]

An expression

```
e[T]
```

becomes

```
S <= transport e after T;
```

All occurrences of the expression e[T] are then replaced with the signal S. All expressions are rewritten recursively until all timed expressions are eliminated. Transport delay is used to assure that no preemption [4] occurs on the signal. In VAL, once an assignment to a signal is made, in cannot be "undone."

[5]The predefined VHDL attribute 'delayed() cannot be used for this because the argument of 'delayed() must be a globally static expression. (See $7.4 of [5].) Although the argument generated by the translation algorithm is a "run-time" constant, it is actually computed at elaboration time using functions defined in a VAL package. Therefore it does not meet the VHDL definition of globally static.

```
signal GBE2 : BOOLEAN;

block
  signal GBE1 , GBE1_delay : BOOLEAN;
  signal GBE1_stable : BOOLEAN := TRUE;
begin
  GBE1 <= e ;
  GBE1_delay <= transport GBE1 after T;
  process(GBE1)
  begin
    if GBE_1'event then
      GBE1_stable <= FALSE;
      GBE1_stable <= transport true after T;
    end if;
  end process;
  GBE2 <= GBE1_stable and GBE1;
end block;
```

Figure 2.4: Translation of Time Qualified Expression

2.3.2 Translation of time qualified expressions

Recall that in an earlier step each time qualified expression was shifted in time such that its upper bound was zero. This can be translated into VHDL as a check for stability over the most recent interval using a VHDL process.[6] The expression

```
e during [T,01
```

is replaced by the signal GBE2 which is defined in VHDL in Figure 4.2.

Whenever the expression changes value, the process is activated and sets a flag to false to indicate that the expression is not stable. The flag is reset if the process is not activated (the expression does not change value) for **T** time units. Whenever the value of the expression changes, the new signal GBE2 is set to true if the expression is true and has been stable and true for the last T time units.

2.3.3 Translation of drive processes

In VAL/VHDL, the drive process can only be used to change the value of the entity state. After the previous transformations, there may be several guarded processes containing a drive statement affecting the entity state or a component of the state. Only one of these, however, should be active at any point in time. Because VHDL requires that a signal may be the target of only a single concurrent signal assignment statement, all of the guarded processes that may influence the state

[6]As with 'delayed(), the predefined VHDL attribute 'stable0 cannot be used in the translation because the argument may not be a globally static expression as defined in VHDL. The argument may not be a globally static expression because the Transformer introduces function calls as part of the translation process. In effect, the transformer generates code to implement the 'stable0 attribute itself.

are brought together into a single VHDL process. This process is sensitive to all of the signals that may influence the state, and checks that only a single assignment to state is active at any point in time.

Consider for example the following VAL code:

```
when G1 then
  state <- El;
end when;
when G2 then
  state <- E2;
end when;
```

This is translated into the VHDL shown in Figure **4.3.**

```
VALSTATE: block
  S1 : state-type;
  s2 : state-type;
begin
  S1 <= El;
  S2 <= E2;
  process ( S1 ,S2 )
    variable count : integer;
  begin
    count := 0;
    if (Gl) then
      count := count + 1;
      state <= Sl;
    end if;
    if (G2) then
      count := count + 1;
      state <= S2;
    end if;
    assert count <= 1
      report "VAL Error: Multiple assignment to state";
  end process;
end block;
```

Figure 2.5: Example of State Maintenance Process

2.3.4 Translation of assertion processes

The guards that guard assertions are eliminated. This is a VAL to VAL transformation step, which for efficiency reasons has been implemented during the final translation to VHDL.

For example, the VAL code

```
when G1 then
  assert B1;
end when;
```

is translated to

```
assert B1 or not Gl;
```

There are four flavors of assertions in VAL: **assert, finally, sometime,** and **eventually.** Each of these assertions is translated into a VHDL process, the details of which depend on the particular flavor of assertion. Because the default severity level in VAL is **WARNING,** the translation must set the severity level of generated VHDL assertions.

Assert

The VAL **assert** process is translated directly into the VHDL **assert** statement.

Finally

The **finally** assertion is translated into a VHDL process that wakes up whenever a signal in the asserted expression changes. The process than sets itself to wake up at the first delta of the next time and checks the value of the assertion. The value of the asserted expression will be the value it held at the end of all of the deltas in the previous time point.

For the assertion

```
finally <test-expression>
  report <message-expression>
  severity <severity-expression>;
```

the corresponding VHDL process is given in Figure 4.4.

