A compositional rule for hardware design refinement

K. L. McMillan

Cadence Berkeley Labs
1919 Addison St., suite 303
Berkeley, CA 94704-1144
mcmillan@cadence.com

Abstract. We present an approach to designing verified digital systems by a sequence of small local refinements. Refinements in this approach are not limited to a library of predefined transformations for which theorems have been previously established. Rather, the approach relies on localizing the refinement steps in such a way that they can be verified efficiently by model checking. Toward this end, a compositional rule is proposed by which each design refinement may be verified independently, in an abstract environment. This rule supports the use of downward refinement maps, which translate abstract behavior detailed behavior. These maps may involve temporal transformations, including delay. The approach is supported by a verification tool based on symbolic model checking.

1 Introduction

Although significant progress has been made in automated verification of digital systems, most designs are still far too large and complex to be verified in a fully automatic way. The classical solution proposed to this problem is compositional reasoning. This means that properties of individual modules or components of a large system are verified in isolation, and these properties are then combined to prove properties of the system as a whole. One commonly proposed specification language for these properties is temporal logic [Pnu85], and systems of compositional inference rules have been developed to support "assume-guarantee" style proofs [Lam83] using various temporal logics (*e.g.*, [GL94]). In a compositional proof, one reasons thus:

$$\frac{\begin{array}{c} P, \phi \models \psi \\ Q \models \phi \end{array}}{P \parallel Q \models \psi}$$

Here, P and Q are processes, and ϕ is an environment assumption, necessary to prove that P satisfies specification ψ. Typically, however, the environment assumptions needed to verify interacting processes are interdependent. For example, process P may guarantee to satisfy an invariant ψ up to time $t+1$ only if Q satisies ψ up to time t, and *vice versa.* Such an inductive argument cannot be expressed in the above rule. If one attempts it, the result is a circular argument. One way to break the circularity is to model the environment as an abstract process. Kurshan [Kur87, Kur94] introduced the following style of reasoning for Moore machines:

$$\frac{\begin{array}{c}P \parallel Q' \Rightarrow P' \\ P' \parallel Q \Rightarrow Q'\end{array}}{P \parallel Q \Rightarrow P' \parallel Q'}$$

where $\Rightarrow$ can be replaced by any suitable process preorder. Here, the abstract process Q' takes the role of environment assumption when verifying P, and P' does the same when verifying Q. The circularity is broken inductively, as a result of the delay of one time unit from input to output of the Moore machines. Alur and Henzinger [AH96] extended this to the case of Mealy machines where there are no combinational cycles.

A limitation of this kind of proof rule is that the abstract processes P' and Q' do not typically have the same inputs and outputs as the detailed processes P and Q. In order for P' and Q' to be simple, they necessarily communicate at a more abstract level. In Kurshan's methology, this problem is approached by using process homomorphisms. This means that the user provides a function ϕ that maps detailed signals to abstract signals. One can thus reason compositionally as follows:

$$\frac{\begin{array}{c}\phi(P) \Rightarrow P' \\ \phi(Q) \Rightarrow Q'\end{array}}{\phi(P \parallel Q) \Rightarrow P' \parallel Q'}$$

Note, however, that we cannot use Q' as an environment assumption unless we are able to effectively invert the function ϕ. This is necessary to translate outputs of the abstract process Q' into inputs of the detailed process P. On the other hand, *downward* maps can be used effectively to provide the both the inputs of P (*i.e.*, its environment) and also the correctness conditions for its outputs, as a function of the abstract behavior of P' and Q'. This effectively puts the verification of P in an abstract context, an observation has been made in the context of symbolic simulation by Bryant and Beatty [BB94] and in the context of theorem provers by Cyrluk [Cyr96].

