a value at any given time point in $\mathsf{snd}(\theta_C\ t)$. The well-formedness of C leads with Theorems 3.3.3.1 and 3.3.2.1 to the fact that *ports"* is a subset of all the signals in C. So any port of C will always have a value in $\mathsf{snd}(\theta_C\ t)$. Furthermore, that value is the same irrespective of whether C is paired with A or with B.

Theorem 4.4.1:

let A = arch *ent* $arch_A$ *ports* $decls_A$ cs_A and B = arch *ent* $arch_B$ *ports* $decls_B$ cs_B
and $\sigma_{AC} = \sigma_A \cup \sigma_C$ and $\sigma_{BC} = \sigma_B \cup \sigma_C$
and $\theta_{AC}\ t = \big((\text{fst}(\theta_A\ t)) \cup (\text{fst}(\theta_C\ t)), (\text{snd}(\theta_A\ t)) \cup (\text{snd}(\theta_C\ t))\big)$
and $\theta_{BC}\ t = \big((\text{fst}(\theta_B\ t)) \cup (\text{fst}(\theta_C\ t)), (\text{snd}(\theta_B\ t)) \cup (\text{snd}(\theta_C\ t))\big)$ in
let $ports'$ = get_ports A and $ports''$ = get_ports C in
let $prts = ports' \cup ports''$ in

$$\begin{aligned}
&\big((dlys_A, now, \sigma_A, \gamma_A, \theta_A) \vdash \langle \text{dflat } A, \tau_A \rangle \xrightarrow{sim} beh_A \wedge \text{quiesce } dlys_A\ now\ \tau_A\ \gamma_A \wedge \\
&(dlys_B, now, \sigma_B, \gamma_B, \theta_B) \vdash \langle \text{dflat } B, \tau_B \rangle \xrightarrow{sim} beh_B \wedge \text{quiesce } dlys_B\ now\ \tau_B\ \gamma_B \wedge \\
&(dlys_{AC}, now, \sigma_{AC}, \gamma_{AC}, \theta_{AC}) \vdash \langle \text{dflat (dpar } A\ C), \tau_{AC} \rangle \xrightarrow{sim} beh_{AC} \wedge \\
&(dlys_{BC}, now, \sigma_{BC}, \gamma_{BC}, \theta_{BC}) \vdash \langle \text{dflat (dpar } B\ C), \tau_{BC} \rangle \xrightarrow{sim} beh_{BC} \wedge \\
&\text{quiesce } dlys_{AC}\ now\ \tau_{AC}\ \gamma_{AC} \wedge \text{quiesce } dlys_{BC}\ now\ \tau_{BC}\ \gamma_{BC} \wedge \\
&\text{well_formed_design } C \wedge \text{well_formed_}\theta\ \theta_C\ C \wedge \\
&\left(\sigma_{AC} \overset{\sigma}{=}_{prts} \sigma_{BC}\right) \wedge \left(beh_A \overset{\tau}{=}_{ports'} beh_B\right) \supset \left(beh_{AC} \overset{\tau}{=}_{prts} beh_{BC}\right)
\end{aligned}$$

The gist of the proof is to show that for any time t and for any signal *sig* in the set *prts*, the value associated with that signal is the same in both $(beh_{AC}\ t)$ and $(beh_{BC}\ t)$. In doing so, the following cases arise:

1. $sig \in ports'$ and $(sig, val) \in (beh_{AC}\ t)$, show $(sig, val) \in (beh_{BC}\ t)$
2. $sig \in ports'$ and $(sig, val) \in (beh_{BC}\ t)$, show $(sig, val) \in (beh_{AC}\ t)$
3. $sig \in ports''$ and $(sig, val) \in (beh_{AC}\ t)$, show $(sig, val) \in (beh_{BC}\ t)$
4. $sig \in ports''$ and $(sig, val) \in (beh_{BC}\ t)$, show $(sig, val) \in (beh_{AC}\ t)$

Because of the symmetry of $\overset{\tau}{=}$, this reduces down to only 1 and 3.

Recall that when a simulation quiesces, the resulting behaviour is a function from time to the state component σ of the trace θ used in the simulation environment. Each of the θ's here is a composite of the trace associated with the components of interest (A and B) and the overall design C. So, each of the beh_{XC} expands to $\big(\text{snd}(\theta_X\ t)\big) \cup \big(\text{snd}(\theta_C\ t)\big)$ where X is either A or B and the traces have also been augmented by the final state of the signals.

For those signals in $ports'$, Theorems 4.2.3.1 and 4.2.3.2 allow one to conclude the appropriate membership information. For the other signals (those explicitly associated with C in $ports''$) the argument devolves to a case analysis based on time. At time *now* the property holds because the starting state of the signals have been assumed to be equivalent. At all other times, the well-formedness of θ_C with respect to C and Theorem 4.2.2.1 lead to the knowledge that any signal mentioned in C has

4.4 Equivalence

One of the most powerful aspects of a formal semantics is its generality. VHDL is presented as a hardware description language permitting many levels of abstraction in the design process. Fundamental to this concept is the notion that one component of a design may be replaced by another equivalent component without affecting the overall design. In effect, this is an affirmation that the equivalence of two programs is a congruence.

What does it mean for two components to be equivalent? The simple answer is that their behaviours are "the same". For purposes of the work presented here, the simulation of a program must quiesce before it can be reasoned about. Once it has done so, the behaviour resulting from rule sim_1 in the definition of $\xrightarrow{sim}$ may be analysed. This means that the equivalence of two components is a comparison of the behaviours generated by each based on the relation $\overset{\tau}{=}$.

The property in question needs several assumptions to hold before it can be proven. For the most part these center around the way in which certain parts of the simulation environment for the overall design are constructed. For instance, the starting state of the signals must be a union of the σ associated with one of the sub-components and the starting σ for the rest of the program. This overall starting state of the signals must furthermore be $\overset{\sigma}{=}$-equivalent on all the signals of the composite program. A like construction with union is also required of the trace of past behaviour. The behaviours of the two subcomponents used in the replacement are required to be $\overset{\tau}{=}$-equivalent on their own ports. Naturally, these sub-programs must correspond to the same entity and have identical ports. Otherwise, it would be impossible to replace one by the other.

