Correct Hardware Compilation with Verilog HDL

Gordon J. Pace

Chalmers University of Technnology, Sweden
gpace@cs.chalmers.se

Abstract. Hardware description languages usually include features which do not have a direct hardware interpretation. Recently, synthesis algorithms allowing some of these features to be compiled into circuits have been developed and implemented. Using a formal semantics of Verilog based on Relational Duration Calculus, we give a number of algebraic laws which Verilog programs obey, using which, we then prove the correctness of a hardware compilation procedure.

1 Introduction

Hardware description languages were originally designed to allow simulation of hardware components to enable the engineer to compare an implementation with the specification in a relatively cheap way. To make this comparison even more efficient, HDLs began to allow procedural modules which described the behaviour in an imperative language style. Unfortunately, these modules were meant for comparing the output of the hardware description without actually having a hardware interpretation from such modules to hardware. Recently, transformations have been implemented in synthesis tools, allowing certain types of behavioural modules to be automatically compiled into hardware.

In [Pac98,PH98] we have defined the semantics of Verilog HDL [Ope93,IEE95], a commercial HDL, widely used in industry. As one of the benefits of this formalisation, was the verification of a compilation procedure from a subset of procedural Verilog code to a more hardware oriented subset of the language.

The semantics of Verilog have been specified in terms of Relational Duration Calculus — a temporal logic. They thus emphasise timing issues in the language. This is the main difference from other similar work, which tend to emphasise the event aspect of the language. Mainly because of this reason, the compilation procedure is not based on any of the ones used by commercial synthesis tools but is rather similar to [HJ94,PL91,May90], except that the output is not a circuit, but another program in the same language.

2 The Semantics of Verilog

2.1 Modules

The semantics of Verilog are given in terms of Relational Duration Calculus [PH98]. A more complete presentation of these semantics can also be found in [Pac98]. We assume the reader to be familiar with Duration Calculus [ZHRR91].

We assume that each module `P` has a number of output wires $\mathsf{out}(P)$ to which no other module may write. Also, the assignments to the outputs of a module must take some time. All modules are allowed to read the output variables of other modules, but reading and writing to and from the same global wires at the same time is not permitted to avoid non-determinism.

The assumption that module outputs are disjoint gives us the opportunity to define parallel composition[1] as:

$$[\![\texttt{P} \parallel \texttt{Q}]\!] \stackrel{def}{=} [\![\texttt{P}]\!] \wedge [\![\texttt{Q}]\!]$$

Continuous assignment: `assign v=e` forces `v` to the value of expression `e`:

$$[\![\texttt{assign v=e}]\!] \stackrel{def}{=} \lceil v = e \rceil^*$$

where $\lceil P \rceil^* = \lceil P \rceil \vee \lceil \rceil$. Mutually dependant continuous assignments are not allowed in the language subset for which we define the semantics.

Procedural behaviour: `initial P` behaves like the sequential program `P`:

$$[\![\texttt{initial P}]\!] \stackrel{def}{=} [\![\texttt{P}]\!]_{\mathsf{out}(P)}(\mathsf{Const}(\mathsf{out}(P)))$$

$[\![\texttt{P}]\!]_W(D)$ describes the behaviour of an individual program module `P` whose output wires are given in set W and which will, upon termination, behave as described by the duration formula D. $\mathsf{Const}(W)$ is defined as follows:

$$\mathsf{Const}(W) \stackrel{def}{=} \forall w \in W \cdot \exists b \cdot (\overleftarrow{w} = b) \wedge (\overrightarrow{w} = b) \wedge \lceil w = b \rceil^*$$

2.2 Imperative Programming Statements

Verilog statements can be split into two sets: imperative programming-like constructs which take no simulation time, and timing control instructions, which are closer to hardware concepts and may take simulation time to execute.

Skip: $[\![\texttt{skip}]\!]_W(D) \stackrel{def}{=} D$

Assignments: $[\![\texttt{v=e}]\!]_W(D) \stackrel{def}{=} (\overrightarrow{v} = \overleftarrow{e} \wedge \mathsf{Const}(W - \{v\}) \wedge \lceil \rceil) \overset{W}{;} D$

Conditionals: $[\![\texttt{if b then P else Q}]\!]_W(D) \stackrel{def}{=} [\![\texttt{P}]\!]_W(D) \triangleleft \overleftarrow{b} \triangleright [\![\texttt{Q}]\!]_W(D)$

Sequential composition: $[\![\texttt{P;Q}]\!]_W(D) \stackrel{def}{=} [\![\texttt{P}]\!]_W([\![\texttt{Q}]\!]_W(D))$

Loops: $[\![\texttt{while b do P}]\!]_W(D) \stackrel{def}{=} \mu X \cdot ([\![\texttt{P}]\!]_W(X) \triangleleft \overleftarrow{b} \triangleright D)$

Fork ... join:

$$[\![\texttt{fork P; Q join}]\!]_W \stackrel{def}{=}$$
$$([\![\texttt{P}]\!]_{W_P}(D) \wedge [\![\texttt{Q}]\!]_{W_Q}(\mathsf{Const}(W_Q) \overset{W_Q}{;} D)) \vee$$
$$([\![\texttt{Q}]\!]_{W_Q}(D) \wedge [\![\texttt{P}]\!]_{W_P}(\mathsf{Const}(W_P) \overset{W_P}{;} D))$$

[1] Note that in Verilog, no symbol is used to denote the parallel composition of modules. We use this notation to make the semantics easier to read and follow.

W_P and W_Q must be disjoint.

The semantics of `do while` loops, `forever` loops, `case` statements, etc can be specified in terms of these constructs.

2.3 Timing Control Instructions

Blocking Assignments. Assignments can be delayed by using guards, which block time until a certain condition is satisfied. The assignment `v=guard e` reads the value of expression `e` and assigns it to `v` as soon as the guard is lowered:

$$[\![\texttt{v=guard e}]\!]_W(D) \stackrel{def}{=} \exists \alpha \cdot [\![\alpha\texttt{=e ; guard ; v=}\alpha]\!]_{W \cup \{\alpha\}}(D)$$

The assignment `guard v=e` waits until the guard is lowered, reads the value of expression `e` and assigns it to `v`:

$$[\![\texttt{guard v=e}]\!]_W(D) \stackrel{def}{=} [\![\texttt{guard ; v=e}]\!]_W(D)$$

Guards. Guards control the flow of time by blocking further execution until they are lowered. Two types of guards are treated here: time delay guards and level triggered guards. Other types of guards can be described in a similar manner.

`#n` blocks the execution of a module by `n` time units:

$$\begin{aligned}[\![\texttt{\#n}]\!]_W(D) \stackrel{def}{=} \; & (l < n \wedge \mathsf{Const}(W)) \vee \\ & (l = n \wedge \mathsf{Const}(W)) \; \underset{\wp}{\overset{W}{;}} \; D\end{aligned}$$

`wait v` blocks execution until v carries the value 1.

$$\begin{aligned}[\![\texttt{wait v}]\!]_W(D) \stackrel{def}{=} \; & (\lceil \neg v \rceil^* \wedge \neg \overrightarrow{v} \wedge \mathsf{Const}(W)) \vee \\ & (\lceil \neg v \rceil^* \wedge \overrightarrow{v} \wedge \mathsf{Const}(W)) \; \underset{\wp}{\overset{W}{;}} \; D\end{aligned}$$

Spikes on communication variables are considered to be undesirable behaviour and are not captured by `wait` statements. A syntactic check suffices to ensure that no spikes will appear on a global variable in the system.