Note also that upward maps can be very complex. In the case of pipelines, for example, the upward abstraction map involves flushing the entire state of the pipeline, which may contain many instructions. Although in some cases this complexity can be dealt with, using BDD's [BF89] or sophisticated decision procedures [BD94, JDB95], we would prefer a methodology that decomposes the verification problem into small subproblems. In the case of pipelines, for example, downward refinement maps involving delay can yield separate verification subproblems for each stage of the pipeline.

To support such a compositional methodology in a model checking context, we present a system based on a generalized compositional rule for Mealy machines. It allows both upward and downward refinement maps, which are represented as arbitrary processes. Hence, maps may involve state and delay, if necessary. Further, the system is flexible enough to allow non-hierarchical abstractions. That is, an abstract specification may have a different structural decomposition from the low level implementation, and many abstract-level components may be multiplexed onto the same collection of low-level components. This flexibility to choose an arbitrary decomposition of the specification can be used to simplify the resulting verification subproblems. The system is implemented on top of the SMV symbolic model checker [McM93].

2 A compositional rule for Mealy machines

We begin by introducing a compositional rule for Mealy machines. For the present purposes, a Mealy machine will be defined as a collection of recurrence equations involving either zero delay or unit delay. For flexibility in specification, we allow machines to be underspecified, in the sense that there may be many solutions of the equations for any given input sequence. This does not however, imply nondeterminism in the automata theoretic sense, since our "machines" have no notion of internal state.

To be more specific, let $\mathcal{S}$ be a finite collection of signals, and let $\mathcal{V}$ be a finite universe of values. We interpret a signal as a sequence of values, or a function $\mathbb{N} \rightarrow \mathcal{V}$. Let a *model* be any function $\pi : \mathcal{S} \rightarrow \mathbb{N} \rightarrow \mathcal{V}$. A *machine* is a predicate M of the form:

$$\bigwedge_{\sigma \in \mathcal{S}} M_\sigma$$

The assertions M_σ, called components, may be in one of two forms, representing generalized *gates* and *latches*. A gate is of the form:

$$\bigvee_j \sigma(t) = f(\gamma_1(t) \ldots \gamma_k(t))$$

where the signals $\gamma_1 \ldots \gamma_k$ are the inputs of the gate, and f is a function $\mathcal{V}^k \rightarrow \mathcal{V}$. The finite disjunction allows the output of the gate to be incompletely specified as a function of its inputs. A latch is similar to a gate, but involves one time unit of delay, and a set of possible initial values. It is a component M_σ of the form:

$$\begin{array}{c} \bigvee_j \sigma(t+1) = f(\gamma_1(t) \ldots \gamma_k(t)) \\ \wedge \\ \bigvee_j \sigma(0) = init_j \end{array}$$

This specifies the possible values of σ at time $t+1$ as of function of the inputs at time t, and also specifies the possible values $init_j$ at time $t = 0$.

We will tacitly identify a machine with the set of models that satisfy it. We will say that machine Q implements machine P when $Q \Rightarrow P$, which is the same as saying that the set of models of Q is contained in the set of models of P.

Now, suppose we wish to prove that $Q \Rightarrow P$. Since P is a conjunction of assertions P_σ, expressible in temporal logic, we could simply use model checking to verify $Q \models P_\sigma$ for each σ. However, this would be unlikely to be effective in practice, since the state space of Q would be too large. To simplify the model checking problem, we could take only a subset of the components of Q as the "environment" when checking P_σ (a technique called localization), but it still might require a large number of components. Instead, assuming that P is simple and abstract, while Q is complex, we might like to take some other components of P as environment assumptions while proving P_σ. Intuitively, this would put the verification of P_σ in a more "abstract" context. Thus, for example, we might assume $P_{\sigma'}$ is correctly implemented when checking P_σ and vice versa. We can show that this reasoning is sound, provided there are no cycles of "gates".