Well-formedness assumptions need only apply to the rest of the design into which the sub-components are being inserted. The reasoning behind this is fairly obvious. The sub-components have already been executed, and have a known behaviour. The program into which the substitution is being made is however not known. But, in order for it to simulate properly it must be well-formed both syntactically and with respect to its trace of past behaviour (which implies that it was well-formed on any simulation σ that was ever used).

simulate *design* ≡
  (well_formed_design *design*) ∧
  let *procs* = dflat *design* in
  (let σ = image (λ*sig*. (*sig*, ⊥)) (get_sigs_design *design*) in
   let γ = { } and τ = (λ*t*. { })
   and θ = (λ*t*. if (*t* = 0) then ({ }, σ) else ({ }, { })) in
      $(0, \sigma, \gamma, \theta) \vdash \langle procs, \tau \rangle \xrightarrow{init} \tau'$) ∧
  (let *dlys* = get_delays_cs *procs* and *now* = next_time τ′ in
   let γ′ = next_γ (τ′ *now*) γ and σ′ = next_σ (τ′ *now*) σ
   and θ′ = add_to_θ σ γ θ 0 *now*
   and τ″ = (λ*t*. if (*t* = 0) then { } else (τ′ *t*)) in
      $(dlys, now, \sigma', \gamma', \theta') \vdash \langle procs, \tau'' \rangle \xrightarrow{sim} beh$)

The Language Reference Manual stipulates that all signals must be stable at the beginning of the initialisation process. To that end, the initial events are an empty set. The starting transactions are also empty. Since signal values are always defined in the past [21], the initial trace is set up to ensure that any signal in the program which makes use of the delayed attribute will access the starting value of the signal.

Strictly speaking, the definition just presented is not quite complete. Ideally, one needs to have a facility that permits some signals begin the simulation with a value other than the default of low. The definition of simulate may be changed to reflect the desired functionality by adding a set of signal-value pairs called *uservals* to its parameter list and revising the way in which the original σ gets set up.

simulate *design uservals* ≡
  (well_formed_design *design*) ∧
  let *procs* = dflat *design* in
  (let *default*_σ = image (λ*sig*. (*sig*, ⊥)) (get_sigs_design *design*) in
   let σ = $\{(s, v)$: $((s, v) \in default_\sigma \wedge s \notin (\text{image fst } uservals)) \vee (s, v) \in uservals\}$ in
      …

It really does not matter if the user-supplied set of values contains signals other than those in the program being simulated. They will be added to the initial state, and will persist throughout the simulation. But, they will not affect its evaluation as they are never accessed by any of the statements in the program.

an example, the conditional if $(t \geq finish)$ then X else if $(start \leq t \geq finish)$ then Y else Z is the same as if $(t \geq finish)$ then X else if $(start \leq t < finish)$ then Y else Z

A useful theorem about the reduction of nested calls to add_to_θ with identical time arguments is:

Theorem 4.3.5.3:

$\forall \sigma' \gamma' \sigma'' \gamma'' \tau\ start\ finish.$

$\mathsf{add_to_}\theta\ \sigma'\ \gamma'\ (\mathsf{add_to_}\theta\ \sigma''\ \gamma''\ \tau\ start\ finish)\ start\ finish =$

$\mathsf{add_to_}\theta\ \sigma'\ \gamma'\ \tau\ start\ finish$

The result follows from the definition of add_to_θ. The nested calls create similarly nested conditionals where the innermost one is hidden by the outermost one. Obviously, this simplifies down to an identical conditional to that of the stand-alone invocation of add_to_θ.

The simulation loop itself is formulated in two rules. The first deals with quiescence, and ensures that the running θ is updated by the final σ and γ before returning a trace of signal values to the user (i.e. the second component of θ at any given time). The second rule states that if the processes in the Femto-VHDL program produce some transactions based on both the current environment and some starting transactions from which the current ones have been deleted, and that if the simulation loop may be performed on these results to give a final behaviour (using the functions defined above), then the processes yield this same behaviour in the starting environment.

$cs \rightarrow transactions \rightarrow transactions \rightarrow Boolean$

$$\{\mathsf{sim}_1\}\quad \frac{}{(dlys, t, \sigma, \gamma, \theta) \vdash \langle cs, \tau \rangle \xrightarrow{sim} \lambda t'.\, \mathsf{snd}(\mathsf{add_to_}\theta\ \sigma\ \gamma\ \theta\ t\ t\ t')}\ \mathsf{quiesce}\ dlys\ t\ \tau\ \gamma$$

$$\{\mathsf{sim}_2\}\quad \frac{(t, \sigma, \gamma, \theta) \vdash \langle cs, \tau'' \rangle \xrightarrow{cyc} \tau' \qquad (dlys, t', \sigma', \gamma', \theta') \vdash \langle cs, \tau' \rangle \xrightarrow{sim} beh}{(dlys, t, \sigma, \gamma, \theta) \vdash \langle cs, \tau \rangle \xrightarrow{sim} beh}$$

$$\tau'' = (\lambda t'.\ \mathsf{if}\ t' = t\ \mathsf{then}\ \{\}\ \mathsf{else}\ (\tau\ t'))$$
$$t' = \mathsf{next_time}\ dlys\ t\ \tau'$$
$$\sigma' = \mathsf{next_}\sigma\ (\tau'\ t')\ \sigma$$
$$\gamma' = \mathsf{next_}\gamma\ (\tau'\ t')\ \sigma$$
$$\theta' = \mathsf{add_to_}\theta\ \sigma\ \gamma\ \theta\ t\ t'$$

Having specified the top-level simulation loop, it is now possible to make a definition characterising a complete simulation:

using them as a filter. A new set of signal-value pairs is then created from the new transactions and the old state less those pairs which have a signal name in common with the new pairs.

next_ $\sigma\ \sigma'\ \sigma''$ ≡
  let *sigs* = image fst σ' in $\{(s,v):(s,v)\in\sigma'\vee((s,v)\in\sigma''\wedge s\notin sigs)\}$

For instance, next_ $\sigma\ \{(a,\mathsf{T})\}\ \{(a,\bot),(b,\bot)\}$ reduces to $\{(a,\mathsf{T}),(b,\bot)\}$.

A similar approach is used to generate a new set of events. The function next_ γ creates a set of signal names from those pairs with a common signal name in the new and old σ's whose values differ.

next_ $\gamma\ \sigma'\ \sigma''$ ≡
  $\{signl:\exists x\ y.\ (signl,x)\in\sigma'\wedge(signl,y)\in\sigma''\wedge(x\neq y)\}$.

Using the same inputs as the previous example, next_ $\gamma\ \{(a,\mathsf{T})\}\ \{(a,\bot),(b,\bot)\}$ becomes $\{a\}$.

The final auxiliary function required by the simulation loop merges a set of events and a set of signal-value pairs into the running trace of past activity. The function add_to_ θ takes the components to be added, the current θ, the beginning of the past time interval and the endpoint of the past time interval (i.e. the point of computation that has just been entered), and returns a new θ. If the beginning time and the endpoint are the same (meaning that the simulation has quiesced, or a δ-step has been made), then the function returned is predicated on the time input to it being the same as or greater than the beginning time, and allows the just-added information to be accessed. Otherwise, the function returned creates an interval of times for which the new trace information may be observed.

add_to_ $\theta\ \sigma\ \gamma\ \theta$ *start finish* ≡
  λt.if $(start = finish)$ then $\big($if $(t\geq start)$ then (γ,σ) else $(\theta\ t)\big)$
    else $\big($if $(start\leq t\geq finish)$ then (γ,σ) else $(\theta\ t)\big)$

There is no cause for concern in the use of ≥ to delineate the upper bound of a time interval. If the simulation has quiesced, then the final values of the signals should be expected to hold for any time after the end of the simulation. Conversely, if the simulation has not quiesced, then the non-decreasing nature of simulation time will ensure that the right information is accessible as the nested conditional is built. As

