Theorem 3.3.2.1:

$\forall dsgn.\ (\text{well_formed_design_sigs}\ dsgn) \supset (\text{get_ports}\ dsgn \subseteq \text{get_sigs_design}\ dsgn)$

The proof was performed in the theorem prover by induction on the structure of designs. This proof, because of its mechanical nature, is not given here; but the argument is fairly straightforward. The interested reader is referred to Appendix A for the actual proof script. The base case follows from an expansion of the definition of well_formed_design_sigs. Once done, one observes that the ports and declarations of the component must be the same as the signals of its concurrent statements (derived by get_sigs_cs). This of course means that the ports are a subset of the total signals. The induction step is also a based on a straightforward expansion of definitions. Once those of well_formed_design_sigs, get_ports and get_sigs_design have been applied to composite designs one arrives at an instance of the induction hypothesis.

3.3.3 Overall Well-Formedness

It is now possible to make an overall definition about the syntactic well-formedness of any Femto-VHDL design. Simply stated, the signals of the design must exhibit the appropriate relationships, and the concurrent statements must not contain multiply driven signals. These requirements are formally defined in the function well_formed_design:

well_formed_design $design \equiv$
$(\text{well_formed_design_sigs}\ design) \wedge (\text{wfF_csss}\ (\text{dflat}\ design))$

A simple fact arising from the definition of well_formed_design is:

Theorem 3.3.3.1:

$\forall design.\ \text{well_formed_design}\ design \supset \text{well_formed_design_sigs}\ design$

The intuition for the proof is to assume well_formed_design $design$, expand the definition and take the first conjunct.

Well-Formedness of Design Signals

In order to ensure the well-formedness of the signals in a Femto-VHDL program, the various constraints on signals mentioned earlier must be met. To reiterate, they were:

- Only declared signals may be used in the program text.
- Directionality of assignments must be preserved.

The function well_formed_design_sigs uses the just defined functions to express these goals for the well-formedness of the signals in a Femto-VHDL design.

well_formed_design_sigs $design \equiv$
  if $design =$ arch $ent\ arch\ ports\ decls\ cs$ then
    let $ip =$ fst $ports$ and $op =$ fst (snd $ports$) and $op =$ snd (snd $ports$)
    and $(isigcs, osigcs) =$ get_sigs_cs cs and $d =$ get_sigs_decls $decls$ in
    let $p = (ip \cup op \cup iop)$ and $sigcs = (isigcs \cup osigcs)$ in
    let $pId = (p \cap d = \{\})$ and $oIi = (op \cap isigcs = \{\})$ and $iIo = (ip \cap osigcs = \{\})$ in
      $(sigcs = p \cup d) \wedge pId \wedge oIi \wedge iIo$
  else
    let (dpar $d'\ d''$) $= design$ in
    let $(P', S') =$ well_formed_design_sigs d'
    and $(P'', S'') =$ well_formed_design_sigs d'' in
      $(P' \wedge P'')$

In the case of stand-alone components, *ip* are the `in` ports, *op* are the `out` ports and *iop* are the `inout` ports. *isigs* represents the signals that are used as inputs in the concurrent statements of the component, whilst *osigs* are those used as outputs. *d* are the locally declared signals. The variable *p* holds all the ports, and *sigcs* contains all the signals of the concurrent statements. *pId* states that the ports and locally declared signals must be disjoint, *oIi* says that the output ports must not be used as inputs, and *iIo* ensures that input ports are not being used as outputs. Conjoining these last three with the statement that the ports of the component are the same as the signals used by the concurrent statements ensures that the all the signals of that component are well-formed.

A useful consequence of the definition of well_formed_design_sigs when reasoning about the dynamic semantics of Femto-VHDL in Chapter 4, will be the knowledge that if the signals in a design display the proper interrelationships, then the ports of the design are a subset of all the signals in that design.

Finally, it is necessary to traverse an entire sea of concurrent processes gathering the same information. The function get_sigs_cs does this, and returns a pair of sets of the same form as those created by get_sigs_ss.

get_sigs_cs $cs \equiv$
  if $cs = (sl : ss)$ then get_sigs_ss ss
  else let $(cs' \,||\, cs'') = cs$ in
    let $(is', os') =$ get_sigs_cs cs' and $(is'', os'') =$ get_sigs_cs cs'' in
      $(is' \cup is'', os' \cup os'')$

As an example of the behaviour of this suite of functions, take the Femto-VHDL processes:

$$\{x\} : a \overset{\text{inert}}{:=} (\text{not } x, 2) \,||\, \{y'\} : b \overset{\text{trans}}{:=} (y' \text{ and } y'', 0) \,||\, \{z\} : c \overset{\text{trans}}{:=} (z, 0)$$

When get_sigs_cs is run over them, the pair $(\{a,b,c\},\{x,y',y'',z\})$ is returned.