To be more precise, let $<_M$, the dependency relation of machine M, be the set of pairs (γ, σ) such that M_σ is a gate (has zero delay) and γ is and input of M_σ. Now suppose there are no cycles in the joint dependency relations of machines Q and P. To verify $Q \Rightarrow P_\sigma$, we may instead verify $\mathcal{E}^\sigma \Rightarrow P_\sigma$, where $\mathcal{E}^\sigma$ is an "environment" machine, made up of arbitarily chosen components of P and Q, provided of course we do not chose P_σ itself.

Theorem 1. *Let P and Q be machines. For all $\sigma \in \mathcal{S}$, let $\mathcal{E}^\sigma$ be a machine such that:*

- *for all $\sigma' \in$ signals: $\mathcal{E}^\sigma_{\sigma'} = P_{\sigma'}$ or $\mathcal{E}^\sigma_{\sigma'} = Q_{\sigma'}$, and*
- *$\mathcal{E}^\sigma_\sigma = Q_\sigma$.*

Let $<^*$ be the relation $(<_P \cup <_Q)^*$. If $<^*$ is irreflexive then the following inference rule is sound:

$$\frac{\textit{for all } \sigma\text{: } \mathcal{E}^\sigma \Rightarrow P_\sigma}{Q \Rightarrow P}$$

Proof. Define a lexical order $<$ over $\mathbb{N} \times \mathcal{S}$ where $(\tau', \sigma') < (\tau, \sigma)$ iff $\tau' < \tau$, or $\tau' = \tau$ and $\sigma' <^* \sigma$. Further, let $P_\sigma(\tau)$ denote P_σ for $t = \tau$. Now, consider a model π. Assume $\pi \models Q$ and assume by inductive hypothesis that $\pi \models \mathcal{E}^\sigma_{\sigma'}(\tau')$ for all $(\sigma', \tau') < (\tau, \sigma)$. Note that by definition, $\pi \models \mathcal{E}^\sigma_\sigma(\tau)$, since $\mathcal{E}^\sigma_\sigma = Q_\sigma$.

Now construct a model π' from π by changing only the values $\sigma'(\tau')$ for $(\sigma, \tau) < (\sigma', \tau')$, such that $\pi' \models \mathcal{E}^\sigma$. This can be done because $<^*$ containts $<_{\mathcal{E}^\sigma}$, hence each $\sigma'(\tau')$ can be chosen only as a function of previous values w.r.t. $<$. Since $\mathcal{E}^\sigma \Rightarrow P_\sigma$ it follows in particular that $\pi' \models P_\sigma(\tau)$, and hence $\pi \models P_\sigma(\tau)$. By induction over $<$, it follows that $\pi \models P$.

We can extend the above result to the case of proving that Q simultaneously implemements a collection of specifications $P_1, \ldots, P_n$. This theorem forms the basis of a system for design refinement, described in the next section. The proof is omitted here, but is along the same lines as the previous theorem.

Theorem 2. *Let Q and $P^1 \ldots P^n$ be machines. For all $i = 1 \ldots n$ and $\sigma \in \mathcal{S}$, let $\mathcal{E}^{i\sigma}$ be a machine such that:*

- *for all $\sigma' \in$ signals: $\mathcal{E}^{i\sigma}_{\sigma'} = Q_{\sigma'}$ or $\mathcal{E}^{i\sigma}_{\sigma'} = P^j_{\sigma'}$ for some j, and*
- *$\mathcal{E}^{i\sigma}_\sigma = Q_\sigma$.*

Let $<^*$ be the relation $[(\bigcup_i <_{P^i}) \cup <_Q]^*$. If $<^*$ is irreflexive then the following inference rule is sound:

$$\frac{\textit{for all } i, \sigma\text{: } \mathcal{E}^{i\sigma} \Rightarrow P^i_\sigma}{Q \Rightarrow \bigwedge_i P_i}$$

3 Partial machines and refinement

We now introduce a *refinement* framework, that makes it possible to define a design by a collection of incremental changes to a specification machine, and to verify that the resulting machine (called the implementation machine) implies the orignal abstract machine. Each incremental change will be referred to as a *layer*, and is essentially a partially defined machine.