$cs \rightarrow transactions \rightarrow transactions \rightarrow Boolean$

$$\{\text{init}_1\}\quad \frac{(t,\sigma,\gamma,\theta) \vdash \langle ss,\tau\rangle \xrightarrow{seq} \tau'}{(t,\sigma,\gamma,\theta) \vdash \langle sl\text{:}ss,\tau\rangle \xrightarrow{init} \tau'}$$

$$\{\text{init}_2\}\quad \frac{\rho \vdash \langle cs',\tau\rangle \xrightarrow{init} \tau' \quad \rho \vdash \langle cs'',\tau\rangle \xrightarrow{init} \tau''}{\rho \vdash \langle cs' \| cs'',\tau\rangle \xrightarrow{init} \tau' \ \text{clean_zip}\ \tau\ \tau'\ \tau''}$$

4.3.5 Rules for the Simulation Loop

A key component of the simulation loop is the ability to move from the current point of computation to the next interesting one. Recall that the first component of the simulation environment μ is an ordered list of times that represent the delays in a given Femto-VHDL program text. So, to determine the next point of computation, it is necessary to run through the list until the first non-empty set of transactions is reached. The function next_time is defined recursively over the structure of finite lists to accomplish this task:

next_time *dlys now* $\tau \equiv$
  if (nil *dlys*) then *now*
  elseif $\neg\left(\left(\tau\ (now + (\text{hd}\ dlys))\right) = \{\}\right)$ then $(now + (\text{hd}\ dlys))$
  else next_time (tl *dlys*) *now* τ

Another important part of the simulation loop is the determination of whether or not a particular simulation has quiesced. The job of finding this out is analogous to discovering the next point of computation. If a non-empty set of transactions is found while iterating through the list of delays, the simulation has not quiesced. Furthermore, after exhausting all possible delays it is essential to check the emptiness of the current set of events. Obviously, a non-empty γ also means that the simulation has not quiesced.

quiesce *dlys now* $\tau \equiv$
  if (nil *dlys*) then $(\gamma = \{\})$
  elseif $\left(\left(\tau\ (now + (\text{hd}\ dlys))\right) \neq \{\}\right)$ then $\bot$
  else quiesce (tl *dlys*) *now* τ

If the simulation has not quiesced, it is necessary to merge the signal changes dictated by the current transactions into the static state of signal values. This is done by first extracting the names of the signals for which a transaction has occurred and

($\lambda t.\{\}$). Since any τ zip'ed with empty transactions is that τ, the desired conclusion follows trivially.

The second theorem expresses the commutativity of two τ's when the original transactions remain unchanged.

Theorem 4.3.3.6:
$\forall orig\ \tau'\ \tau''.\ \mathsf{clean_zip}\ orig\ \tau'\ \tau'' = \mathsf{clean_zip}\ orig\ \tau''\ \tau'$

Once one recalls that zip is a commutative operation, the result follows trivially from the definition of clean_zip.

Armed with the definition of clean_zip, the rules that define a simulation cycle may be given as:

$cs \to transactions \to transactions \to Boolean$

$$\{\mathsf{cs}_1\}\quad \frac{}{(t,\sigma,\gamma,\theta) \vdash \langle sl : ss, \tau\rangle \xrightarrow{cyc} \lambda t.\{\}} \quad (sl \cap \gamma) = \{\}$$

$$\{\mathsf{cs}_2\}\quad \frac{(t,\sigma,\gamma,\theta) \vdash \langle ss, \tau\rangle \xrightarrow{seq} \tau'}{(t,\sigma,\gamma,\theta) \vdash \langle sl : ss, \tau\rangle \xrightarrow{cyc} \tau'} \quad (sl \cap \gamma) \neq \{\}$$

$$\{\mathsf{cs}_3\}\quad \frac{\rho \vdash \langle cs', \tau\rangle \xrightarrow{cyc} \tau' \quad \rho \vdash \langle cs'', \tau\rangle \xrightarrow{cyc} \tau''}{\rho \vdash \langle cs' \,||\, cs'', \tau\rangle \xrightarrow{cyc} \tau'\ \mathsf{clean_zip}\ \tau\ \tau'\ \tau''}$$

Note that inactive processes return empty transactions as in the intermediate solution above. Also, the call to zip in the rule defining the execution of concurrent processes has been replaced by a call to clean_zip. These modifications guarantee that as long as the program is well-formed, there can be no clash of signal names in the transactions resulting from the parallel execution of two processes.

4.3.4 Rules for Initialisation

Initialisation in VHDL is characterised by a more general version of the simulation cycle. All processes are activated irrespective of sensitivity lists, and run once [21]. In terms of rules, it is possible to remove the rule about non-active processes from the rules for concurrent statements, as well as the side condition from the second rule. This gives rise to the much simpler relation $\xrightarrow{init}$.

The problem with the above rules is that it is possible for a perfectly well-formed program to generate conflicting transactions, thereby not allowing Theorem 4.3.3.1 to be used to get back to Zip. Take for instance the simple case of a program made up of two processes A and B where the starting τ contains a transaction scheduled for signal x. Assume A does not activate, and returns the original transactions unchanged. Also assume that B does activate, and pre-empts the transaction on x. The result of zip'ing together the old and new transactions is a conflict on signal x. According to our intuitive understanding of a simulation cycle, there should not have been a conflict resulting from the execution of these processes.

The seemingly obvious solution to the problem is to have inactive processes return empty transactions ($\lambda t.\{\}$). While this would take care of most situations, the case of a process activating but not scheduling any new transactions (i.e. passing on the old ones unchanged through null statements) engenders exactly the same problem. Clearly, some augmentation of zip is required.

Intuitively, any original transactions need to be stripped out of the returned τ's before they are zip'ed together. Furthermore, any signals for which there are scheduled transactions in the new τ's need to have any reference to them removed from the original transactions. To that end, the function clean_zip is defined as:

clean_zip *orig* τ' τ'' ≡
  let strip_τ' = $(\tau'\ t) - (orig\ t)$
  and strip_τ'' = $(\tau''\ t) - (orig\ t)$ in
  let τ = strip_τ' zip strip_τ'' in
  let $new_orig\ t = (orig\ t) - \{(x,y): (x \in (\text{image fst } (\tau\ t))) \wedge ((x,y) \in (orig\ t))\}$ in
    τ zip *new_orig*

A pair of theorems about clean_zip now follow. The first one deals with the simple case of the original transactions being empty ($\lambda t.\{\}$), and shows that such a case degenerates to a simple zip of the remaining τ's.