2.4 Non-blocking Assignments

The semantics of non-blocking assignment are defined by:

$$[\![\texttt{v<=guard e}]\!]_W(D) \stackrel{def}{=} [\![\texttt{v=guard e}]\!]_{\{v\}}(\mathsf{Const}(v)) \underset{\{v\}}{\overset{\|}{;}} D$$

Since both processes running in parallel can control the variable `v`, they are composed together using a merging parallel composition operator. Whenever a variable is assigned to by both processes it non-deterministically takes one of the values it is assigned to[2].

[2] Note that this is weaker than the simulation semantics of Verilog, which performs non-blocking assignments only once no enabled threads remain. Our method is too non-deterministic — thus it permits us to make only sound judegments with respect to the simulation semantics of Verilog.

It is also necessary to maintain an extra boolean state for every Verilog variable v: α_v, which holds in those time slots when variable v has just been assigned a value.

$$\begin{aligned}
[\![P \underset{\{v\}}{\|} Q]\!]_W(D) &\stackrel{def}{=} \exists v_P, v_Q, \alpha_{v_P}, \alpha_{v_Q} \cdot \\
&\qquad P[v_P, \alpha_{v_P}/v, \alpha_v] \wedge \\
&\qquad Q[v_Q, \alpha_{v_Q}/v, \alpha_v] \wedge \\
&\qquad \mathsf{Join}(v_P, v_Q, v) \\
\mathsf{Join}(v_1, v_2, v) &\stackrel{def}{=} \lceil \alpha_v = \alpha_{v_1} \vee \alpha_{v_2} \rceil^* \wedge \\
&\qquad \left\lceil \begin{array}{l} (\alpha_{v_1} \wedge \neg\alpha_{v_2} \Rightarrow v = v_1) \\ \wedge\ (\alpha_{v_2} \wedge \neg\alpha_{v_1} \Rightarrow v = v_2) \\ \wedge\ (\alpha_{v_1} \wedge \alpha_{v_2} \Rightarrow v = v_1 \vee v = v_2) \\ \wedge\ (\neg\alpha_{v_1} \wedge \neg\alpha_{v_2} \Rightarrow v = 1 \gg v) \end{array} \right\rceil^*
\end{aligned}$$

Where $(P \gg n)(t) = P(t-n)$. Obviously, the state variables α_v need to be maintained by the model. This is done by adding the information that α_v is true immediately after assignments [Pac98].

2.5 Other Issues

To avoid unnecessarily long program descriptions, we will write `while b do P` as `b*P`, `do P while b` as `P*b`, `if b then P else Q` by `P` $\triangleleft$ `b` $\triangleright$ `Q` and `fork P; Q join` as `P || Q`. Overriding the || symbol is justified by the following algebraic law:

$$\texttt{initial } P \| \texttt{ initial } Q = \texttt{initial fork } P\texttt{; } Q \texttt{ join}$$

To avoid problems with programs such as `while true do skip`, we insist that bodies of loops must take time to terminate. A simple syntactic check is sufficient to ensure this. If dur(P) holds, we can prove that P must always takes time to execute. dur(P) is defined as follows:

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{dur}(\texttt{wait v}) & \stackrel{def}{=} & \text{false} & \mathsf{dur}(\texttt{skip}) & \stackrel{def}{=} & \text{false} \\
\mathsf{dur}(\texttt{v=e}) & \stackrel{def}{=} & \text{false} & \mathsf{dur}(\texttt{v<=g e}) & \stackrel{def}{=} & \text{false} \\
\mathsf{dur}(\texttt{v=g e}) & \stackrel{def}{=} & \mathsf{dur}(\texttt{g}) & \mathsf{dur}(\texttt{g v=e}) & \stackrel{def}{=} & \mathsf{dur}(\texttt{g}) \\
\mathsf{dur}(\texttt{\#0}) & \stackrel{def}{=} & \text{false} & \mathsf{dur}(\texttt{\#}(n+1)) & \stackrel{def}{=} & \text{true} \\
\mathsf{dur}(b * P) & \stackrel{def}{=} & \text{false} & \mathsf{dur}(P * b) & \stackrel{def}{=} & \mathsf{dur}(P) \\
\mathsf{dur}(P; Q) & \stackrel{def}{=} & \mathsf{dur}(P) \vee \mathsf{dur}(Q) & \mathsf{dur}(P \| Q) & \stackrel{def}{=} & \mathsf{dur}(P) \vee \mathsf{dur}(Q) \\
\mathsf{dur}(P \triangleleft b \triangleright Q) & \stackrel{def}{=} & \mathsf{dur}(P) \wedge \mathsf{dur}(Q) & & &
\end{array}$$

3 Algebraic Laws

3.1 Notation

The laws given in this section state an equality (or inequality) between pairs of programs. It is obviously important to state what we mean by $P \sqsubseteq Q$ (P is refined by Q). For P and Q to be comparable, they have to share the same

alphabet. The other condition is that for any possible continuation, the P can always exhibit (at least) all behaviours of Q. Equality then follows from this definition. Formally, this may be written as:

$$P \sqsubseteq Q \stackrel{def}{=} [\![Q]\!]_V(D) \Rightarrow [\![P]\!]_V(D)$$
$$P = Q \stackrel{def}{=} P \sqsubseteq Q \ \wedge \ Q \sqsubseteq P$$

where D can range over all valid relational duration formulae and V is the alphabet of P and Q.

3.2 Monotonicity

The first laws state that the programming constructs in Verilog are monotonic — if we selectively refine portions of the program, we are guaranteed a refinement of the whole program. If we have a program context C, then we can guarantee that

$$\text{if } P \sqsubseteq Q \text{ then } C(P) \sqsubseteq C(Q)$$
$$\text{if } P = Q \text{ then } C(P) = C(Q)$$

To guarantee monotonicity, programs may not use non-blocking assignments. This constraint is somewhat weakened later in the paper.

3.3 Parallel and Sequential Composition

Sequential composition is associative:

$$P; (Q; R) = (P; Q); R$$

Parallel composition is commutative and associative. Also, if $\mathsf{dur}(P)$ holds, then `#1` is a unit of parallel composition:

$$P \parallel Q = Q \parallel P$$
$$P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$$
$$P \parallel \texttt{\#1} = P$$

In fact `#1` distributes in and out of parallel composition:

$$\texttt{\#1}; P \parallel \texttt{\#1}; Q = \texttt{\#1}; (P \parallel Q)$$

3.4 Non-determinism and Assumptions

It will be found useful to introduce new Verilog constructs. These will feature in our proofs but will eventually be removed to reduce the program back to a standard Verilog program.

One useful construct is non-deterministic composition. The non-deterministic composition of two programs can behave as either of the two. More formally, we define it as:

$$[\![P \sqcap Q]\!]_W(D) \stackrel{def}{=} [\![P]\!]_W(D) \vee [\![Q]\!]_W(D)$$

From this definition it immediately follows that non-determinism is commutative, associative, idempotent and monotonic.