Let a *layer* M be an assertion of the form $\wedge_{\sigma \in \mathcal{S}(M)} M_\sigma$ where $\mathcal{S}(M) \subseteq \mathcal{S}$ and the assertions M_σ are either gates or latches, as before. A *design* is a partial order

$\mathcal{D} = (\mathcal{M}, <_{\mathcal{D}})$, where $\mathcal{M}$ is a set of layers. The intuition behind $<_{\mathcal{D}}$ is that $Q <_{\mathcal{D}} P$ when Q is intended as an incremental modification of P, in which case we say Q *refines* P. In order for an implementation to be uniquely defined, we require that for any signal σ, there is a unique least layer $\mathcal{I}^\sigma$ w.r.t. $<_{\mathcal{D}}$ such that $\sigma \in \mathcal{S}(\mathcal{I}^\sigma)$. The conjunction of these minimal definitions $\mathcal{I}^\sigma$ is termed the *implementation machine* of $\mathcal{D}$ and is denoted $\mathcal{I}^{\mathcal{D}}$. In the simplest case $<_{\mathcal{D}}$ will be a linear order over machines $\mathcal{M}_1, \ldots, \mathcal{M}_n$. In this case, the implementation machine is the result obtained by starting with $\mathcal{M}_1$ (the specification) and substituting components of $\mathcal{M}_2, \ldots, \mathcal{M}_n$ in sequence.

A design $\mathcal{D}$ will be said to be *correct* when

$$\mathcal{I}^{\mathcal{D}} \Rightarrow \bigwedge \mathcal{M}_{\mathcal{D}}$$

that is, when the implementation machine implies every layer of $\mathcal{D}$. In the linear order case, this implies in particular that it implements the original specification $\mathcal{M}_1$.

Note that we can verify correctness of a design compositionally using the inference rule of theorem 2. This requires us to choose an environment machine $\mathcal{E}^{M\sigma}$ to verify each component M_σ of each layer M in the design, excepting the implementation components. While the environments may be chosen manually, the following two heuristics can be applied automatically:

- For each σ, choose $\mathcal{E}^\sigma = M_\sigma$, where M is the maximal layer under $<_{\mathcal{D}}$ that defines σ.
- Drop any signal definitions that topologically cannot influence σ.

If we use these rules when verifying a sequence of local modifications, the verification of any given modification does not see the other modifications, since the environment is selected from the earliest, most abstract definitions.

3.1 Implementation in SMV

The verification framework described in the previous two sections has been implemented on top of the SMV model checker. The system has a simple language for describing Mealy machines. In this language, a gate is described by a statement of the form:

<σ> := <$f(\gamma_1, \ldots, \gamma_k)$>;

while a latch is specified in the following way:

init(<σ>) := <$init_\sigma$>;
next(<σ>) := <$f(\gamma_1, \ldots, \gamma_k)$>;

In either case, we can leave a signal underspecified by indicating a choice of values in set brackets. For example,

```
x := y + {0,1};
```

stands for

$$\begin{array}{c} x(t) = y + 0 \\ \vee \\ x(t) = y + 1 \end{array}$$

The language also includes some "syntactic sugar" over the basic gates and latches, including nested conditional statements, and a method of specifying default values when one branch of a conditional is unspecified.

Each layer of the design is given a name, and is introduced by the keyword "layer". The partial order $<_{\mathcal{D}}$ is specified by statements of the form:

<Q> `refines` <P>;

which denotes $Q <_{\mathcal{D}} P$. The SMV system verifies that the design thus specified is correct, according to the definition of the previous section. It does this by translating each non-implementation component of M_σ of each layer M into temporal logic.[1] For each such component an evironmeont $\mathcal{E}^{M\sigma}$ is selected, using user input and the above described heuristics. This environment is used as a model for model checking the temporal formula. The system also verifies the side condition of the compositional rule, requiring that the joint dependency relation be acyclic.