Theorem 4.3.3.5:
$\forall \tau'\ \tau''.\ (\text{clean_zip}\ (\lambda t.\{\})\ \tau'\ \tau'') = (\tau'\ \text{zip}\ \tau'')$

The argument is based on the fact that any set s less the empty set is s. Therefore, strip_τ' and strip_τ'' in the definition of clean_zip reduce to τ' and τ'' respectively. As a result, τ becomes a zip of τ' and τ''. It is also known that the empty set less any other set is always the empty set. As a consequence, *new_orig* reduces to

Theorem 4.3.3.1: (Equality of Zip and zip)
$\forall \tau' \; \tau'' \; t.$
let $x =$ image fst $\{(x,y):(x,y) \in (\tau' \; t)\}$ and $y =$ image fst $\{(x,y):(x,y) \in (\tau'' \; t)\}$ in
$(\text{disjoint } x \; y) \Rightarrow ((\tau' \; \text{Zip} \; \tau'') = (\tau' \; \text{zip} \; \tau''))$

Armed with the reassurance provided by this fact, use may be made of the simpler zip in the rules which are to come. This is because the two definitions are interchangeable as long as the pre-conditions on Zip hold.

Some other useful theorems for manipulating expressions containing zip include commutativity, associativity and reduction when one of the arguments is a τ that always points to an empty σ. As zip is based on set union, they mirror properties associated with it.

Theorem 4.3.3.2: $\forall \tau' \; \tau''. (\tau' \; \text{zip} \; \tau'') = (\tau'' \; \text{zip} \; \tau')$

Theorem 4.3.3.3: $\forall \tau \; \tau' \; \tau''. (\tau \; \text{zip} \; (\tau' \; \text{zip} \; \tau'')) = ((\tau \; \text{zip} \; \tau') \; \text{zip} \; \tau'')$

Theorem 4.3.3.4: $(\forall \tau. ((\lambda t. \{\}) \; \text{zip} \; \tau) = \tau) \wedge (\forall \tau. (\tau \; \text{zip} \; (\lambda t. \{\})) = \tau)$

Given the definition of zip, a first attempt at the rules that define the execution of a simulation cycle may be attempted. The first rule states that if the sensitivity list of a particular process does not intersect with the set of current events, the transactions remain unchanged. The second rule is the inverse of the first. That is to say that if the intersection of sensitivity list and events is non-empty, then the resulting transactions arise from the execution of the sequential statements of the process. The final rule says that if starting from the same set of transactions, two concurrent statements produce their own version of future transactions, then their parallel execution results in the zip'ing together of those transactions.

cs → transactions → transactions → Boolean

$$\{\text{cs}_1\} \quad \frac{}{(t,\sigma,\gamma,\theta) \vdash \langle sl : ss, \tau \rangle \xrightarrow{cyc} \tau} \quad (sl \cap \gamma) = \{\}$$

$$\{\text{cs}_2\} \quad \frac{(t,\sigma,\gamma,\theta) \vdash \langle ss, \tau \rangle \xrightarrow{seq} \tau'}{(t,\sigma,\gamma,\theta) \vdash \langle sl : ss, \tau \rangle \xrightarrow{cyc} \tau'} \quad (sl \cap \gamma) \neq \{\}$$

$$\{\text{cs}_3\} \quad \frac{\rho \vdash \langle cs', \tau \rangle \xrightarrow{cyc} \tau' \quad \rho \vdash \langle cs'', \tau \rangle \xrightarrow{cyc} \tau''}{\rho \vdash \langle cs' || cs'', \tau \rangle \xrightarrow{cyc} \tau' \; \text{zip} \; \tau''}$$

4.3.3 Rules for Concurrent Statements

The rules that define the evaluation of Femto-VHDL concurrent statements are really those that describe a single iteration of a simulation cycle. As explained in Chapter 1, a simulation cycle is based on the notion of processes running in parallel, and returning a composite view of future behaviour after they have all finished executing. To effect this amalgamation, it should only be necessary to perform a simple set union at all times in two returned τ's on the σ's found there. It now becomes evident why the multiple driver restriction has been placed on Femto-VHDL. If such a union was being done for two particular τ's at some time t, and both σ's found there contained a projected value on the same signal, the consistency constraint of unique values for signals would be violated. In a fuller semantics for the VHDL simulation engine, it would be necessary to replace the set union with a more complex function that took into account user-defined resolution functions.

The kind of behaviour required in joining together transactions could be given by the function Zip with the property that:

$$\forall \tau'\ \tau''\ t.$$
$$\text{let } x = \text{image fst } \{(x,y):(x,y) \in (\tau'\ t)\} \text{ and } y = \text{image fst } \{(x,y):(x,y) \in (\tau''\ t)\} \text{ in}$$
$$(\text{disjoint } x\ y) \Rightarrow ((\tau'\ \textbf{Zip}\ \tau'') \equiv \lambda t.(\tau'\ t) \cup (\tau''\ t))$$

Zip is different from previous definitions that have been made in that it is partially specified. The only statement that has been made is what should occur when the two sets are disjoint. The behaviour is undefined when they are not. One would expect the simulation of a well-formed Femto-VHDL program to ensure that those conditions are met. Recall that the multiple driver restriction ensures that only one process may assign to a particular signal, which should cause the transactions at any particular time to be disjoint during simulation. It is therefore possible to re-cast the operation as a more familiar and simple definition:

$$\tau'\ \textbf{zip}\ \tau'' \equiv \lambda t.(\tau'\ t) \cup (\tau''\ t)$$

In fact, the following obvious theorem about the relationship between Zip and zip holds:

inert_post *sig val curval* τ *now dly* ≡
  if (is_stable *dly* τ *now sig curval*) then trans_post *sig val* τ (*now* + *dly*)
  else trans_post *sig val* (purge *dly* τ *now sig val curval*) (*now* + *dly*)

The arguments to inert_post should be read as (from left to right) the signal in question, the value being assigned, the current value of the signal, the projected transactions, the current time and the delay associated with the assignment.

The rules defining the evaluation of Femto-VHDL sequential statements are now given. The relation makes use of the one defined for Boolean expressions in many places. The $\langle x, \tau \rangle$ notation is used for grouping together a given statement with the current transactions (e.g. $\langle \mathsf{null}, \tau \rangle$). An axiom defines that a null statement should do nothing to the transactions, passing them along unchanged. Sequencing of Femto-VHDL statements and the evaluation of conditionals proceed in the expected way. It is, however, in defining the evaluation of the two kinds of assignment statements that the driving force of the simulation engine is addressed. The result of performing an assignment statement is the modification of future transactions through inert_post and trans_post.