Non-determinism also distributes over sequential composition:

$$P;(Q \sqcap R) = (P;Q) \ \sqcap \ (P;R)$$
$$(P \sqcap Q);R = (P;R) \ \sqcap \ (Q;R)$$

Another useful statement is the assumption. The statement *assume* b, expressed as $b^\top$, claims that expression b has to be true at that point in the program:

$$[\![b^\top]\!]_W(D) \stackrel{def}{=} (\lceil\rceil \vee \lceil b\rceil; \mathbf{true}) \wedge D$$

Conjunction of two conditions results in sequential composition of the conditions. Two corollaries of this are commutativity and idempotency of assumptions with respect to sequential composition:

$$(b \wedge c)^\top = b^\top; c^\top$$
$$b^\top = b^\top; b^\top$$
$$b^\top; c^\top = c^\top; b^\top$$

Disjunction of two conditions acts like non-determinism:

$$(b \vee c)^\top; P = (b^\top; P) \sqcap (c^\top; P)$$

Assumptions make a program more deterministic:

$$P \sqsubseteq b^\top; P$$

3.5 Conditional

Sequential, parallel and non-deterministic composition distributes out of conditionals:

$$(P \lhd b \rhd Q);R = (P;R) \lhd b \rhd (Q;R)$$
$$P \sqcap (Q \lhd b \rhd R) = (P \sqcap Q) \lhd b \rhd (P \sqcap R)$$
$$P \parallel (Q \lhd b \rhd R) = (P \parallel Q) \lhd b \rhd (P \parallel R)$$

Provided that the values of the variables in expression b are not changed immediately by programs P and Q, a conditional can be expressed in terms of non-determinism and assumptions:

$$P \lhd b \rhd Q = (b^\top; P) \sqcap (\neg b^\top; Q)$$

The precondition is satisfied if P has a prefix P_1 such that $\mathsf{dur}(P_1)$ and the variables of b are not assigned in P_1. Similarly for Q.

3.6 Loops

The recursive nature of loops can be expressed quite succinctly in terms of algebraic laws:

Unique least fixed point: $Q \;=\; (P;Q) \triangleleft b \triangleright \texttt{skip}$ if and only if $Q = b * P$.

As immediate corollaries of this law, we have:

$$(b * P) \triangleleft b \triangleright \texttt{skip} \;=\; b * P$$

$Q = P;Q$ if and only if $Q = \texttt{forever}\ P$

$Q = P;(Q \triangleleft b \triangleright \texttt{skip})$ if and only if $Q = P * b$

The first law can be strengthened to:

$$Q = (P;Q) \triangleleft b \triangleright R \text{ if and only if } Q = (b * P);R.$$

The following law allows us to move part of a loop body outside:

$$\text{If } Q;P;Q \;=\; Q;Q \text{ then } (P;Q) * b \;=\; P;(Q * b)$$

3.7 Continuous Assignments

For convenient presentation of certain properties, we will allow programs of the form P ; `assign v=e`. This will act like:

$$[\![P]\!]_W([\![\texttt{assign v=e}]\!] \wedge \mathsf{Const}(W - \{v\}))$$

Similarly, P ; `(assign v=e` $\|$ Q`)` acts like:

$$[\![P]\!]_W([\![\texttt{assign v=e} \parallel \texttt{initial}\ Q]\!])$$

3.8 Communication

The use of synchronisation signals can greatly increase the readability of code. These 'channels' can be implemented as global variables which normally carry a value of zero, but briefly go up to one when a signal is sent over them. 'Wait for signal s', $s?$, is thus easily implemented as:

$$s? \stackrel{def}{=} \texttt{wait}\ s$$

To send a signal, without blocking the rest of the program for any measurable simulation time but leaving the signal on for a non-zero time measure, the non-blocking assignment statement can be rather handy:

$$s! \stackrel{def}{=} s\texttt{=1}\ ;\ s\texttt{<=}\ \#\delta\ 0$$

What value of δ is to be used? Obviously, δ has to be larger than zero. However, taking a value of 1 or larger leads to a conflict in the program: $s!\ ;\ \#\delta\ ;\ s!$

The solution is to use a value of 0.5 for δ. This does not invalidate previous reasoning based on discrete time. Effectively, what we have done is to reduce the size of the smallest time step to a level which normal Verilog programs will not have direct access to.

If non-blocking assignments are only used for signals, and signals are accessed only using $s!$ and $f(s)?$, we can guarantee the monotonicity of any program context despite the presence of non-blocking assignments.

3.9 Algebraic Laws for Communication

Signal Output We will say that $s! \in P$ is command $s!$ occurs somewhere in program P. Similarly, $s! \notin P$ means that $s!$ does not occur anywhere in program P. In both cases, we assume that s is a signal in the output alphabet of P.

Signals start off as false, provided that they are not initially written to. Also, between any two time consuming programs which do not output on the signal, the signal is false. If $\mathsf{dur}(P)$, $\mathsf{dur}(Q)$ and $s! \notin P$, $s! \notin Q$:

$$\texttt{initial}\ P; R = \texttt{initial}\ \neg s^\top; P; R$$
$$P; Q = P; \neg s^\top; Q$$

$s!$ sets signal s to true:

$$s! = s!; s^\top$$

Output on a signal can be moved out of parallel composition:

$$(s!; P) \parallel Q = s!; (P \parallel Q)$$

Signalling and assumptions commute:

$$s!; b^\top = b^\top; s!$$

Wait on Signal Waiting stops once the condition is satisfied:

$$(s^\top; P) \parallel (s?; Q) = (s^\top; P) \parallel Q$$

Execution continues in parallel components until the condition is satisfied. If no signal is sent during the initial part of the program, we can afford an extra time unit. Provided that $s! \notin P$:

$$(\neg s^\top; P; Q) \parallel (s?; R) = \neg s^\top; P; (Q \parallel s?; R)$$
$$(\neg s^\top; P; Q) \parallel (\texttt{\#1}; s?; R) = \neg s^\top; P; (Q \parallel s?; R) \qquad \text{provided that } \mathsf{dur}(P)$$

Furthermore, if P is the process controlling a signal s, and s starts off with a value 0, the value of $f(s)$ will remain that of $f(0)$ until P takes over. If $s \in \mathsf{out}(P)$:

$$(s\texttt{=}0)^\top; f(s)?; P = (s\texttt{=}0)^\top; f(0)?; P$$

Continuous Assignment Signals It will be found useful write to some signals using continuous assignments. The laws given to handle signal reading and writing no longer apply directly to these signals and hence need modification.

The following laws thus allow algebraic reasoning about a signal s written to by a continuous assignment of the form:

$$\texttt{assign}\ s = f(s_1, \ldots, s_n)$$

where all variables s_1 to s_n are signals. For s to behave like a normal signal (normally zero except for half unit long phases with value one), $f(0, 0, \ldots 0)$

has to take value 0. These laws allow us to transform a signal assigned to by a continuous assignment to one controlled by a sequential program (or vice-versa):

If $b \Rightarrow f(\overline{s})$ then:

$$\texttt{assign } s\texttt{=}f(\overline{s}) \parallel b^{\top}; P = b^{\top}; s!; (\texttt{assign } s\texttt{=}f(\overline{s}) \parallel P)$$

If $b \Rightarrow \neg f(\overline{s})$ and for all i, $s_i \notin P$ then:

$$\texttt{assign } s\texttt{=}f(\overline{s}) \parallel b^{\top}; P; Q = b^{\top}; P; (\texttt{assign } s\texttt{=}f(\overline{s}) \parallel Q)$$

Furthermore, instances of a signal controlled by a continuous assignment can be removed using the following law:

$$\texttt{assign } s\texttt{=}f(\overline{s}) \parallel P(s?) = \texttt{assign } s\texttt{=}f(\overline{s}) \parallel P(f(\overline{s})?)$$

3.10 Signals and Merge

Signals now allow us to transform a process to use a different set of variables. The trick used is to define a merging process which merges the assignments on a number of variables to a new variable.