3.2 Example

As an example of compositional verification using the system, consider a "resource manager" circuit, which is used to allocate and free a collection of resources (say a collection of packet buffers). The module maintains a vector of status bits that indicate, for each resource, whether it is currently allocated or not. When an "allocate" request is received, the module may output the index of some free resource (changing the status of this resource to "allocated") or it may output a negative acknowledgement (NACK). When a "free" request is received, the module inputs a resource index and changes the status of that resource to "free". Allocation requests have either high or low priority. A low priority request must result in a NACK if fewer than k buffers are currently free (where k is a fixed constant).

A naïve and somewhat underspecified original specification of the resource manager is shown in figure 1. When an allocate request is received, it chooses a resource arbitrarily. If that resource is currently allocated, it produces a NACK. To check whether a low priority request is allowed, it simply sums up the vector of "allocated" bits and compares the result to k. Notice the "dafault" construct in the definition of "allocated". The meaning of this construct is that the first statement provides the default value for any given element of the vector when the second statement does not define it. For a latch, the default in case neither statement defines it is to keep the old value.

There are two refinements we would like to make to this specification. First, it is too time consuming to sum up the vector of "allocated" bits on every cycle. We would prefer to use an up/down counter to maintain a running total the number of "allocated" bits that are set. This refinement is shown in figure 2. Second, we need to choose a policy for selecting a resource to allocate. To do this, we will use a priority encoder to choose the unallocated resource of lowest index. To compute the NACK signal more quickly, we simply test the "sum"

[1] Note, this requires a minor extension to CTL that allows the "next" value of a variable to be expressed. This does not increase the complexity of the model checking problem over ordinary CTL.

```
layer toplevel : {
  init(allocated) := 0;
  index_out := 0 ..(n - 1);
  NACK := alloc_req & (allocated[index_out]
                     | ~high_priority & (n - sum) < k);
  default
    if(free_req)next(allocated[index_in]) := 0;
  in
    if(alloc_req & ~NACK)next(allocated[index_out]) := 1;
  sum := sigma(i = 0; i < n; i = i + 1)allocated[i];
}
```

Fig. 1. Original specification of resource manager.

value to see if there are any available resources. This refinement is shown in figure 2.

```
layer refinement1 : {
   init(sum) := 0;
   next(sum) := sum + (alloc_req & ~NACK)
            - (free_req & allocated[index_in]);
}
refinement1 refines toplevel;

layer refinement2 : {
  index_out := priority_encode(~allocated);
  NACK := alloc_req & (high_priority ? sum = n : (n - sum) < k);
}
refinement2 refines toplevel;
```

Fig. 2. Two refinements of the resource manager.

Note that these two refinements are mutually dependent. That is, if the refined "sum" logic computes its value incorrectly, then the NACK signal we produce may be incorrect. On the other hand, if the refined "NACK" logic is incorrect, causing an already allocated buffer to be allocated, then the "sum" counter will be corrupted. These two parts of the circuit are, in effect, engaged in a protocol, where each part guarantees to produce a correct output only if all its previous inputs have been correct. Despite this circularity, we can use the compositional rule to verify the two refinements separately, where each refinement uses the original specification as its environment. This simplifies the verification process, since the original specification has fewer latches than the refined version. Note also, that if the resource manager is used as part of the system, we can use the simple original specification as part of the enviroment when verifying other parts of the system, and need not take into account the refinements.