Sometime

The translation for the **sometime** assertion closely resembles that for **finally**. Whenever a signal in the test expression changes, a process wakes up and checks if the test expression is true. The process then sets itself to wake up on the first delta of the next simulated time. When it wakes up at the next simulated time, the process checks that the expression was true in at least one delta in the previous simulation cycle. The translation for **sometime** is given in Figure 4.5.

Eventually

The **eventually** assertion is similar to **finally,** except that once the test expression goes true it must remain true during all deltas in the remainder of the time point. the translation is thus very similar to that for **finally,** with the addition that the process must check that the test expression never makes the transition from false to true and back to false at the same time point.

The translation for **eventually** is given in Figure 4.6.

```
VAL_FINALLY 1 : block
  signal next-t ime   : BOOLEAN;
  signal assert-expr : BOOLEAN;
begin
  assert-expr <= <test-expression>;
  process ( assert-expr ,next_time )
    variable first : BOOLEAN := TRUE;
    variable oneb  : BOOLEAN := TRUE;
  begin
    if next-t ime ' event then
      assert oneb
        report <message-expression>
        severity  <severity-expression>;
      first := TRUE;
    end if;
    if assert-expr ' event  then
      if (assert-expr /= oneb) then
        oneb := assert-expr;
        if first then
          next-time <= not next-time after 1fs;
          first := FALSE;
        end if;
      end if;
    end if;
 end process;
end block VAL FINALLYI;
```

Figure 2.6: Translation of Finally assertion

```
VAL-SOMETIME : block
  signal next-time, assert-expr : boolean := FALSE;
begin
  assert-expr <= <test-expression>;
  sometimes-label : process ( assert_expr,next_time'transaction )
    variable initial-cycle : boolean := TRUE;
    variable oneb  : boolean := FALSE;
    variable first : boolean := TRUE;
  begin
    if initial-cycle then
      initial-cycle := FALSE;
      next-time <= not next-time after 1 fs;
      first := FALSE;
    end if;
    if next-time 'event then
      assert oneb
        report <message-expression>
        severity <severity-expression>;
      first := TRUE;
      oneb :=  FALSE;
    end if;
    if(assert_expr'event or not next-time 'event) then
      oneb := oneb or assert-expr;
      if (first and not assert-expr) then
        next-time <= not next-time after ifs;
        first := false;
      end if;
    end if;
 end process sometimes-label;
end block VAL-SOMETIME;
```

Figure 2.7: Translation of Sometime assertion

```
VAL-EVENTUALLY : block
  signal next-time, assert-expr : boolean := FALSE;
begin
  assert-expr <= <test-expression>;
  eventually-label : process ( assert_expr,next_time )
    variable glitch : boolean := FALSE;
    variable oneb   : boolean := FALSE;
    variable first  : boolean := TRUE;
  begin
    if (not assert-expr 'event and not next-time 'event) then
      next-time <= not next-time after Ifs;
      first := FALSE;
    end if;
    if next-time 'event then
      assert oneb
        report <test-message>
        severity <severity-expression>;
      first := TRUE;
      glitch := FALSE;
    end if;
    if (assert-expr 'event) then
      glitch := glitch or (oneb and not assert-expr);
      oneb := assert-expr;
      if (first and not oneb) then
        next-time <= not next-time after ifs;
        first := FALSE;
      end if;
    end if;
  end process eventually-label;
end block VAL-EVENTUALLY;
```

Figure 2.8: Translation of Eventually assertion

2.4 Transformation of Attributes

2.4.1 Changed

This boolean attribute may have a parameter, such as in 'CHANGED(val) or may not have one, such as in 'CHANGED. In the latter case a parameter "any" is assumed.

If X'CHANGED is TRUE, it means that X changed its value in the last VHDL simulation cycle. If X'CHANGED is FALSE, it means that X did not change its value since the last VHDL simulation cycle. Thus, X'CHANGED is transformed as presented in figure 2.9.

```
(NOT X'STABLE)
```

Figure 2.9: Transformation of X'CHANGED

If X'CHANGED(val) is TRUE, it means that X has changed its value to "val" in the last simulation cycle. If X'CHANGED(val) is FALSE, it means that X has not changed to "val" in the last VHDL simulation cycle. Thus, X'CHANGED(val) is transformed as presented in figure 2.10.

```
((NOT X'STABLE) AND (X = val))
```

Figure 2.10: Transformation of X'CHANGED(val)

2.5 Transformation of Architecture VAL-Annotations

A VAL-annotated VHDL architecture is translated in a VHDL architecture, called "expanded". The prefix "Val" is added to the original name of the architecture in order to produce a unique identifier. The expanded architecture belongs to the VALIN, entity that is produced by transforming the original entity. The VALIN, entity has a ValState port of mode in that provides the expanded architecture with visibility over the state of the Monitor (the ValEntityState within the Monitor).