A companion function to get_sigs_cs is get_sigs_design. It extracts all the signal names from a design without regard for their associated direction by making use of get_sigs_cs.

get_sigs_design $design \equiv$
  if $design =$ arch $ent\ arch\ ports\ decls\ cs$ then
    let $(is, os) =$ get_sigs_cs cs in $is \cup os$
  else
    let (dpar $d'\ d''$) $= design$ in (get_sigs_design d') $\cup$ (get_sigs_design d'')

In a similar fashion to the previous example, the architectures

$$\text{dpar}\left(\begin{array}{l}\text{arch } e'\ a'\ p'\ d'\left(\{x\} : a \overset{\text{inert}}{:=} (\text{not } x, 2)\right) \\ \text{dpar}\left(\begin{array}{l}\text{arch } e''\ a''\ p''\ d''\left(\{y'\} : b \overset{\text{trans}}{:=} (y' \text{ and } y'', 0)\right) \\ \text{arch } e'''\ a'''\ p'''\ d'''\left(\{z\} : c \overset{\text{trans}}{:=} (z, 0)\right)\end{array}\right)\end{array}\right)$$

would yield the single set $\{a,b,c,x,y',y'',z\}$ when put through get_sigs_design.

get_ports $design \equiv$
  if $design =$ arch $ent\ arch\ ports\ decls\ cs$ then
    $(\text{fst}\ ports) \cup (\text{fst}\ (\text{snd}\ ports)) \cup (\text{snd}\ (\text{snd}\ ports))$
  else let $(\text{dpar}\ d'\ d'') = design$ in $(\text{get_ports}\ d') \cup (\text{get_ports}\ d'')$

The whole idea of the function is to remove any reference to directionality, and return as a set only the names of the ports of a particular design.

The above functions extract the declared signals in a design. However, in order to make the kind of checks that are necessary, functions that return the signals that are actually used in the concurrent statements of a design are also required. A series of functions that extract these signals, paying particular attention to their intended direction, are now defined.

At the lowest level, the signals will need to be returned from Boolean expressions. Because these expressions are only capable of holding input signals (i.e. they are always read, but never written to), all that is required in get_sigs_bool is to gather together any signal names found in a Boolean expression, and return them as a set.

get_sigs_bool $bexp \equiv$
  if $(bexp = \text{not}\ e)$ then $(\text{get_sigs_bool}\ e)$
  elseif $(bexp = e'\ \text{and}\ e'') \vee (bexp = e'\ \text{or}\ e'') \vee (bexp = e'\ \text{nand}\ e'') \vee$
      $(bexp = e'\ \text{nor}\ e'') \vee (bexp = e'\ \text{xor}\ e'') \vee (bexp = e'\ \text{xnor}\ e'')$ then
    $(\text{get_sigs_bool}\ e') \cup (\text{get_sigs_bool}\ e'')$
  elseif $(bexp = sig) \vee (bexp = sig\text{'delayed}(t)) \vee (bexp = sig\text{'event})$ then $\{sig\}$
  else $\{\}$

Sequential statements contain both input and output signals. The only output signals are those on the left-hand side of signal assignment statements. All other signals are assumed to be inputs. The function get_sigs_ss returns a pair of sets, the first of which is the set of input signals (including those found in any Boolean expressions), while the second is the set of output signals.

get_sigs_ss $ss \equiv$
  if $(ss = ss'\ \textbf{;}\ ss'') \vee (ss = e\ \textbf{=>}\ ss'\ \textbf{|}\ ss'')$ then
    let $(is', os') = \text{get_sigs_ss}\ ss'$ and $(is'', os'') = \text{get_sigs_ss}\ ss''$
    and $bools =$ if $(ss = e\ \textbf{=>}\ ss'\ \textbf{|}\ ss'')$ then get_sigs_bool e else $\{\}$ in
      $(is' \cup is'' \cup bools, os' \cup os'')$
  elseif $\left(ss = s\ \overset{\text{inert}}{\textbf{:=}}\ (e, dly)\right) \vee \left(ss = s\ \overset{\text{trans}}{\textbf{:=}}\ (e, dly)\right)$ then $(\text{get_sigs_bool}\ e, \{s\})$
  else $(\{\}, \{\})$

If the just-used concurrent statements were passed as arguments to wfF_csss, it is clear from the above results that an invocation on the processes which assign to both a and b would fulfil the multiple driver restriction, while one where both processes address the signal a would not.