The first thing to do is to make sure that we know whenever a variable has been assigned to. The technique we use is simply to send a signal on α_v to make it known that v has just been assigned a value. Note that we will only allow this procedure to be used on global variables, which guarantees that no more than one assignment on a particular variable can take place at the same time.

We thus replace all assignments: `v=g e` by (`v=g e;` $\alpha_v!$) and similarly `g v=e` by (`g v=e;` $\alpha_v!$)

The program Merge will collapse the variables v_1 and v_2 into a single variable v, provided that they are never assigned to at the same time:

$$\begin{aligned} \mathsf{Merge} \stackrel{def}{=} \quad & \texttt{assign } \alpha_v\texttt{=}\alpha_{v_1} \vee \alpha_{v_2} \\ \parallel & \texttt{assign } v\texttt{=}(v_1 \triangleleft \alpha_{v_1} \triangleright v_2) \triangleleft \alpha_v \triangleright v^{-} \\ \parallel & \texttt{assign } v^{-}\texttt{=\#0.5 } v \end{aligned}$$

The following laws relate Merge with parallel composition and conditional:

$$P; Q \sqsubseteq (P[v_1/v]; Q[v_2/v]) \parallel \mathsf{Merge}$$
$$P \triangleleft b \triangleright Q \sqsubseteq (P[v_1/v] \triangleleft b \triangleright Q[v_2/v]) \parallel \mathsf{Merge}$$

Note that these laws can be strengthened to equalities if we hide the extra variables appearing on the right hand side.

4 Triggered Imperative Programs

We will not be compiling just any program, but only ones which are triggered by a start signal and upon termination issue a finish signal. The environment is assumed not to interfere with the program by issuing a further start signal before the program has terminated.

Triggered programs can be constructed from general imperative programs:

$$\psi_s^f(P) \stackrel{def}{=} \texttt{forever}\,(s?;\ P;\ f!)$$

At the topmost level, these programs also ensure that the termination signal is initialised to zero:

$$i\psi_s^f(P) \stackrel{def}{=} \texttt{initial}\ \ f = 0;\ \psi_s^f(P)$$

The environment constraint for two signals s and f may now be easily expressed:

$$\epsilon_s^f \sqsupseteq \texttt{forever}\ (\ s!;\ \texttt{\#1};\ f?; \Delta_0)$$

where Δ_0 is a statement which, once triggered, waits for an arbitrary length of time (possibly zero or infinite) before allowing execution to resume.

$$[\![\Delta_0]\!]_W(D) \stackrel{def}{=} \mathsf{Const}(W) \vee \mathsf{Const}(W) \overset{W}{\circ\!\!\circ} D$$

The unit delay ensures that if a start signal is sent immediately upon receiving a finish signal, we do not interpret the old finish signal as another finish signal. Now, we ensure that if we start off with a non-zero delay, the start signal is initially off:

$$i\epsilon_s^f \sqsupseteq ((\neg s^\top; \Delta) \sqcap \texttt{skip})\ ;\ \epsilon_s^f$$

Δ is a statement almost identical to Δ_0 but which, once triggered, waits for a non-zero arbitrary length of time (possibly infinite) before allowing execution to resume.

$$[\![\Delta]\!]_W(D) \stackrel{def}{=} \mathsf{Const}(W) \vee (l > 0 \wedge \mathsf{Const}(W)) \overset{W}{\circ\!\!\circ} D$$

Δ obeys a number of laws which we will find useful later:

$$\Delta_0 = \texttt{skip}\ \sqcap\ \Delta$$

$$\text{If } \mathsf{dur}(P) \text{ then } \Delta \sqsubseteq P$$

If s is a signal then its behaviour is a refinement of:

$$(\texttt{skip} \sqcap \neg s^\top; \Delta); \texttt{forever}\ s!; \Delta$$

The results which follow usually state a refinement which holds if a particular environment condition holds. Hence, these are of the form:

$$\text{environment condition} \Rightarrow (P \Rightarrow Q)$$

To avoid confusion with nested implications, we define the conditional refinement $i\epsilon_s^f \vdash P \sqsubseteq Q$ as follows:

$$i\epsilon_s^f \vdash P \sqsubseteq Q \stackrel{def}{=} \vdash i\epsilon_s^f \Rightarrow (Q \Rightarrow P)$$

$$i\epsilon_s^f \vdash P = Q \stackrel{def}{=} (i\epsilon_s^f \vdash P \sqsubseteq Q) \wedge (i\epsilon_s^f \vdash Q \sqsubseteq P)$$

The following proofs assume that all programs (and sub-programs) satisfy $\mathsf{dur}(P)$ (hence programs take time to execute). We also add the constraint that programs do not read or write as soon as they a executed ($P = Q; R$ such that Q does not read or write data). Note that if all primitive programs we use satisfy these conditions, so do programs constructed using sequential composition, conditionals, `fork join` and `do while` loops.

5 Hardware Compilation

5.1 Basic Results

We start by establishing ways of decomposing our programs into a number of smaller ones running in parallel. The proofs of the following three theorems can be found in the appendix.

Theorem 1.1: Sequential composition can be thus decomposed:

$$i\epsilon_s^f \vdash i\psi_s^f(P;Q) \sqsubseteq i\psi_s^m(P') \parallel i\psi_m^f(Q') \parallel \mathsf{Merge}$$

where for any program P, we will use P' to represent $P[v_P/v]$. Merge has been defined in section 3.10.

Theorem 1.2: Conditional statements can be thus decomposed:

$$\epsilon_s^f \vdash \psi_s^f(P \lhd b \rhd Q) \sqsubseteq \psi_{s_P}^{f_P}(P') \parallel \psi_{s_Q}^{f_Q}(Q') \parallel \mathsf{Merge} \parallel \mathcal{I}\mathrm{nterface}$$

where

$$\begin{aligned}\mathcal{I}\mathrm{nterface} = {} & \mathtt{assign}\ s_P = s \wedge b \quad \parallel \\ & \mathtt{assign}\ s_Q = s \wedge \neg b \ \parallel \\ & \mathtt{assign}\ f = f_P \vee f_Q\end{aligned}$$

Theorem 1.3: Loops can be decomposed into their constituent parts.

$$\epsilon_s^f \vdash \psi_s^f(P * b) \sqsubseteq \psi_{s_P}^{f_P}(P) \parallel \mathcal{I}\mathrm{nterface}$$

where

$$\begin{aligned}\mathcal{I}\mathrm{nterface} = {} & \mathtt{assign}\ s_P = s \vee (f_P \wedge b) \parallel \\ & \mathtt{assign}\ f = f_P \wedge \neg b\end{aligned}$$

5.2 Compilation

Using these refinements, we can now define a compilation process:

$$\begin{aligned}\Psi_s^f(P;Q) &\stackrel{def}{=} \Psi_s^m(P') \parallel \Psi_m^f(Q') \parallel \mathsf{Merge} \\ \Psi_s^f(P \lhd b \rhd Q) &\stackrel{def}{=} \Psi_{s_P}^{f_P}(P') \parallel \Psi_{s_Q}^{f_Q}(Q') \parallel \mathsf{Merge} \parallel \mathcal{I}\mathrm{nterface}_C \\ \Psi_s^f(P * b) &\stackrel{def}{=} \Psi_{s_P}^{f_P}(P) \parallel \mathcal{I}\mathrm{nterface}_L \\ \Psi_s^f(P) &\stackrel{def}{=} \psi_s^f(P) \text{ otherwise}\end{aligned}$$

We know that the individual steps of the compilation process are correct. However, it is not yet clear whether the topmost environment condition is sufficient to show that the compiled program is a refinement of the topmost program.