4 Hiding internal state of specifications

Typically, a specification contains some intermediate signals that are not intended to be part of the implementation *per se*, but are used only for specification purposes. For example, we might want to specify that a machine counts, producing a one at its output for every n ones occurring at the input. To do this, we could introduce a signal representing, for example, a binary modulo-n counter:

```
init(count) := 0;
next(count) := count + inp mod n;
out := inp & (count = n - 1);
```

There is, however, no reason why "count" should appear in the implementation, as it would be perfectly valid for the implementation to use, for example, a "one hot" encoded counter. What we would actually like to specify is that, for any implementation behavior, *there exists* a valuation for the signal "count" that makes it a legal behavior of the specification. In other words, we would like to be able to hide certain signals in the specification, in order to specify only externally visible behavior. Toward this end, we define a notion of "projected design" that makes it possible to write specifications with hidden internal state.

Let a *projected design* be a structure $\mathcal{P} = (\mathcal{D}, A_\mathcal{P})$, where $\mathcal{D} = (\mathcal{M}, <_\mathcal{D})$ is a design, and $A_\mathcal{P} \subseteq \mathcal{S}$ is the set of internal, or unobservable signals. The implementation of a projected design is

$$\mathcal{I}^\mathcal{P} = \bigwedge_{\sigma \notin A_\mathcal{P}} \mathcal{I}^\mathcal{D}_\sigma$$

That is, in the projected design, the implementation includes only those implementation components of $\mathcal{D}$ that are *not* considered internal to the specification. A projected design $\mathcal{P}$ is said to be *correct* when

$$\mathcal{I}^\mathcal{P} \Rightarrow \exists A_\mathcal{P}.\ \bigwedge \mathcal{M}_\mathcal{D}$$

That is, for every behavior of the implementation, there must exist a valuation of the internal signals such that every layer in $\mathcal{D}$ is satisfied. This is in fact guaranteed to hold provided $\mathcal{D}$ is correct, and the unprojected implementation machine $\mathcal{I}^\mathcal{D}$ satisfies a simple condition: no signal not in $A_\mathcal{P}$ may depend on a signal in $A_\mathcal{P}$, via a gate or a latch. Put another way, the signals that remain in the projected implementation must be closed under the dependency relation. If this is the case, then for every model of the projected implementation, we can construct a valuation for the internal signals to create a model of the unprojected implementation. Thus, if the unprojected design is correct, then the projected design must also be correct.

Note that in the unprojected implementation, we do allow a dependency of internal signals on "visible" signals. These functions play the role of witness functions for the hidden signals, as we see in the proof of the following theorem:

Theorem 3. *Let $\mathcal{P} = (\mathcal{D}, A_\mathcal{P})$ be a projected design, such that for all $\sigma \notin A_\mathcal{P}$, for all inputs γ of $\mathcal{I}^\mathcal{D}_\sigma$, $\gamma \notin A_\mathcal{P}$. If*

$$\mathcal{I}^\mathcal{D} \Rightarrow \bigwedge \mathcal{M}_\mathcal{D}$$

then

$$\mathcal{I}^\mathcal{P} \Rightarrow \exists A_\mathcal{P}.\ \bigwedge \mathcal{M}_\mathcal{D}$$

Proof. Let π be a model of $\mathcal{I}^{\mathcal{P}}$ and let $<$ be the same lexical order used in the proof of theorem 1. By modifying only the values of signals in $A_{\mathcal{P}}$, we can construct a model π' of $\mathcal{I}^{\mathcal{D}}$, the unprojected design. This is because the values of these signals depend functionally on only previous values w.r.t $<$, and no signals not in $A_{\mathcal{P}}$ depend on values that are modified. It follows that $\pi' \models \bigwedge \mathcal{M}_{\mathcal{D}}$ and hence $\pi \models \exists A_{\mathcal{P}}.\ \bigwedge \mathcal{M}_{\mathcal{D}}$.

4.1 Refining internal signals – witness functions

An internal signal that is underspecified may be thought of as representing a nondeterministic choice. By refining this signal, we can in effect provide a "witness" that shows why any given execution of the implementation satisfies the specification. Note that neither the original specification nor the refinement of an internal signal is part of the implementation. The witness function merely serves as part of the proof of correctness of the design.