In order to make use of wfF_csss for larger components, it will be necessary to flatten out architectural hierarchies into a sea of processes. To this end, the function dflat has been defined. The function takes one or more Femto-VHDL components, and returns only the processes associated with each one. In the case of multiple architectures, the processes of each are glued together in parallel.

dflat *design* ≡
  if *design* = arch *ent arch ports decls cs* then *cs*
  else let (dpar d' d'') = *design* in (dflat d') || (dflat d'')

For instance, the design dpar (arch e' a' p' d' cs') (arch e'' a'' p'' d'' cs'') would, when passed through dflat, be reduced to $cs' \,||\, cs''$.

3.3.2 Signal Relationships

The other restrictions on the syntactic well-formedness of Femto-VHDL programs hinge heavily on the interrelationships of the signals as they are found in individual components. In order to make any statement about those relationships, it will first be necessary to extract the necessary signals from a program text. Having done so, it is then possible to go on to make a definition of what is required of those signals in the overall design.

Signal Extraction

When dealing with Femto-VHDL designs, two kinds of signals are of interest. One group is the external ports. The other is the locally declared signals. The function get_sigs_decls accesses the set of locally declared signals in a stand-alone component.

get_sigs_decls (signal *sigs*) ≡ *sigs*

For instance, if the function were applied to the declarations of arch e a p (signal d) cs, it would return the set d. The ports of a design may be extracted with get_ports. The function is defined recursively over the structure of designs in an analogous manner to dflat.

set for any other process in the design, the program text is not well-formed. At the lowest level of this check, one needs to extract a set of signal names that are assigned a value in the statements of a given process. Having done so, the requirement is then to make sure that these signals are not being assigned to in any other process.

As the first step in the check, one defines the function well_formed_Femto_ss over the structure of sequential statements:

well_formed_Femto_ss $ss \equiv$
if $(ss = ss'\ \mathbf{;}\ ss'') \vee (ss = e\ \mathbf{=>}\ ss'\ \mathbf{|}\ ss'')$ then
$(\text{well_formed_Femto_ss}\ ss') \cup (\text{well_formed_Femto_ss}\ ss'')$
elseif $\left(ss = s\ \overset{\text{inert}}{\mathbf{:=}}\ (e, dly)\right) \vee \left(ss = s\ \overset{\text{trans}}{\mathbf{:=}}\ (e, dly)\right)$ then $\{s\}$
else $\{\}$

The function returns, as a set, the signal names that appear on the left-hand side of any assignment statement. For instance, well_formed_Femto_ss would return the set $\{a, b\}$ if given the conditional statement $e\ \mathbf{=>}\ a\ \overset{\text{inert}}{\mathbf{:=}}\ (x, 2)\ \mathbf{|}\ b\ \overset{\text{trans}}{\mathbf{:=}}\ (y, 0)$.

Use may now be made of well_formed_Femto_ss in gathering together a collection of assigned-to signals for all the processes in a given architecture. The function well_formed_Femto_cs, again defined recursively on structure, does just that. It returns a tuple of sets. The first element of the pair represents the intersection that is of interest, while the second is the set of all assigned-to signals.

well_formed_Femto_cs $cs \equiv$
if $cs = (sl\ \mathbf{:}\ ss)$ then $(\{\}, \text{well_formed_Femto_ss}\ ss)$
else let $(cs'\ ||\ cs'') = cs$ in
let $a = \text{snd}(\text{well_formed_Femto_cs}\ cs')$
and $b = \text{snd}(\text{well_formed_Femto_cs}\ cs'')$ in
$(a \cap b, a \cup b)$

If the two processes $\{x\}\ \mathbf{:}\ a\ \overset{\text{inert}}{\mathbf{:=}}\ (x, 2)\ ||\ \{y\}\ \mathbf{:}\ b\ \overset{\text{trans}}{\mathbf{:=}}\ (y, 0)$ were used as an example, the function would return the pair $(\{\}, \{a, b\})$. However, had both signal assignments been addressing signal a, the resulting pair would have been $(\{a\}, \{a\})$.

Having defined the two previous functions, all that remains to be done is ascertain the emptiness of the first element of the pair returned by well_formed_Femto_cs. The function wfF_csss is defined as a top-level interface, and makes the desired comparison:

wfF_csss $cs \equiv (\text{fst}(\text{well_formed_Femto_cs}\ cs) = \{\})$

```
foo x ≡
  let a = x + 1 and b = x − 1 in
  let c = a + b in
      (x, c)
```

The function takes a natural number as its argument and returns a pair of naturals. The nesting of let-expressions allows for the calculation of c. Scoping is such that a and b are not visible to one another, but are visible in the second let-expression when c is formed. If one expanded the first let-expression, the result would be:

```
foo x ≡
  let c = (x + 1) + (x − 1) in
      (x, c)
```

Use will also be made of the form if x then … (elseif y then …) else … to express conditionals. Lists will be required in some of the definitions that come later, and list processing functions such as hd (head), tl (tail) and . (cons) will therefore be needed as well. Finally, it will be necessary to make use of tuples in some definitions, and the functions fst and snd will be used to respectively access the first and second elements of them.