In fact, we prove that a stronger invariant holds thoughout the compilation process. This invariant is that for any start signal s and related finish signal f:

$$\int f \leq \int s \gg 1 \text{ and } \int f \leq \int s \leq \int f + 0.5$$

We start by establishing that this invariant is sufficient to guarantee the environment condition (lemma 2.1). Furthermore, $i\epsilon_s^f$ and $i\psi_s^f(P)$ guarantee the invariant (lemmata 2.2, 2.3).

Lemma 2.1: Provided that s and f are signals, if $\int f \leq \int s \gg 1$ and $\int f \leq \int s \leq \int f + 0.5$ are valid duration formulae, then so is $i\epsilon_s^f$.

Proof: The proof is given in the appendix.

Lemma 2.2: $i\epsilon_s^f \Rightarrow \int s \leq \int f + 0.5$.

Proof: The proof is given in the appendix.

Lemma 2.3: Provided that $\mathsf{dur}(P)$:

$$i\psi_s^f(P) \Rightarrow \int f \leq \int s \ \wedge \ \int f \leq \int s \gg 1$$

Proof: The proof is given in the appendix.

Using these results, we can now show that the invariant is guaranteed by the compilation process.

Lemma 2.4: The environment conditions follow along the compilation process:

$$\Psi_s^f(P) \Rightarrow \int f \leq \int s$$

Proof: The proof can be found in the appendix.

Lemma 2.5: $\Psi_s^f(P) \Rightarrow \int f \leq \int s \gg 1$

Proof: The proof follows almost identically to that of lemma 2.4 (see appendix).

5.3 Compiler Correctness

Theorem 2: If s is a signal, then:

$$\int s \leq \int f + 0.5 \vdash \psi_s^f(P) \sqsubseteq \Psi_s^f(P)$$

Proof: Assume that $\int s \leq \int f + 0.5$.

$\Psi_s^f(P)$ guarantees that $\int f \leq \int s$ (lemma 2.4) and that $\int f \leq \int s \gg 1$ (lemma 2.5).

Hence, by lemma 2.1, we know that $i\epsilon_s^f$.

The proof now follows by induction on the structure of the program P.

In the base case, when P is a simple program, $\Psi_s^f(P)$ is just $\psi_s^f(P)$, and hence trivially guarantees correctness.

For the inductive case we consider the different possibilities:

Sequential composition: We need to prove that $\int s \leq \int f + 0.5 \vdash \psi_s^f(Q;R) \sqsubseteq \Psi_s^f(Q;R)$

But, by definition of Ψ, and the further application of lemma 2.4:

$$\Psi_s^m(Q') \| \Psi_m^f(R') \ \wedge \ \int m \leq \int s \ \wedge \ \int f \leq \int m$$

Hence, combining the above inequalities with the previous ones:

$$\int m \leq \int f + 0.5 \ \ \wedge \ \ \int s \leq \int m + 0.5$$

By the inductive hypothesis, we thus conclude that:

$$\psi_s^m(Q') \parallel \psi_m^f(R')$$

But we also know that $i\epsilon_s^f$. Thus we can apply theorem 1.1 to conclude that $\psi_s^f(Q;R)$.

Therefore, $\int s \leq \int f + 0.5 \vdash \psi_s^f(P;Q) \sqsubseteq \Psi_s^f(P;Q)$.

Conditional: We need to prove that:

$$\int s \leq \int f + 0.5 \vdash \psi_s^f(Q \lhd b \rhd R) \sqsubseteq \Psi_s^f(Q \lhd b \rhd R)$$

As before, we know that: $\int f \leq \int s \leq \int f + 0.5$.

Also, by definition of Ψ and lemma 2.4:

$$\begin{array}{rcl} \Psi_{s_Q}^{f_Q}(Q) \parallel \Psi_{s_R}^{f_R}(R) \parallel \mathcal{I}\text{nterface}_C & & \\ \int f_R & \leq & \int s_Q \\ \int f_R & \leq & \int s_R \end{array}$$

Using simple duration calculus arguments on the interface part, we can conclude that:

$$\begin{array}{l} \int s = \int s_Q + \int s_R \\ \int f = \int f_Q + \int f_R - \int(f_Q \wedge f_R) \end{array}$$

Hence:

$$\begin{array}{l} \int s \leq \int f + 0.5 \\ \Rightarrow \int s_Q + \int s_R \leq \int f_Q + \int f_R + 0.5 - \int(f_Q \wedge f_R) \\ \Rightarrow \int s_Q \leq \int f_Q + 0.5 - (\int s_R - \int f_R) - \int(f_Q \wedge f_R) \\ \Rightarrow \int s_Q \leq \int f_Q + 0.5 \end{array}$$

The last step is justified since $\int s_R \geq \int f_R$.

The same argument can be used to show that $\int s_R \leq \int f_R + 0.5$. Hence, we can use the inductive hypothesis to conclude that:

$$\psi_{s_Q}^{f_Q}(Q) \parallel \psi_{s_R}^{f_R}(R) \parallel \mathcal{I}\text{nterface}_C$$

But, since $i\epsilon_s^f$ holds, we can use theorem 1.2 to conclude that:
$\int s \leq \int f + 0.5 \vdash \psi_s^f(Q \lhd b \rhd R) \sqsubseteq \Psi_s^f(Q \lhd b \rhd R)$.

Loops: Finally, we need to prove that $\int s \leq \int f + 0.5 \vdash \psi_s^f(Q * b) \sqsubseteq \Psi_s^f(Q * b)$.

The argument is almost identical to the one given for the conditional statement, except that the equality we need to derive from the interface so as to enable us to complete the proof is that:

$$\int s_P = \int s + \int f_P - \int f - \int s \wedge f_P \wedge b$$

Hence, by induction, we can conclude that $\int s \leq \int f + 0.5 \vdash \psi_s^f(P) \sqsubseteq \Psi_s^f(P)$.

□

Corollary: $i\epsilon_s^f \vdash i\psi_s^f(P) \sqsubseteq \Psi_s^f(P)$.

Proof: Follows immediately from lemma 2.2 and theorem 2.