The component declarations within the expanded architecture may contain, in addition to their original ports, a port called ValState of a type indicated by the associated VAL annotation. If such a VAL annotation is missing, the component declarations maintain their original ports.

The instances of components that have the additional ValState port, have this port connected to a signal named *InstanceName*_State. Also, all references to *InstanceName*. State are transformed into refences to the signal *InstanceName-St* ate.

In order to make the out ports visible to annotations, a signal is declared for each out port.

2.6 Transformation VAL Configuration Annotations

The VAL user can select the entities to be monitored, with the VAL constructs `--| ValEntity`, and `--| ValArchitecture`.

- The `--| Valentity` construct specifies that an entity is to be monitored using interface annotations.
- The -- | Valarchitecture construct specifies that an architecture is to be monitored using body annotations.

Both annotations appear inside a VHDL component configuration, right after the binding indication. Component configurations appear in either architecture body or configuration declaration. Currently the two constructs is supported only in configuration declarations.

For each entity E, with an architecture A, the Val transformer will produce two interfaces and four architectures :

1. entity `Valout_E` : entity E with an extra Valstate out port ;
2. architecture `StateMonitor` of `valout_E` : monitor with a socket for the Valin_E entity, state maintenance and assertion checking ;
3. architecture `DummyMonitor` of `Valout_E` : monitor with a socket for the Valin_E entity, but neither state maintenance nor assertion checking ;
4. entity `Valin_E` : entity E with an extra ValState in port ;
5. architecture `Val_A` of `Valin_E` : the architecture A of entity E, where all components have been added an extra Valstate out port (if they themselves have a state model), and where the Val annotations have been translated into VHDL statements.
6. architecture A of `Valin_E` : the architecture A of entity E, where all components have been added an extra Valstate out port, but without translation for Val annotations.

Assuming now a component configuration :

```
for . . . use entity E (A) ;
```

the VAL transformer will always add an extra level in the component hierarchy. Depending on the specified configuration annotations, the transformer will pick from the available architectures. If the -- | Valentity construct appears in the component configuration, the transformer will select the StateMonitor architecture. If not, it will select the DummyMonitor. If the -- | Valarchitecture construct appears in the component configuration, the transformer will select the Val_A architecture. If not, it will select the A architecture.

```
for ... use entity E(A) ;
    for A
        ...
    end for ;
end for
for ... use entity Valout_E (DummyMonitor) ;
    for DummyMonitor
        for actual : socket use entity Valin_E (A) ;
            for A
                ...
            end for ;
        end for ;
    end for ;
end for

for ... use entity E(A) ;
    --| Valentity ;
    for A
        ...
    end for ;
end for
for ... use entity Valout_E (StateMonitor) ;
    for StateMonitor
        for actual : socket use entity Valin_E (A) ;
            for A
                ...
            end for ;
        end for ;
    end for ;
end for

for ... use entity E(A) ;
    --| Valarchitecture ;
    for A
        ...
    end for ;
end for
for ... use entity Valout_E (DummyMonitor) ;
    for DummyMonitor
        for actual : socket use entity Valin_E (Val_A) ;
            for Val_A
                ...
            end for ;
        end for ;
    end for ;
end for

for ... use entity E(A) ;
    --  Valentity ;
    --  Valarchitecture ;
    for A
        ...
    end for ;
end for
for ... use entity Valout_E (StateMonitor) ;
    for StateMonitor
        for actual : socket use entity Valin_E (Val_A) ;
            for Val_A
                ...
            end for ;
        end for ;
    end for ;
end for
```

Figure 2.11 provides an example where the original VHDL architecture is to be used in conjunction with the Val annotated entity.

