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

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