ss → *transactions* → *transactions* → *Boolean*

$$\{\mathsf{ss}_1\} \quad \frac{}{\rho \vdash \langle \mathsf{null}, \tau \rangle \xrightarrow{seq} \tau}$$

$$\{\mathsf{ss}_2\} \quad \frac{\rho \vdash \langle ss', \tau \rangle \xrightarrow{seq} \tau' \quad \rho \vdash \langle ss'', \tau' \rangle \xrightarrow{seq} \tau''}{\rho \vdash \langle ss'; ss'', \tau \rangle \xrightarrow{seq} \tau''}$$

$$\{\mathsf{ss}_3\} \quad \frac{\rho \vdash e \xrightarrow{bool} \mathsf{T} \quad \rho \vdash \langle ss', \tau \rangle \xrightarrow{seq} \tau'}{\rho \vdash \langle e \Rightarrow ss' \mid ss'', \tau \rangle \xrightarrow{seq} \tau'}$$

$$\{\mathsf{ss}_4\} \quad \frac{\rho \vdash e \xrightarrow{bool} \bot \quad \rho \vdash \langle ss'', \tau \rangle \xrightarrow{seq} \tau'}{\rho \vdash \langle e \Rightarrow ss' \mid ss'', \tau \rangle \xrightarrow{seq} \tau'}$$

$$\{\mathsf{ss}_5\} \quad \frac{(t, \sigma, \gamma, \theta) \vdash e \xrightarrow{bool} x}{(t, \sigma, \gamma, \theta) \vdash \left\langle sig \overset{\mathsf{inert}}{:=} (e, dly), \tau \right\rangle \xrightarrow{seq} \mathsf{inert_post}\ sig\ x\ (\mathsf{valcalc}\ sig\ \sigma)\ \tau\ t\ dly}$$

$$\{\mathsf{ss}_6\} \quad \frac{(t, \sigma, \gamma, \theta) \vdash e \xrightarrow{bool} x}{(t, \sigma, \gamma, \theta) \vdash \left\langle sig \overset{\mathsf{trans}}{:=} (e, dly), \tau \right\rangle \xrightarrow{seq} \mathsf{trans_post}\ sig\ x\ \tau\ (t + dly)}$$

The main concern before doing the assignment is therefore to ensure the stability of the input waveform. To do so, pending transactions during the assignment interval will have to be examined. At the lowest level, a check will have to be made on planned σ to see that the signal value that has been projected in that σ is the same as the one trying to be scheduled.

check σ *sig val* ≡
  let $sigval = \{(x,y):(x,y) \in \sigma \wedge (x = sig)\}$ in
    if $(sigval = \{\})$ then T else $\left(sigval = \{(sig, val)\}\right)$

The function also flags as acceptable those σ that have no projected value for the given signal.

The above check will have to be made on a specific range in the future transactions. Those that are of interest lie between whatever the current simulation time is and that time plus the delay associated with the assignment statement in question.

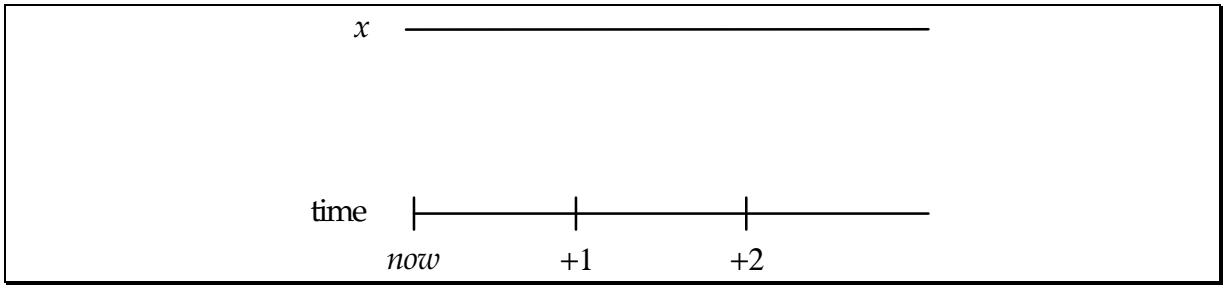
is_stable *dly* τ *now sig val* ≡
  if $(dly = 0)$ then T
  else $\left(\text{check } (\tau\ (now + dly))\ sig\ val\right) \wedge \left(\text{is_stable } (dly - 1)\ \tau\ now\ sig\ val\right)$

Signals are assumed to be trivially stable at the current time. For all other times in the delay, one recurses back check'ing each time point.

If there is an instability, it will be necessary to purge any spikes that are found in the assignment interval. The function purge, similar to is_stable, accomplishes the task:

purge *dly* τ *now sig val* ≡
  if $(dly = 0)$ then τ
  else let $\tau' = \lambda t.$if $(t = now + dly)$ then
    $(\tau(now + dly)) - \{(x,y):(x,y) \in \tau(now + dly) \wedge x = sig \wedge y \neq val\}$
    else τ *t* in
    purge $(dly - 1)$ τ′ *now sig val*

Armed with the definitions of is_stable and purge it is now possible to give the function which performs the scheduling of inertial delay assignments. Intuitively, as long as the stability constraint holds, the assignment is nothing more than what was done for transport delay. If the signal is not stable during the interval, then the spikes should be purged from the interval before the assignment can take place.



The difference results from the fact that x is not stable during the long assignment interval, and is disqualified from assignment at time $now+1$ because the transaction scheduled there represents a spike in the interval between times now and $now+2$.

Transport delay will expressed by the function trans_post:

trans_post $sig\ val\ \tau\ t \equiv$
let stripped $t' = (\tau\ t') - \{(x,y):(x,y) \in (\tau\ t') \wedge (x = sig) \wedge (t' \geq t)\}$ in
$\lambda t'$.if $(t' = t)$ then $((sig, val)$ insert (stripped $t'))$
else (stripped t')

The function makes use of insert to insert a new element in to a set (i.e. 1 insert $\{3,4\}$ becomes $\{1,3,4\}$). The current transactions (τ), the signal for which the assignment is to be made (sig), the value that the signal is supposed to take on (val) and the time at which that assignment is to take place (t) are employed to create a new version of future transactions by returning a function from time to σ. If the time supplied to this function is the one at which the signal assignment is to take place, a set consisting of a "stripped" version of the projected σ with the given signal-value pair inserted into it is returned. Otherwise, all that is returned is the stripped version of the projected state. The stripping process is performed by a function local to trans_post called stripped, which in fact does the pre-emption part of scheduling a new transaction. stripped, when given a time t', will return a version of the starting τ in which all pending transactions on signal sig have been removed when t' is greater than or equal to the time at which the current scheduling is to take place.

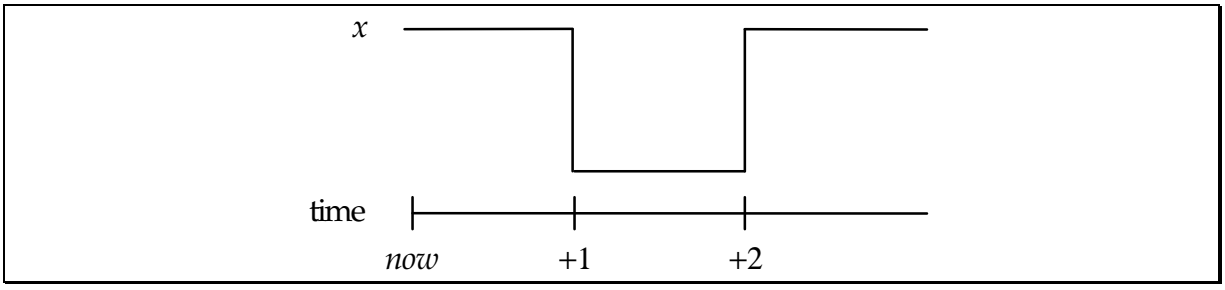
Inertial delay signal assignment statements (i.e. those with the kind of set-up and hold constraints mentioned earlier) make use of a function that is rather different from trans_post to perform the posting of transactions. The difference lies in the stability check and its consequences on the act of scheduling. With transport delay, a transaction will always be posted. With inertial delay, this is not the case as a non-stable waveform will cause transactions to be unscheduled.

to effect that assignment. These transactions are in turn used by the overall simulation loop to determine where the next point of computation lies.