3.3 Syntactic Well-Formedness

What constitutes a syntactically valid Femto-VHDL program? First, output signals must not have multiple drivers. This means that a signal must not be assigned to from more than one source. Next, the signals used in the program text must be only those that happen to be declared as ports or as local signal declarations. Finally, those signals must also be assigned to in an appropriate manner (i.e. signals of direction `in` must not be driven, and signals of direction `out` must not be read). The rest of the chapter involves a series of definitions leading up to the function well_formed_design which encompasses these properties. Readers not interested in the details of these definitions may safely progress to Chapter 4.

3.3.1 Multiple Drivers

To check the multiple driver condition, it is necessary to determine that a given signal name appears on the left-hand side of the signal assignments of only one process in the program text. In other words, if the set of assigned-to signal names for one process of the design has a non-empty intersection with a similarly derived

definition of *design*, represent the name of the entity and the name of the architecture respectively. It is therefore entirely possible for individual components to have the same entity name, but different architecture names.

The following tables give the relationship between the abstract syntax and the concrete syntax for various syntactic classes of Femto-VHDL. No mapping will be shown for Boolean expressions, as the abstract and concrete syntaxes are the same. For reasons of brevity the abstract syntax will be used in describing the semantics of Femto-VHDL.

design	

arch $string_e$ $string_a$ *ports decls cs*	`entity` $string_e$ `is`
	`port` *ports*`;`
	`end` $string_e$`;`
	`architecture` $string_a$ `of` $string_e$ `is`
	`signal` $s_1,\ldots,s_n$`;`
	`begin` *cs* `end` $string_a$`;`

cs	

$\{s_1,\ldots,s_n\}$: *ss*	`process` $(s_1,\ldots,s_n)$
	`begin`
	ss
	`end process;`

ss	
$signal \overset{\text{inert}}{:=} (bexp, n)$	*signal* `<=` *bexp* `after` *n*`;`
$signal \overset{\text{trans}}{:=} (bexp, n)$	*signal* `<= transport` *bexp* `after` *n*`;`
null	`null;`
bexp => *ss* | *ss*	`if` *bexp* `then` *ss* `else` *ss* `end if;`

3.2 Definitional Style

When defining functions and relations associated with the semantics, recourse will be made to the following definitional style. The symbol ≡ indicates a definition. The function name, when first defined, will appear in **bold sans serif** typeface. Variables will be shown in *math italic*. Internal functions and variables will be defined using let … (and …) in constructions. As an example, the function foo may defined as:

The symbol *bexp* ranges over Femto-VHDL Boolean expressions. These are made up of single signal names, attributed signals ('event and 'delayed), or compound expressions using the various VHDL Boolean operators. Attributes allow the user to access properties of signals. The ones in Femto-VHDL are used for ascertaining whether or not a signal has an event on it now ('event) and for looking at the value of a signal delayed by n units of time expressed as a natural number ('delayed). Sequential statements (*ss*) may be either two statements in sequence (;), a null statement, a conditional statement guarded by some Boolean expression, an inertial delay signal assignment statement, or a transport delay signal assignment statement. Each type of signal assignment statement maps a Boolean value to a signal after some delay also expressed in terms of the naturals. Concurrent statements (*cs*) are either two such statements in parallel, or a single process statement made up of a sensitivity list and some sequential statements. Sensitivity lists (*sl*) are simply sets of signal names.

The overall structure of design hierarchies is expressed as:

design	::=	arch $string_e$ $string_a$ *ports decls cs* | dpar *design design*
decls	::=	signal $\{s_1, \ldots, s_n\}$

Local signal declarations may be made (*decls*), and do not have associated directions. Ports (*ports*) are the external interfaces for VHDL architectures. They have associated directions, and are represented in Femto-VHDL, by a triple of sets. Signals of direction `in` (inputs) are in the first part of the triple, `out` signals (outputs) are in the second part and `inout` signals (bi-directionals) are in the final part. A design is composed of either a stand-alone component or a nested hierarchy of them. It should be noted that neither || nor dpar have any corresponding construct in the concrete syntax of VHDL. They are given here simply as a means of grouping together language constructs.