Figure 2.12 provides an example where the Val annotated architecture is to be used in conjunction with the Val annotated entity.

```
        for all: DFlipFlop use
          entity DFlipFlop (simple) ;
          --| ValEntity;
        end for;

    -- is transformed into

        for all: DFlipFlop use
          entity ValOut_DFlipFlop (StateMonitor);
          for StateMonitor
            for actual: Socket use
              entity Valin_DFlipFlop (Simple) ;
            end for ;
          end for ;
        end for;
```

Figure 2.11: Transformation of Configuration that uses “ValEntity”

```
        for all: DFlipFlop use
          entity DFlipFlop (Simple);
          --|ValEntity;
          --| ValArchitecture;
        end for;

    -- is transformed into

        for all: DFlipFlop use
          entity ValOut_DFlipFlop (StateMonitor);
          for StateMonitor
            for actual: Socket use
              entity ValIn_DFlipFlop (Val_Simple) ;
            end for;
          end for ;
        end for;
```

Figure 2.12: Transformation of Configuration that uses both “ValEntity” and “ValArchitecture”

2.7 Summary

The VAL entity annotation is transformed into a Monitor that watches a socket in which an architecture of the annotated entity can be plugged. Both the socket and the Monitor are hosted in a generated VHDL architecture belonging to an VALIN_ entity.

Whenever such an entity is used, the configuration specifies which architecture to use. In turn the configuration of the generated architecture specifies some other architecture for the Actual "chip" to be plugged in the socket.

This chapter presented the transformation of various VAL specific constructs into VHDL, in order to implement the monitoring activity.

Chapter 3

Utility Packages

3.1 Abstract Syntax Tree

Each VAL-VHDL file is transformed into a tree *(Abstract Syntax* Tree), before the VAL statements are transformed. The tree is inspired from the DIANA tree designed for ADA [3]. The tree is including a symbol table, described in section 3.3. The internal tree format allows for fast access of all information required at compile time. The tree is language independent. It is customized for each language represents, by defining constants for node names and attributes.

For language customization, the tree uses constants generated by the parser generator for tree node names, for terminal names and grammar rule names. It also relies on user defined constants that, describe structural and semantic attributes.

3.2 Parser Generator

The PGEN parser generator [2] takes as input a LALR1 description of the VHDL grammar. It produces parse tables and a list of constants (tree node names, terminal names and grammar rule names). Parse tables are used by the parser to produce from a VHDL file an abstract syntax tree, on which the translation will be performed. The constants are used in pretty much every operation performed on the tree.

3.3 Symbol Table

All identifiers encountered in a VAL-annotated VHDL description are stored in a symbol table, along with any relevant semantic information such as type information.

The symbol table information is important in order to ensure the uniqueness of the generated identifiers, as well as to perform some transformations based on the nature of the referenced object. For example, all right hand side references to out ports in a generated Monitor or StateMonitor are transformed into references to local signals containing the contribution of the original architecture to the value of the signal connected to the given out port.

The symbol table is necessary for the following operations:

1. Determining the uniqueness of generated identifiers.

2. Determining if an identifier is an out mode port.

3. Finding the type of generated intermediate signals.

4. Finding the body of a macro call.

The symbol table does not need to perform overload resolution since the transformer never needs to uniquely determine the actual function called.

Chapter 4

Transformer Structure

4.1 Overview

The transformer accepts as input an abstract syntax tree that corresponds to the annotated program. This tree is created by the parser (automatically generated by PGEN). Accessing the information in the tree is done using the abstract syntax tree (AST) package.

4.2 Transformer

The transformer performs tree to tree transformations on nodes of the tree that correspond to VAL constructs. The subtree corresponding to each VAL construct is eventually replaced by a subtree that consists of VHDL constructs only and represents the translated version of the VAL construct. Sometimes the translation process requires the declaration of new signals/variables, the VHDL subtrees corresponding to these are created and inserted in the right places. The transformed tree is then passed to the pretty printer which traverses the tree and creates an ascii file that contains the transformed version of the annotated program.

The transformer thus uses the AST package for tree to tree transformations and the pretty printer for printing the transformed program. In addition, it also uses the symbol table package to find out attributes of identifiers and to create unique names.

The transformation process is broken down into 6 stages that are executed one after the other. Each stage of the transformation is implemented as an ADA separate procedure. The procedures and their corresponding files are listed below. They are described in the next section. Each package exports a single procedure which we call the main procedure.