During signal assignment, it is essential that the pre-emptive scheduling required by VHDL take place [21]. This type of scheduling is a particular artifact of the VHDL simulation model that may delete certain previously posted transactions, and is best illustrated by example. Assume that signal x is scheduled to take on a value at time t. Also assume that an attempt is made to post another value for the same signal at time t'. If t' is less than or equal to t, then the first value scheduled at time t (along with any other value previously posted at or after time t') is deleted from the future transactions, and the new transaction for time t' is posted. The inverse situation (t' greater than t) is not part of pre-emptive scheduling, but rather relates to inertial delay assignments in particular.

Transport delay, barring pre-emptive scheduling, is analogous to saying that whatever value gets assigned to a signal eventually appears. Inertial delay makes use of exactly the same mechanism as transport delay, but in a constrained way. In order for a particular transaction to be posted, inertial delay further requires that the inputs to the assignment be stable during the entirety of the assignment interval, and thereby removes any "spikes" on the input.

As an example of the difference between the two kinds of delay, consider the following example. Assume that the signal x has a high value. Furthermore, assume that the signal is scheduled to take on a low value at time $now+1$. If one then attempts to post a high for x at time $now+2$ using transport delay, the waveform for x would look like:



If, however, the same scheduling were to take place with inertial delay, the waveform would take on the following form:

not of that expression evaluates to the logical negation of x. The symbols T and $\perp$ below stand for "true" and "false" respectively. Furthermore, $\oplus$ is the exclusive or operation.

$bexp \rightarrow Boolean \rightarrow Boolean$

$$\{b_1\}\ \frac{}{(now,\sigma,\gamma,\theta) \vdash sig \xrightarrow{bool} \text{valcalc } sig\ \sigma}$$

$$\{b_2\}\ \frac{}{\rho \vdash \text{true} \xrightarrow{bool} \mathsf{T}} \qquad \{b_3\}\ \frac{}{\rho \vdash \text{false} \xrightarrow{bool} \perp}$$

$$\{b_4\}\ \frac{}{(now,\sigma,\gamma,\theta) \vdash sig\text{'delayed}(t) \xrightarrow{bool} \text{valcalc } sig\ (\text{snd}(\theta(now-t)))}$$

$$\{b_5\}\ \frac{}{(now,\sigma,\gamma,\theta) \vdash sig\text{'event} \xrightarrow{bool} sig \in \gamma}$$

$$\{b_6\}\ \frac{\rho \vdash e \xrightarrow{bool} x}{\rho \vdash (\text{not } e) \xrightarrow{bool} \neg x}$$

$$\{b_7\}\ \frac{\rho \vdash e' \xrightarrow{bool} x \quad \rho \vdash e'' \xrightarrow{bool} y}{\rho \vdash (e' \text{ and } e'') \xrightarrow{bool} (x \wedge y)} \qquad \{b_8\}\ \frac{\rho \vdash e' \xrightarrow{bool} x \quad \rho \vdash e'' \xrightarrow{bool} y}{\rho \vdash (e' \text{ or } e'') \xrightarrow{bool} (x \vee y)}$$

$$\{b_9\}\ \frac{\rho \vdash e' \xrightarrow{bool} x \quad \rho \vdash e'' \xrightarrow{bool} y}{\rho \vdash (e' \text{ nand } e'') \xrightarrow{bool} \neg(x \wedge y)} \qquad \{b_{10}\}\ \frac{\rho \vdash e' \xrightarrow{bool} x \quad \rho \vdash e'' \xrightarrow{bool} y}{\rho \vdash (e' \text{ nor } e'') \xrightarrow{bool} \neg(x \vee y)}$$

$$\{b_{11}\}\ \frac{\rho \vdash e' \xrightarrow{bool} x \quad \rho \vdash e'' \xrightarrow{bool} y}{\rho \vdash (e' \text{ xor } e'') \xrightarrow{bool} (x \oplus y)} \qquad \{b_{12}\}\ \frac{\rho \vdash e' \xrightarrow{bool} x \quad \rho \vdash e'' \xrightarrow{bool} y}{\rho \vdash (e' \text{ xnor } e'') \xrightarrow{bool} \neg(x \oplus y)}$$

The only non-trivial rule is perhaps b_4 which defines the calculation of a delayed value for a particular signal *sig*. The result arises from looking at the state of the signal values $now - t$ units ago and extracting the value of the signal in question via valcalc. The state of interest is found by examining the trace of past history represented by θ at the desired time, and extracting the second element of the pair.

4.3.2 Rules for Sequential Statements

The formalisation of the sequential statements of Femto-VHDL is for the most part as obvious as that for Boolean expressions. The rules, in addition to the other sequential constructs in the subset, encapsulate the statements which drive the simulation forward – inertial and transport delay signal assignment. Whenever a signal assignment statement of either form is encountered, it becomes necessary to "post", or schedule, a transaction to take place at some future point of computation

The definition of $\stackrel{\tau}{=}$ makes it clear that any signal named in *ports* will have the same signal-value pair in both τ' and τ'' at any chosen time. Each of the desired implications is a consequence of this.

4.3 Rules of the Semantics

The rules that define the operational semantics of the Femto-VHDL subset fall under five broad headings – rules for Boolean expressions, rules for sequential statements, rules for concurrent statements (the simulation cycle), rules for initialisation and rules for the top-level simulation loop. The presentation of these rules now follows, beginning with those defining the evaluation of Boolean expressions. Their format is based on the presentation discussed in Section 4.1.2.

4.3.1 Rules for Boolean Expressions

The rules that define the semantics of Boolean expressions are numerous, but are conceptually very simple. They are, in essence, describing an evaluator for Boolean expressions. A key component of that evaluation will be the determination of the value of actual signals – a task accomplished by the function valcalc:

$$\textbf{valcalc}\ sig\ \sigma \equiv \mathsf{choice}\ \{val : (sig, val) \in \sigma\}$$

The function calculates the value of a signal *sig* in a state σ, by making use of the function choice which selects an arbitrary element from a non-empty set. The use of choice is possible in this context because the set from which the choice is made will always be a singleton set. This is due to the fact that signals have unique values in any σ as long as they are well-formed. For example, $\mathsf{valcalc}\ x\ \{(y,a),(x,b)\} = b$.