VHDL components may be specified at different levels of abstraction. To this end, the concept of design *entity* and design *architecture* are available to the designer. An entity is nothing more than a specification of connections to the outside world via ports. For purposes of Femto-VHDL, an architecture groups together concurrent statements that express the behaviour of a particular component (e.g. an ALU). Two particular architectures may actually represent the same component at different levels of abstraction. In that case the architectures are really part of the same entity, as they make use of the same ports to plug into the outside world irrespective of their internal construction. The two strings $string_e$ and $string_a$ in the arch part of the

activation. The sequential statements addressed are the `if-then-else` conditional, as well as inertial and transport delay signal assignments. Architectural hierarchies allow the user to group concurrent statements into large blocks with specific inputs and outputs. When simulated, these hierarchies will be flattened out into a sea of concurrent processes. For purposes of the subset, the architectures here are limited to those without VHDL `generic` [21] statements. Furthermore, the following assumptions are made:

- Signals may only be Boolean-valued. This is for simplicity, and does not materially affect the exposition of the simulation model.
- Resolved (multiply driven) signals are not allowed. The implication is that constructs such as wired-and's and wired-or's are not permitted.

Why these particular assumptions? The first one is understandable in that assignment to Boolean-valued signals is the same as assignment to signals of any other value. The focus here is on the act of assignment, not on the type of the assigned-to signal. The second assumption is a bit more cryptic. In essence, it ensures that no signal may be assigned to from more than one `process` statement. If this was allowed, two active processes might try to assign to the same signal at the same time causing a clash over which value that signal should take on. Full VHDL provides a way of dealing with this eventuality through the use of user-defined resolution functions [21]. In an effort to keep the discussion of the simulation model simple, these particular types of actions have been dis-allowed. As will be seen in Chapter 4, it is possible to add resolved signals to the semantics, but an extension is required.

Given the above listing of supported statements and assumptions, one can ask what physically constitutes a Femto-VHDL program. At the outermost level, a program text again is made up of one or more architectural components. Each component is comprised of either one or more concurrent statements, or the nesting of more components. The concurrent statements in turn contain sequential statements. The following abstract syntax for Boolean expressions, sequential statements and concurrent statements in Femto-VHDL may be given:

$$
\begin{array}{lll}
bexp & ::= & signal \mid signal\text{'delayed}(n) \mid signal\text{'event} \mid \text{not } bexp \mid \\
 & & bexp \text{ and } bexp \mid bexp \text{ or } bexp \mid bexp \text{ nand } bexp \mid \\
 & & bexp \text{ nor } bexp \mid bexp \text{ xor } bexp \mid bexp \text{ xnor } bexp \mid \text{true} \mid \text{false} \\
ss & ::= & ss \text{ ; } ss \mid \text{null} \mid bexp \text{ => } ss \mid ss \mid string \overset{\text{inert}}{:=} (bexp, n) \mid signal \overset{\text{trans}}{:=} (bexp, n) \\
cs & ::= & cs \parallel cs \mid sl : ss \\
sl & ::= & \{ s_1, \ldots, s_n \}
\end{array}
$$

CHAPTER 3

STATIC SEMANTICS

Before embarking on a formal discussion of the way VHDL simulates, the subset of the language that will be addressed needs to be identified. Furthermore, it will be necessary to define a way of checking that particular programs written in that subset are properly constructed. The exposition that follows begins by introducing the syntax of the subset and informally discussing the kind of restrictions that have been placed upon it. The chapter concludes with a suite of definitions that formalise the restrictions about the well-formedness of VHDL programs written in the subset.

3.1 Femto-VHDL

VHDL is a language that can best be thought of as one whose execution is based upon that of concurrent blocks. The most basic of these is the `process` statement. Each block contains sequential statements that describe the algorithm that it is supposed to be implementing. To a large extent, these sequential statements are exactly those of sequential Ada[1]. Only one particular class of sequential statements, signal assignments, is essential for driving the simulation. The emphasis in the definition of the subset will therefore be to highlight these two important types of statement (processes and signal assignments), and to provide enough other syntactic machinery to ensure that the semantics fully exercises them.

The VHDL subset for which a semantics has been defined is therefore rather small. Its name is derived from the smallest user-addressable unit of time in full VHDL – the femtosecond. Femto-VHDL does, however, contain enough constructs of the full language to illustrate the important features of the simulation model. As just discussed, only one kind of VHDL concurrent statement is supported, namely `process` statements. These are further restricted to those equipped with explicit sensitivity lists of signal names, which can be thought of as guards on their

[1]Ada is a registered trademark of the U.S. Government – Ada Joint Program Office.