□

5.4 Basic Instructions

Implementation of a number of basic instructions in terms of continuous assignments can also be easily done. Consider, for example:

$$\begin{aligned} \Psi_s^f(\texttt{\#1}) &\stackrel{def}{=} \texttt{assign } f = \texttt{\#0.5 } m \\ &\parallel \texttt{assign } m = \texttt{\#0.5 } s \\ \Psi_s^f(\texttt{\#1 } v = e) &\stackrel{def}{=} \Psi_s^f(\texttt{\#1}) \\ &\parallel \texttt{assign } v^- = \texttt{\#0.5 } v \\ &\parallel \texttt{assign } v = e \triangleleft f \triangleright v^- \end{aligned}$$

For a definition $\Psi_s^f(P) \stackrel{def}{=} Q$, it is enough to verify that $i\epsilon_s^f \vdash \psi_s^f(P) \sqsubseteq Q$. The result of the corollary can then be extended to cater for these compilation rules. Hence, these laws can be verified, allowing a *total* compilation of a program written in terms of these instructions and the given constructs into continuous assignments.

5.5 Single Runs

Finally, what if we are interested in running a compiled program just once? It is easy to see the $i\epsilon_s^f$ inequality is satisfied by `initial` $s!$. Also, we can prove that:

$$\texttt{initial } s! \parallel \psi_s^f(P) = \texttt{initial } s!; P; f!$$

Hence, for a single run of the program, we simply add an environment satisfying the desired property: `initial` $s!$.

6 Comparisons and Conclusions

Most published Verilog and VHDL formal semantics are operational in style, mainly because this complements the event based nature of their simulation cycle, which is used to informally define the semantics of the language in official documentation. [KB95,Bor95] give a rather comprehensive (if dated) overview of the work done in formalising VHDL semantics. The need for the formalisation of Verilog semantics was advocated in [Gor95], since when a number of semantics have been published [GG98,SX98,Sas99,FLS99].

This paper applies to Verilog a number of techniques already established in the hardware compilation community [KW88,May90], giving us a number of compilation rules which translate a sequential program into a parallel one. Most of the proof steps involve a number of applications of the laws of Verilog, and would thus benefit from machine verification.

One interesting result of the approach applied here is the separation placed between the control and data paths, which is clearly visible from the compilation procedure.

The method used here is very similar to the compilation procedure used with Occam in [May90] and Handel in [HJ94,PL91]. The transformation depends heavily on the timing constraints — unlike the approach usually taken by commercial synthesis tools which usually synchronise using global clock and reset signals [Pal96]. The main difference between the compilation of Verilog programs we define with that of Occam or Handel is the fact that timing control can be explicitly expressed in Verilog. It is thus not acceptable to assume that immediate assignments take a whole time unit to execute (as is done in the case of Occam and Handel). It was however necessary to impose the constraint that all compiled programs take some time to execute. This limitation obviously allows us to compile only a subset of Verilog programs. However, clever use of algebraic laws can allow the designer to modify code so as to enable compilation. How much of this can be done automatically and efficiently by the compiler itself is still an open question.

References

[Bor95] Editor Dominique Borrione. Formal methods in system design, special issue on VHDL semantics. Volume 7, Nos. 1/2, Aug 1995.

[FLS99] John Fiskio-Lasseter and Amr Sabry. Putting operational techniques to the test: A syntactic theory for behavioural verilog. In *The Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS'99)*, 1999.

[GG98] M.J.C. Gordon and A. Ghosh. Language independent RTL semantics. In *Proceedings of IEEE CS Annual Workshop on VLSI: System Level Design, Florida, USA*, 1998.

[Gor95] Mike Gordon. The semantic challenge of Verilog HDL. In *Proceedings of the tenth annual IEEE symposium on Logic in Computer Science (LICS '95) San Diego, California*, pages 136–145, June 1995.

[HJ94] Jifeng He and Zheng Jianping. Simulation approach to provably correct hardware compilation. In *Formal Techniques in Real-Time and Fault Tolerant Systems*, number 863 in Lecture Notes in Computer Science, pages 336–350. Springer-Verlag, 1994.

[IEE95] IEEE. *Draft Standard Verilog HDL (IEEE 1364)*. 1995.

[KB95] Carlo Delgado Kloos and Peter T. Breuer. *Formal Semantics for VHDL*. Number 307 in The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1995.

[KW88] K. Keutzer and W. Wolf. Anatomy of a hardware compiler. In David S. Wise, editor, *Proceedings of the SIGPLAN '88 Conference on Programming Lanugage Design and Implementation (SIGPLAN '88)*, pages 95–104. ACM Press, June 1988.

[May90] D. May. Compiling occam into silicon. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, University of Texas at Austin Year of Programming Series, chapter 3, pages 87–106. Addison-Wesley Publishing Company, 1990.

[Ope93] Open Verilog International. *Verilog Hardware Description Language Reference Manual (Version 2.0)*. Open Verilog, March 1993.

[Pac98] Gordon J. Pace. *Hardware Design Based on Verilog HDL.* PhD thesis, Computing Laboratory, University of Oxford, 1998.

[Pal96] Samir Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis.* Prentice Hall, New York, 1996.

[PH98] Gordon J. Pace and Jifeng He. Formal reasoning with Verilog HDL. In *Proceedings of the Workshop on Formal Techniques in Hardware and Hardware-like Systems, Marstrand, Sweden*, June 1998.

[PL91] Ian Page and Wayne Luk. Compiling occam into field-programmable gate arrays. In Wayne Luk and Will Moore, editors, *FPGAs*, pages 271–283. Abingdon EE&CS books, 1991.

[Sas99] H. Sasaki. A Formal Semantics for Verilog-VHDL Simulation Interoperability by Abstract State Machine. In *Proceedings of DATE'99 (Design, Automation and Test in Europe), ICM Munich, Germany*, March 1999.

[SX98] G. Schneider and Q. Xu. Towards a formal semantics of verilog using duration calculus. *Lecture Notes in Computer Science*, 1486:282–??, 1998.

[ZHRR91] Chaochen Zhou, Michael R. Hansen, Anders Ravn, and Hans Rischel. Duration specifications for shared processors. In J. Vytopil, editor, *Formal Techniques in Real Time and Fault Tolerant Systems*, number 571 in Lecture Notes in Computer Science, pages 21–32. Springer-Verlag, 1991.

A Compilation Theorems

Theorem 1.1: Sequential composition can be thus decomposed:

$$i\epsilon_s^f \vdash i\psi_s^f(P;Q) \sqsubseteq i\psi_s^m(P') \parallel i\psi_m^f(Q') \parallel \mathsf{Merge}$$

where for any program P, we will use P' to represent $P[v_P/v]$. Merge has been defined in section 3.10.