In the rules, the symbol $\xrightarrow{bool}$ denotes the evaluation relation for Femto-VHDL Boolean expressions. The type of $\xrightarrow{bool}$ is $bexp \to Boolean \to Boolean$, and is given in the upper left-hand corner of the box displaying the rules. This type information will be a recurrent pattern in the presentation of the other rule sets. Axioms are provided for the evaluation of individual signals, the values of the constants true and false, the calculation of the delayed value of a signal and the determination of whether or not there is an event occurring on a signal. The other inference rules define the relationship between Femto-VHDL Boolean operators, and the logical Boolean expressions that they evaluate to. For instance, rule b_6 states that if some Femto-VHDL Boolean expression e evaluates to the logical value x, then the

It is an equivalence relation, and obviously exhibits the required properties of reflexivity, symmetry and transitivity. A similar relationship, may be defined for σ:

$$\sigma' \underset{sigs}{\overset{\sigma}{=}} \sigma'' \equiv \left(\{(s,v) : s \in \sigma' \wedge s \in sigs\} = \{(s,v) : s \in \sigma'' \wedge s \in sigs\}\right)$$

Both τ and θ are a bit more involved when defining equivalence. This is due to the fact that they are functions of time. Therefore, the simple approach taken above must be augmented to check that the sets are the same for all times. For τ, the relation $\overset{\tau}{=}$ may is defined as:

$$\tau' \underset{sigs}{\overset{\tau}{=}} \tau'' \equiv \forall t. \left(\{(s,v) : s \in (\tau'\ t) \wedge s \in sigs\} = \{(s,v) : s \in (\tau''\ t) \wedge s \in sigs\}\right)$$

The definition of a relation on θ is much the same. Note that the definition of $\overset{\theta}{=}$ takes both the trace of γ and the trace of σ into account.

$$\theta' \underset{sigs}{\overset{\theta}{=}} \theta'' \equiv \forall t. \left(\{s : s \in (\mathsf{fst}\ (\theta'\ t)) \wedge s \in sigs\} = \{s : s \in (\mathsf{fst}\ (\theta''\ t)) \wedge s \in sigs\}\right) \wedge$$
$$\left(\{(s,v) : s \in (\mathsf{snd}\ (\theta'\ t)) \wedge s \in sigs\} = \{(s,v) : s \in (\mathsf{snd}\ (\theta''\ t)) \wedge s \in sigs\}\right)$$

Later on in the presentation of the semantics it will become necessary to reason about membership of signal value pairs in equivalent τ's. The following complementary theorems will aid in that reasoning. They merely state that if two τ's are know to be equivalent on a set of ports, then given any signal named in those ports which is in one of τ's the same signal-value pair must appear the other one as well.

Theorem 4.2.3.1:

$$\forall \tau'\ \tau''\ t\ sig\ val\ ports.$$
$$\left(\left(\tau' \underset{ports}{\overset{\tau}{=}} \tau''\right) \wedge (sig \in ports) \wedge (sig,val) \in (\tau'\ t)\right) \supset (sig,val) \in (\tau''\ t)$$

Theorem 4.2.3.2:

$$\forall \tau'\ \tau''\ t\ sig\ val\ ports.$$
$$\left(\left(\tau' \underset{ports}{\overset{\tau}{=}} \tau''\right) \wedge (sig \in ports) \wedge (sig,val) \in (\tau''\ t)\right) \supset (sig,val) \in (\tau'\ t)$$

the top-level simulation loop, will ensure that traces of past behaviour are constructed to contain intervals rather than time points.

The function well_formed_θ is defined to encapsulate the two requirements on the proper construction of individual θ. These constraints apply, as in well_formed_σ, with respect to the design that a particular θ is associated with. The first of the two constraints is that any signal found as an event in the trace must be one of the signals of the design in question. The second requires that for any time point mentioned, the σ found there must be well-formed.

$$\textbf{well_formed_}\theta\ \theta\ \mathit{design} \equiv \\ \forall t\ \mathit{sig}.\left(\mathit{sig} \in \left(\mathsf{fst}\ (\theta\ t)\right) \supset \mathit{sig} \in (\mathsf{get_sigs_design}\ \mathit{design})\right) \wedge \\ \left(\mathsf{well_formed_}\sigma\ \left(\mathsf{snd}\ (\theta\ t)\right)\ \mathit{design}\right)$$

It will be necessary in later reasoning to know about membership in a trace of past activity based on the well-formedness of that history:

Theorem 4.2.2.1:

$$\forall \theta\ \mathit{dsgn}. \\ \mathsf{well_formed_}\theta\ \theta\ \mathit{dsgn} \supset \forall t\ s.s \in \mathsf{get_sigs_design}\ \mathit{dsgn} \supset \exists v.\ (s,v) \in \mathsf{snd}(\theta\ t)$$

If one assumes well_formed_θ θ *dsgn*, expand its definition (as well as that of well_formed_σ), reduce the resulting internal let expressions, and take the second conjunct, the required result is obtained after discharging the initial assumption and generalising both θ and *dsgn*.

4.2.3 Equivalence

Equivalence for γ, σ, τ and θ is not a straight equality. Rather it is predicated on the user supplying some signals of interest. The reasoning behind this is perhaps best illustrated by example. Assume that one wished to compare the simulation γ associated with architecture A with that for architecture B. Both A and B have the same external ports, but differ in their internally declared signals. Since any comparison of A with B is only meaningful on externally visible signals, it is essential that the internal signals be factored out of each γ before making the equality test on the two sets. So, the relation $\overset{\gamma}{=}$ is defined in the following way:

$$\gamma' \underset{sigs}{\overset{\gamma}{=}} \gamma'' \equiv \left(\{s : s \in \gamma' \wedge s \in sigs\} = \{s : s \in \gamma'' \wedge s \in sigs\}\right)$$

get_delays_cs $cs \equiv$
  if $cs = (sl : ss)$ then get_delays_ss ss []
  else let $(cs' \,||\, cs'') = cs$ in merge_delays (get_delays_cs cs') (get_delays_cs cs'')

An example of the use of this suite of functions may be had by taking some example processes used in Chapter 3,

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

and running get_delays_cs over them to return the list $[0, 2]$.

4.2.2 Well-Formedness

A well-formed σ has two essential parts. The first being that there are no signal-value pairs that have the same signal name as their first element (i.e. signals have unique values). The second criterion is that if an architecture simulates with a particular σ in its environment, then all the signals in the architecture must have an associated value in that σ. The definition of well_formed_σ encapsulates these properties:

well_formed_σ σ $design \equiv$
  let $sigs$ = image fst σ in
    card σ = card $sigs \wedge$
    $\forall sig.(sig \in (\text{get_sigs_design } design)) \supset \exists val.\, (sig, val) \in \sigma$

The function uses card to calculate the cardinality of a finite set (i.e. card $\{a, b\} = 2$ and card $\{\,\} = 0$), and image to extract the set of signal names from the given σ (i.e. image fst $((x, a), (y, b), (y, c)) = \{x, y\}$). The first conjunct simply states that the σ in question must have the same number of elements as the set of signal names associated with that σ. The second conjunct stipulates that any signal that is in the set of signal names extracted from the design, must have a value associated with it in σ.