As an example, figure 3 shows an abstract specification for a two-way synchronous arbiter. An underspecified signal called "choice" determines which of the two requesters will be acknowledged. This is an internal signal, declared elsewhere using the keyword "abstract". Note that "choice" is nondeterministic when both request simultaneously. Figure 3 also shows a refinement of this specification, in which a latched signal called "turn" is used to break ties in a fair manner. The signal "choice" is redefined to be a function of "turn". This definition is not part of the implementation, but is simply used to prove that there exists a valuation of "choice" that makes the specification true in all cases.

```
layer toplevel : {
   choice := (req[0] & ~req[1]) ? 0 : (req[1] & ~req[0]) ? 1 : {0,1};
   ack[0] := req[0] & (choice = 0);
   ack[1] := req[1] & (choice = 1);
}

layer refinement : {
  init(turn) := 0;
  if(ack[turn]) next(turn) := ~turn;
  ack[0] := req[0] & (~req[1] | turn = 0);
  ack[1] := req[1] & (~req[0] | turn = 1);
  choice := (req[0] & ~req[1]) ? 0 : (req[1] & ~req[0]) ? 1 : turn;
}
refinement refines toplevel;
```

Fig. 3. Using a witness function for a nondeterministic choice.

5 Refinement maps

One important use of refinements is in specifying the downward refinement maps that give the detailed signals in terms of abstract signals. In the present framework, a refinement map is simply an itermediate layer in the design. To use a refinement map, one creates a sequence of two layers. The first defines the refinement maps, giving some implementation signals as a function of specification signals. Each component of the refinement map must be verified. When verifying one component, we can we can choose to use any other refinement map components in the environment. Thus, when verifying that the outputs of a given module are correct, we can use the refinement maps to generate the inputs to that module as a function of the abstract specification. In fact, the signal "sum" in the example of figure 1 can be viewed as playing the role of a refinement map. Notice how it divides the refinement verification problem into two parts, where each can be verified in the environment of the abstract specification. Note also that when we specify refinement maps, we can use any function that can be defined by a Mealy machine. In particular, this allows us to use refinement maps that involve delay. This is useful for hardware structures that have "latency", such as pipelines. The refinement maps may also include arbitrary finite state machines.

5.1 Example

As an example, suppose we would like to design a tranmitter/receiver pair that sends n-bit bytes (for fixed n) serially over a single wire. The original specification might look something like the code of figure 4. A "send" signal indicates that input data are ready to be sent, "NACK" indicates that the transmitter is busy, "DAV" indicates that data are available at the receiving end, and "received" indicates the the receiving end is ready for new data. Notice that "DAV" is underspecified, in the sense that when data are available, it may be either true or false. This is intended to allow for arbitrary delay in the actual transmission of the data. Notice also the conditional in the definition of "output_data". Because this is a gate, the default in case the condition is false is that the signal is unspecified. This means the implementation may output any data value in the case when "DAV" is false.

```
layer toplevel :  {
  init(full) := 0;
  next(full) := received ? 0 : send ? 1 : full;
  NACK := send & full;
  if(send & ~full) next(data) := input_data;
  DAV := full ? {0,1} : 0;
  if(DAV) output_data := data;
  received := DAV ? {0,1} : 0;
}
```

Fig. 4. Specification of transmitter/receiver.

The next step in the design is to formulate a refinement map that defines the sequence of bits seen on the serial line as a function of the abstract specification signals. To do this, we need to introduce some state, in the form of a counter that keeps track of the bit number being transmitted. The refinement map is shown in figure 5.