Proof: First note the following result:

$$\begin{array}{ll} & i\epsilon_s^f \parallel \psi_s^f(P;Q) \\ = & \{ \text{ communication laws } \} \\ & (\neg s^\top; \Delta \sqcap \texttt{skip}); s!; P; Q; f!; \Delta_0; (\epsilon_s^f \parallel \psi_s^f(P;Q)) \\ = & \{ \text{ by law of } \Delta_0 \ \} \\ & (\neg s^\top; \Delta \sqcap \texttt{skip}); s!; P; Q; f!; (\neg s^\top; \Delta \sqcap \texttt{skip}); (\epsilon_s^f \parallel \psi_s^f(P;Q)) \\ = & \{ \text{ by definition of } i\epsilon \ \} \\ & (\neg s^\top; \Delta \sqcap \texttt{skip}); s!; P; Q; f!; (i\epsilon_s^f \parallel \psi_s^f(P;Q)) \\ = & \{ \text{ definition of } \texttt{forever} \ \} \\ & \texttt{forever } (\neg s^\top; \Delta \sqcap \texttt{skip}); s!; P; Q; f! \\ \sqsubseteq & \{ \text{ new signal introduction and laws of signals } \} \\ & \texttt{forever } \neg m^\top; (\neg s^\top; \Delta \sqcap \texttt{skip}); s!; P; m!; Q; f! \end{array}$$

Now consider the other side of the refinement:

$$
\begin{array}{ll}
& i\epsilon_s^f \parallel i\psi_s^m(P') \parallel \psi_m^f(Q') \\
= & \{ \text{ communication laws } \} \\
& \neg m^\top ; (\neg s^\top ; \Delta \sqcap \texttt{skip}); P'; m!; Q'; f!; \neg m^\top ; \Delta_0 ; (\epsilon_s^f \parallel \psi_s^m(P') \parallel \psi_m^f(Q')) \\
= & \{ \text{ definition of } i\epsilon, \, i\psi \text{ and law of } \Delta_0 \, \} \\
& \neg m^\top ; (\neg s^\top ; \Delta \sqcap \texttt{skip}); P'; m!; Q'; f!; (i\epsilon_s^f \parallel i\psi_s^m(P') \parallel \psi_m^f(Q')) \\
= & \{ \text{ definition of } \texttt{forever} \, \} \\
& \texttt{forever } \neg m^\top ; (\neg s^\top ; \Delta \sqcap \texttt{skip}); P'; m!; Q'; f!
\end{array}
$$

We can now prove the desired refinement:

$$
\begin{array}{ll}
& i\epsilon_s^f \parallel i\psi_s^f(P;Q) \\
\sqsubseteq & \{ \text{ definition of } i\psi \text{ and proved inequality } \} \\
& \neg f^\top ; \texttt{forever } \neg m^\top ; (\neg s^\top ; \Delta \sqcap \texttt{skip}); s!; P; m!; Q; f! \\
= & \{ \text{ laws of merge from section 3.10 } \} \\
& \mathsf{Merge} \parallel (\neg f^\top ; \texttt{forever } \neg m^\top ; (\neg s^\top ; \Delta \sqcap \texttt{skip}); s!; P'; m!; Q'; f!) \\
= & \{ \text{ above claim } \} \\
& \mathsf{Merge} \parallel (\neg f^\top ; (i\epsilon_s^f \parallel i\psi_s^m(P') \parallel \psi_m^f(Q')) \\
= & \{ \text{ definition of } i\psi \text{ and associativity of } \parallel \} \\
& \mathsf{Merge} \parallel i\epsilon_s^f \parallel i\psi_s^m(P') \parallel i\psi_m^f(Q')
\end{array}
$$

□

Theorem 1.2: Conditional statements can be thus decomposed:

$$\epsilon_s^f \vdash \psi_s^f(P \lhd b \rhd Q) \sqsubseteq \psi_{s_P}^{f_P}(P') \parallel \psi_{s_Q}^{f_Q}(Q') \parallel \mathsf{Merge} \parallel \mathcal{I}\text{nterface}$$

where

$$
\begin{array}{rl}
\mathcal{I}\text{nterface} = & \texttt{assign } s_P = s \wedge b \quad \parallel \\
& \texttt{assign } s_Q = s \wedge \neg b \; \parallel \\
& \texttt{assign } f = f_P \vee f_Q
\end{array}
$$

Proof: The proof is similar to that of Theorem 1.1.

Theorem 1.3: Loops can be decomposed into their constituent parts.

$$\epsilon_s^f \vdash \psi_s^f(P * b) \sqsubseteq \psi_{s_P}^{f_P}(P) \parallel \mathcal{I}\text{nterface}$$

where

$$
\begin{array}{rl}
\mathcal{I}\text{nterface} = & \texttt{assign } s_P = s \vee (f_P \wedge b) \parallel \\
& \texttt{assign } f = f_P \wedge \neg b
\end{array}
$$

Proof: Again, the proof is similar to that of Theorem 1.1.

Complete proofs of Theorems 1.2 and 1.3 can be found in [Pac98].

B Envrionment Theorems

Lemma 2.1: Provided that s and f are signals, if $\int f \leq \int s \gg 1$ and $\int f \leq \int s \leq \int f + 0.5$ are valid duration formula, then so is $i\epsilon_s^f$.

Proof: Since the inequality holds for all prefix time intervals, and s and f are both signals, we can use duration calculus reasoning to conclude that:

$$\Box((\lceil s \rceil \wedge l = 0.5); \mathbf{true}; (\lceil s \rceil \wedge l = 0.5) \Rightarrow l = 1; \mathbf{true}; \lceil f \rceil; \mathbf{true})$$

This allows us to deduce that $s!; \Delta; s!; \Delta \Rightarrow s!; \mathtt{\#1}; f?; \Delta_0; s!; \Delta$.

But s is a signal, and hence satisfies $(\neg s^\top; \Delta \sqcap \mathtt{skip}); \mathtt{forever}\ s!; \Delta$.

$$\begin{aligned}
& \mathtt{forever}\ s!; \Delta \\
= & \{ \text{ definition of } \mathtt{forever} \text{ loops } \} \\
& s!; \Delta; s!; \Delta; \mathtt{forever}\ s!; \Delta \\
\Rightarrow & \{ \text{ by implication just given } \} \\
& s!; \mathtt{\#1}; f?; \Delta_0; s!; \Delta; \mathtt{forever}\ s!; \Delta \\
= & \{ \text{ definition of } \mathtt{forever} \text{ loops } \} \\
& s!; \mathtt{\#1}; f?; \Delta_0; \mathtt{forever}\ s!; \Delta \\
\Rightarrow & \{ \text{ definition of } \mathtt{forever} \text{ loops } \} \\
& \mathtt{forever}\ s!; \mathtt{\#1}; f?; \Delta_0
\end{aligned}$$

Hence, from the fact that s is a signal, we can conclude the desired result:

$$\begin{aligned}
& (\neg s^\top; \Delta \sqcap \mathtt{skip}); \mathtt{forever}\ s!; \Delta \\
\Rightarrow & (\neg s^\top; \Delta \sqcap \mathtt{skip}); \mathtt{forever}\ s!; \mathtt{\#1}; f?; \Delta_0 \\
= & i\epsilon_s^f
\end{aligned}$$

□

Lemma 2.2: $i\epsilon_s^f \Rightarrow \int s \leq \int f + 0.5$.