When dealing with the well-formedness of traces of past behaviour (θ), two constraints must be met. Both will apply at any time rather than at specific offsets associated with a particular program. The reason is twofold. First, it makes little intuitive sense to have a past which is only applicable at certain time points. Second, a 'delayed signal attribute may attempt to sample the past at a time offset which is not one of the signal assignment delays. The semantics, particularly the definition of

4.2.1 Delay Extraction

Bearing in mind the usefulness of the just mentioned delays, functions are now defined to extract them from Femto-VHDL programs. Before doing so, two functions that ensure the ordering of lists of delays are given. The first function is list recursive, and adds a new delay to the existing list of delays. It guarantees that any such addition preserves the order of the list.

add_one_delay $dly\ dlys \equiv$
  if nil $dlys$ then $[dly]$
  elseif $dly = (\text{hd}\ dlys)$ then $dlys$
  elseif $dly < (\text{hd}\ dlys)$ then $(dly\ .\ dlys)$
  else $(\text{hd}\ dlys\ .\ \text{add_one_delay}\ delay\ (\text{tl}\ dlys))$

The next function is also list recursive, and merges together two lists of delays. Use is made of add_one_delay to preserve the ordering. merge_delays is really doing nothing more than a sorted appending operation that deletes duplicate elements.

merge_delays $dlys'\ dlys'' \equiv$
  if nil $dlys'$ then $dlys''$
  else merge_delays $(\text{tl}\ dlys)\ (\text{add_one_delay}\ (\text{hd}\ dlys')\ dlys'')$

As an example, merge_delays $[2,5,1]\,[4,5,8]$ would yield the list $[1,2,4,5,8]$.

Having defined both add_one_delay and merge_delays, the functions that actually extract the delays from Femto-VHDL program texts may be defined. It is assumed that any design hierarchies have been flattened out into a sea of concurrent processes. The first function extracts delays from sequential statements. As anticipated, delays are only to be found on signal assignment statements, and are added to the running list by calls to add_one_delay.

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

The second function extracts delays from the collection of parallel processes. Any delays that are found in individual processes through the use of get_delays_ss, are merged with those found on any other processes via calls to merge_delays.

$$env \vdash a \xrightarrow{eval} b$$

which should be read, "a evaluates to b in the environment *env* using the relation $\xrightarrow{eval}$". The environment is the static state of the evaluation (i.e. the current time, the state of the signals, etc.). The turnstile ($\vdash$) separates the environment from the operation of evaluation from a into b.

4.2 Information Organisation

The information needed to make the semantics work is really very simple. The way it is presented here was conceived to correspond to the intuition of VHDL users about the structuring of information in VHDL itself. Much use will be made of set notation in what follows.

Time will be represented by the natural numbers, values by Booleans, and signal names by strings. Whenever there is an event on signal x, its name will be included in a set of events (γ). The state of signals (σ) is given as a set of name-value pairs. Future transactions (τ) is a function from time to σ, and will provide the framework for scheduling. The trace of past behaviour (θ), which is necessary for proper interpretation of attributed signals, is also a function on time, but maps it to a γ-σ pair. The environment in which an individual simulation cycle runs (ρ) holds the current time, σ, γ, and θ. The overall simulation environment (μ) augments ρ with an ordered list of times representing the delays associated with the signal assignments of the Femto-VHDL program in question.

$$\text{time} = N \qquad \text{value} = \text{Boolean} \qquad \text{name} = \text{string}$$
$$\gamma = (\text{name})\text{set} \quad \sigma = (\text{name} \times \text{value})\text{set} \quad \tau = \text{time} \to \sigma \quad \theta = \text{time} \to (\gamma \times \sigma)$$
$$\rho = (\text{time} \times \sigma \times \gamma \times \theta) \qquad \mu = ((\text{time})\text{list} \times \rho)$$

The delays are added to the overall simulation environment to allow the simulation loop to move forward to the next point of computation. Recall that the loop should progress to the nearest point of computation where there are transactions to process. The way that these transactions are created is through the execution of signal assignment statements. So by gathering together all the delays associated with the signal assignments of a given program into an ordered list, we provide a way of knowing the only reachable offsets from the current time.

- $0 \in N$
- if $x \in N$ then $x+1 \in N$

Many other sets also have these properties. Examples are the set of integers (positive and negative), the set of rational numbers and the set of real numbers. However, these properties have special significance for N; N is the *least* set which has both these properties. This means that if X is any other set with these properties then $N \subseteq X$. To prove this one uses mathematical induction to show that $n \in X$ for every natural number n. More precisely, mathematical induction is employed to show the property $P(n)$ is true for every natural number n where $P(x)$ is defined by:

$P(x)$ iff $x \in X$

The base case, $0 \in X$, is trivially true because X satisfies the first property. The fact that X satisfies the second property means that the inductive step is also true: $k \in X$ implies $k+1 \in X$. One can therefore conclude that $n \in X$ for every natural number n, i.e. $N \subseteq X$. In general many sets, relations, etc. will be defined as the least ones satisfying a set of conditions or rules. Each such definition gives rise to a form of induction, called *Rule induction*, for proving properties of the sets, relations, etc. For this reason these are called *inductive definitions*.

4.1.2 Inductive Definitions and Femto-VHDL

The definitions that will be used in the semantics of Femto-VHDL are made up of sets of rules expressed as conditionals:

if **premises** then **conclusion**

Where the premises may take on one of two forms, those associated with the relation being defined and other truths which must be known. These other truths are known as *side conditions*. When writing the rules of the semantics, the following form will be used:

$\frac{\textbf{premises}}{\textbf{conclusion}}$ **side conditions**

Any rule without premises is known as an *axiom*.

The form that each of the premises and the conclusion will take on in the rules is:

CHAPTER 4

DYNAMIC SEMANTICS

The only official specification of the meaning of VHDL is the Language Reference Manual. It describes the semantics of the language in an informal manner through recourse to terms such as "elaboration" and "execution". These are words that implicitly lead to a formulation of meaning in terms of simulation. The material that follows is a formalisation of the semantics of Femto-VHDL that reflects this dynamic intuition. The discussion will begin by presenting a short overview of operational semantics, and will continue with a presentation of how information will be structured in the semantics. The definitions concerning the well-formedness and equivalence of some of those structures will then follow. The chapter will conclude with the definitions and rules that make up the semantics of the language.

4.1 Operational Semantics

The semantics of Femto-VHDL will be given in an operational manner [32]. As alluded to earlier, operational semantics it is a formalism that allows for the specification of and reasoning about dynamic systems. Since VHDL is defined even in the informal context of [21] by means of the simulation process, the idea of using this particular semantic approach is intuitively appealing. In order to define the semantics in an operational way, it will be essential to understand a special form of definition. An example taken from [20] is perhaps the easiest way of coming to grips with what is required.

4.1.1 Inductive Definitions

If one rephrases mathematical induction as a more generally applicable form of induction, the set N of natural numbers satisfies the following conditions (or rules):