Given this refinement map, we can now design and verify the receiver and transmitter separately. The actual implementation of these components might use shift registers, as shown in figure 6. The enviromnent for verifying each of

```
layer refinement_map :  {
  init(transmitting) := 0;
  if(send & ~NACK) next(transmitting) := 1;
  else if(count = (n - 1)) next(transmitting) := 0;
  init(count) := 0;
  if(transmitting){
    next(count) := count + 1 mod n;
    serial_line := data[count];
  }
}
refinement_map refines toplevel;
```

Fig. 5. Refinement map defining serial line behavior

these two component refinements includes the refinement map and the abstract specification, but not the other component. The purpose of the refinement map is to define the interface between two components, and thus allow separate design and verification of the two components. In general, refinement maps can provide a way of managing the complexity of interfaces between modules, by defining interface signals, which may be encoded in fairly complex ways, in terms of simpler, more abstract data streams.

```
layer tx_refinement :  {
  if(send & ~NACK) next(tx_shifter) := input_data;
  else next(tx_shifter) := tx_shifter >> 1;
  serial_line := tx_shifter[0];
}
tx_refinement refines refinement_map;

layer rx_refinement :  {
  if(transmitting){
    next(rx_shifter[n-1]) := serial_line;
    next(rx_shifter[(n-2)..0]) := rx_shifter >> 1;
  }
  DAV := full & ~transmitting;
  output_data := rx_shifter;
}
rx_refinement refines refinement_map;
```

Fig. 6. Separate refinements of transmitter and receiver.

6 Conclusions

We have described a compositional framework for the verification of hardware designs. It is designed to allow the expression of downward refinement masps as Mealy machines, and to support design by a sequence of incremental modifications that may be verified independently. The framework has been implemented on top of the SMV symbolic model checking system. Although it is not discussed here, the system also supports assume-guarantee style reasoning using linear time temporal logic. This is intended mainly for reasoning about eventualities.

One extension to the system that is planned is to support a notion of streams (such as streams of intructions of memory transactions) that are ordered, but are not assigned specific times in the implementation. Each element of such a stream could be defined as a finite state machine representing the different stages of execution of the given operation in the machine. By making each abstract operation a distinct "layer" of the specification, one could verify in a modular way that a single operation is processed through the system correctly, and then deal separately with other issues such as ordering guarantees and liveness. Finally, it is also possible that a hybrid approach could be taken, where some refinement obligations are handled by a model checker and others by a general-purpose proof assistant, or a collection of predefined transformations. This might be particularly useful for handling large, regular structures such as memories, and hardware that manipulates large data items, such as packets. A compositional framework of this sort could provide a practical way of integrating model checking and theorem proving.

References

[AH96] R. Alur and T. A. Henzinger. Reactive modules. In *11th annual IEEE symp. Logic in Computer Science (LICS '96)*, 1996.

[BB94] D. L. Beatty and R. E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *31st Design Automation Conference*, pages 596–602, 1994.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In D. L. Dill, editor, *Conf. Computer-Aided Verification (CAV '94)*, volume 818 of *LNCS*. Springer-Verlag, 1994.

[BF89] S. Bose and A. Fisher. Verifying pipelined hardware using symbolic logic simulation. In *IEEE International Conference on Computer Design*, 1989.

[Cyr96] D. Cyrluk. Inverting the abstraction mapping: a methodology for hardware verification. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *LNCS*. Springer-Verlag, 1996.

[GL94] O. Grümberg and D. E. Long. Model checking and modular verification. *ACM Trans. Programming Languages and Systems*, 16(3):843–871, 1994.

[JDB95] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *IEEE/ACM Int. Conf. on Computer Aided Design (ICCAD '95)*, 1995.

[Kur87] R. P. Kurshan. Reducibility in analysis of coordination. In *LNCS*, volume 103, pages 19–39. Springer-Verlag, 1987.

[Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton, 1994.

[Lam83] L. Lamport. Specifying concurrent program modules. *ACM Trans. Programming Languages and Systems*, 5:190–222, 1983.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer, 1993.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–144. Springer-Verlag, 1985.

[Wol83] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.