Proof: The proof of this lemma follows by induction on the number of times that the environment loop is performed. We first note that $i\epsilon_s^f$ can be rewritten as:

$$(\neg s^\top; \Delta \sqcap \mathtt{skip}); s!; \mathtt{\#1}; \mathtt{forever}\ \ (f?; \Delta_0; s!; \mathtt{\#1})$$

Using the law $f? \ = \ f^\top \sqcap (\neg f^\top; \mathtt{\#1}; f?)$ and distributivity of non-deterministic choice, it can be shown that this program is equivalent to:

$$(\neg s^\top; \Delta \sqcap \mathtt{skip}); s!; \mathtt{\#1}; \mathtt{forever} \begin{pmatrix} f^\top; s!; \mathtt{\#1} \\ \sqcap\ f^\top; \Delta; s!; \mathtt{\#1} \\ \sqcap\ \neg f^\top; \mathtt{\#1}; f?; \Delta_0; s!; \mathtt{\#1} \end{pmatrix}$$

Using the laws of loops this is equivalent to:

$$(\neg s^\top; \Delta \sqcap \mathtt{skip}); s!; \mathtt{\#1}; \mathtt{forever} \begin{pmatrix} f^\top; s!; \mathtt{\#1} \\ \sqcap\ \neg s^\top; f^\top; \Delta; s!; \mathtt{\#1} \\ \sqcap\ \neg s^\top; \neg f^\top; \mathtt{\#1}; f?; \Delta_0; s!; \mathtt{\#1} \end{pmatrix}$$

The semantic interpretation of this program takes the form:

$$P' \vee \exists n : \mathbb{N} \cdot P; Q^n; Q'$$

where P' corresponds to the partial execution of $(\neg s^\top; \Delta \sqcap \texttt{skip}); s!; \texttt{\#1}$, and P to its full execution. Similarly, Q' and Q correspond to the partial and complete execution of the loop body.

$$P \Rightarrow \lceil \neg s \rceil^*; (\lceil s \rceil \wedge l = 0.5); \lceil \neg s \rceil^*$$
$$Q \Rightarrow (\mathbf{true}; \lceil f \rceil; \mathbf{true} \ \wedge \ \lceil \neg s \rceil^*; (\lceil s \rceil \wedge l = 0.5)); \lceil \neg s \rceil^*$$
$$P' \Rightarrow \lceil \neg s \rceil^* \vee \lceil \neg s \rceil^*; (\lceil s \rceil \wedge l = 0.5); \lceil \neg s \rceil^*$$
$$Q' \Rightarrow \lceil \neg s \rceil^* \vee (\lceil \neg s \rceil^*; (\lceil s \rceil \wedge l = 0.5); \lceil \neg s \rceil^* \ \wedge \ \mathbf{true}; \lceil f \rceil; \mathbf{true})$$

Since $P' \Rightarrow \int s = 0.5$, it immediately follows that $P' \Rightarrow \int s \leq \int f + 0.5$.

We can also show, by induction on n, that $P; Q^n$ implies this invariant. An outline of the inductive case is given below:

$$\begin{aligned} & P; Q^{n+1} \\ = \ & P; Q^n; Q \\ \Rightarrow \ & (\textstyle\int s \leq \int f + 0.5); Q \\ \Rightarrow \ & (\textstyle\int s \leq \int f + 0.5); (\int s = 0.5 \wedge \int f \geq 0.5) \\ \Rightarrow \ & \textstyle\int s \leq \int f + 0.5 \end{aligned}$$

Finally, we can use this result to show $P; Q^n; Q' \Rightarrow \int s \leq \int f + 0.5$.

$$\begin{aligned} & P; Q^n; Q' \\ \Rightarrow \ & (\textstyle\int s \leq \int f + 0.5); Q' \\ \Rightarrow \ & (\textstyle\int s \leq \int f + 0.5); \int s = 0 \vee \\ & (\textstyle\int s \leq \int f + 0.5); (\int s = 0.5 \wedge \int f \geq 0.5) \\ \Rightarrow \ & \textstyle\int s \leq \int f + 0.5 \end{aligned}$$

This completes the required proof.

□

Lemma 2.3: Provided that $\texttt{dur}(P)$:

$$i\psi_s^f(P) \Rightarrow \textstyle\int f \leq \int s \ \wedge \ \int f \leq \int s \gg 1$$

Proof: Note that $i\psi_s^f(P)$ is a refinement of $(\neg f^\top; \Delta \sqcap \texttt{skip}); \texttt{forever}\ f!; s?; \Delta$ which is almost identical to $i\epsilon_f^s$.

The proof follows almost identically to that of lemma 2.2 except that, unlike the environment condition, $i\psi_s^f(P)$ cannot signal on f as soon as it receives a signal on s (since P must take some time to execute). This allows us to gain the extra 0.5 time unit.

□

Lemma 2.4: The environment conditions follow along the compilation process:

$$\Psi_s^f(P) \Rightarrow \textstyle\int f \leq \int s$$

Proof: The proof uses structural induction on the program:

In the base case, P cannot be decomposed any further, and hence $\Psi_s^f(P) = \psi_s^f(P)$. Therefore, by lemma 2.3, we can conclude that $\int f \leq \int s$.

Inductive case: We proceed by considering the three possible cases: $P = Q; R$, $P = Q \lhd b \rhd R$ and $P = Q * b$.

Sequential composition: $P = Q; R$

$$
\begin{aligned}
& \Psi_s^f(P) \\
= & \{ \text{ by definition of } \Psi \} \\
& \Psi_s^m(Q) \parallel \Psi_m^f(R) \\
\Rightarrow & \{ \text{ by inductive hypothesis } \} \\
& \int f \leq \int m \wedge \int m \leq \int s \\
\Rightarrow & \{ \leq \text{ is transitive } \} \\
& \int f \leq \int s
\end{aligned}
$$

Conditional: $P = Q \triangleleft b \triangleright R$

$$
\begin{aligned}
& \Psi_s^f(P) \\
= & \{ \text{ by definition of } \Psi \} \\
& \Psi_{s_Q}^{f_Q}(Q) \parallel \Psi_{s_R}^{f_R}(R) \parallel \mathcal{I}\text{nterface}_C \\
\Rightarrow & \{ \text{ by inductive hypothesis } \} \\
& \int f_Q \leq \int s_Q \wedge \int f_R \leq \int s_R \wedge \mathcal{I}\text{nterface}_C \\
\Rightarrow & \{ \text{ by definition of } \mathcal{I}\text{nterface}_C \} \\
& \int f_Q \leq \int s_Q \wedge \int f_R \leq \int s_R \wedge \int s_Q + \int s_R = \int s \wedge \\
& \qquad \int f = \int f_Q + \int f_R - \int (f_Q \wedge f_R) \\
\Rightarrow & \{ \text{ by properties of } \leq \text{ and } \int \} \\
& \int f \leq \int s
\end{aligned}
$$

Loops: $P = Q * b$

$$
\begin{aligned}
& \Psi_s^f(P) \\
= & \{ \text{ by definition of } \Psi \} \\
& \Psi_{s_Q}^{f_Q}(Q) \parallel \mathcal{I}\text{nterface}_L \\
\Rightarrow & \{ \text{ by inductive hypothesis } \} \\
& \int f_Q \leq \int s_Q \wedge \mathcal{I}\text{nterface}_L \\
\Rightarrow & \{ \text{ by definition of } \mathcal{I}\text{nterface}_L \text{ and integral reasoning } \} \\
& \int f_Q \leq \int s_Q \wedge \int f = \int s - (\int s_Q - \int f_Q) - \int (f_Q \wedge b \wedge s) \\
\Rightarrow & \{ \text{ by properties of } \leq \} \\
& \int f \leq \int s
\end{aligned}
$$

This completes the inductive step and hence the result holds by induction.

□

Lemma 2.5: $\Psi_s^f(P) \Rightarrow \int f \leq \int s \gg 1$

Proof: The proof follows almost identically to that of lemma 2.4.

□