The top level package is called TRANSFORMER and it exists in the files sxform_v.a and sxform_b.a. A file ending in _v.a contains the package interface whereas a file ending in _b.a contains the package body. The procedures call sequence of a typical transformer run is given below. It correspond to the file dependency graph of the transformer program. Each line of the figure is a procedure call, and the file in which the procedure is written. Each | . . . sequence indicates the nesting level of each file in the file dependency graph.

```
xform "xform/xform.a"
  . .parser.init_tables "parser/parser_b.a"
```

```
|...parser.parse "parser/parserp_b.a"
|...transformer.transform "xform/sxform_b.a"
|...|...transformer.expand "xform/expand.tex"
|...|...make-skeleton "xform/skeleton.a"
|...|...push_time_in "xform/push.tex"
|...|...eliminate-derived-syntax "xform/eliminate.a"
|...|...rescale,timed-expression "xform/rescale.a"
|...|...flatten_when_statements "xform/flatten.a"
|...|...|...xform,assert "xform/xform_assert.a"
|...|...|...xform_eventually "xform/xform_eventually.a"
|...|...|...xform_finally "xform/xform,finally.a"
|...|...|...xform,sometimes "xform/xform_sometimes.a"
|...|...remove-annotations "xform/sxform_b.a"
|...|...print_tree "pp/exter_b.a"
|...pretty-printer.print "pp/prtty_b.a"
|...make.command "xform/make_b.a"
```

In addition there is another package called SUBSTITUTE located in files substitute-v.a and substitute_b.a. The main procedure is called REPLACE. This package is used by the package SKELETONMAKER.

The file called xform.a contains calls to the parser, transformer and pretty printer in that order. The user thus provides a text file containing an annotated program and gets back a text file containing the transformed version.

4.3 The AST and the symbol-table package

The following figure shows package dependencies in the ast package. The type definitions for the AST and the symbol table are found in package ast/val-defs-b.a. The two types are in the same package since they are mutually dependent. The procedural interface to the symbol table package is in ast/val_symbol_table_v.a and ast/val_symbol_table_b.a. The procedural interface to the AST package is in val_ast_v.a and val_ast_b.a. A |... indicates the package or procedure is made visible using an ADA with clause. A !... indicates the package or procedure is separate.

```
val,symbol-table "ast/val_symbol_table_v.a"
|...val_defs "ast/val_defs-v.a"

val,symbol-table "ast/val_symbol_table_b.a"
|...errorp "ast/errorp_v.a"
|...symtab_io "ast/symtab_io_v.a"
|...namer "ast/namer-b.a"
|...utilities "ast/util_v.a"
|...val_ast "ast/val_ast_v.a"
|...parse_table_defs "ast/uconsts.a"
|...val,attrs "ast/val_attrs_v.a"
```

```
|...val-defs "ast/val_defs_v.a"
|...create-standard-environment "ast/create_standard_environment.a"
|...get-scope-defined-by "ast/get_scope_defined_by.a"
|...insert-context "ast/context.a"
|...get-child-scope "ast/get-child-scope.a"
|...walk-tree "ast/build.a"
|...get_type "ast/get_type.a"

val,ast "ast/val_ast_v.a"
|...sets "ast/sets_v.a"
|...parse-table-defs "ast/uconsts.a"
|...val_attrs "ast/val_attrs_v.a"
|...val-defs "ast/val_defs_v.a"

val_ast "ast/val_ast_b.a"
|...new-unchecked-deallocation
|...text_io
|...hash-table "ast/hash_v.a"
|...val,symbol-table "ast/val_symbol_table_v.a"
|...diana-mappings "ast/val_ast_b.a"
|...diana-mappings "ast/mapb1_b.a"

val,defs "ast/val_defs-v.a"
|...symbol-table "ast/symb_v.a"
|...val,attrs "ast/val_attrs_v.a"
|...parse,table-defs "ast/uconsts.a"
```

Bibliography

[1] L. M. Augustin. *An Algebra of Waveforms.* Technical Report , Computer Systems Laboratory, Stanford University, 1989. Submitted to the IFIP International Workshop on Applied Formal Methods For Correct VLSI Design, Leuven, Belgium, Nov. 1989.

[2] Rob Chang. *ParseGen: A LALR(1) Parser Generator.* Technical Note 85-283, Stanford University, November 1985.

[3] G. Goos, W. A. Wulf, A. Evans Jr., and K. J. Butler. *DIANA, An Intermediate Language for Ada.* Volume 161, Springer-Verlag, 1983.

[4] D. C. Luckham, A. Stanculescu, Y. Huh, and S.Ghosh. The semantics of timing constructs in hardware description languages. In *IEEE International Conference on Computer Design: VLSI in Computers (ICCD* 'SF), pages 10-14, Port Chester, New York, October 1986. Also published as Stanford Univerity Computer Systems Laboratory Technical Report CSL-TR-86-303.

[5] *IEEE Standard VHDL Language Reference Manual.* IEEE, Inc., 345 East 47th Street, New York, NY, 10017, March 1987. IEEE Standard 1076-1987.