Mechanized Verification of Refinement

Niels Maretti

Department of Computer Science, Technical University of Denmark,
DK-2800 Lyngby, Denmark. e-mail: nbm@id.dtu.dk

Abstract. This paper describes a mechanized approach to verifying that one concrete design is a refinement of another abstract design. A widely used notion of refinement is trace inclusion, which implies that each externally visible behavior of the concrete design can also be caused by the abstract design. In some cases this is too restrictive and the verification technique proposed here is based on a more liberal notion where information about the environment is exploited. A verification technique is presented for designs written in the design language SYNCHRONIZED TRANSITIONS. The verification technique is supported by a prototype tool for mechanizing 1) the axiomatization of the design descriptions in the logic of an existing theorem prover, and 2) the generation of proof obligations. Based on the axiomatization of the design descriptions, the proof obligations can be discharged using the theorem prover.

1 Introduction

This paper describes a mechanized approach to verifying that one concrete design is a refinement of another abstract design. For mechanized verification to be practical it is important to find verification techniques which break the proof into a number of independent steps of modest complexity. The main contribution of this work is a notion of refinement which is both powerful enough to allow interesting designs to be verified and yet simple enough to make mechanization feasible.

Several approaches exists for formal verification of refinement, for example [1, 5, 7, 8]. [5, 8] use forward and backward simulation, whereas [1, 7] use refinement mappings and prophecy variables. In the field of hardware verification, two very different approaches are [2] and [4]. In [4] the hardware is described in higher order logic and within this logic refinement corresponds to equivalence or implication. Bryant [2] uses a simulator to prove refinement. Common for all of the approaches is that they use trace inclusion as the fundamental notion of equivalence.

Trace inclusion means that each externally visible behavior of the concrete design can also be caused by the abstract design. However, in some cases trace inclusion is too restrictive. Consider, for example, an abstract description of a multiplier that performs the multiplication of two positive integers in a single operation, i.e., if two inputs x, y are provided, the result s $(= x * y)$ is available immediately afterwards. Following an example in [4], the multiplier can be realized by accumulating y in s x times. In this case a number of intermediate results $(y, 2*y, \ldots, (x-1)*y)$ are observed, before the right value of s $(x*y)$ is observed. Consequently, the notion of trace inclusion does not apply to this example. In case of the multiplier, it would be useful to be able to disregard s during the computation and only to focus on the final value.

This paper proposes a notion of refinement that allows for temporarily leaving s out of consideration by taking the environment into account. Based on information about the environment, the notion of refinement ensures that the concrete design can correctly replace the abstract in the environment, even though the concrete design does not refine the abstract according to trace inclusion. The notion of refinement is based on trace inclusion, but extended to take the environment into account. A prototype tool is developed for verifying that one design refines another according to this notion of refinement.

Section 2 defines the model of a design. Section 3 explains how the environment is taken into account. Section 4 defines the notion of refinement in terms of the model. To verify that one design is a refinement of another, section 5 provides a verification technique for designs written in the design language SYNCHRONIZED TRANSITIONS. The verification technique is developed in preparation for using a theorem prover. In section 6, the verification technique is applied to part of the Tamarack microprocessor [6].

2 Computational Model

Above, the terms design and environment have been used informally. Below, designs and environments are described as *cells*. The computational model of a cell, C, is a transition system identified by (1) the state space, $\mathcal{S}_C$, spanned by state variables in $\mathcal{V}_C$, (2) the set of initial states, $\mathcal{I}_C$, and (3) the set of state transitions, $\mathcal{T}_C$. These components are described below.

A cell, C, operates on a set of typed state variables, $\mathcal{V}_C$. Some of the state variables are hidden from the environment, they are called *local* variables, $\mathcal{L}_C$. The rest of the state variables are accessible for the environment, they are called *interface* variables, $\mathcal{E}_C$. Communication between cells takes place by means of shared interface variables. The state space of C, determined by the state variables and their corresponding types is denoted $\mathcal{S}_C$. For $v \in \mathcal{V}_C$ and $\sigma \in \mathcal{S}_C$, $v.\sigma$ denotes the value of v in σ. A state in which x has the value 5 and y has the value 2 is written $\{x = 5, y = 2\}$. For $x \in \mathcal{V}_C$, let $val(x)$ denote the set of values that x can have according to its type. For $v \in val(x)$, $\sigma[v/x]$ denotes the state σ where the value of x is replaced by the value v, and the values of the rest of the variables are unchanged.

In order to compare parts of states, projection is defined. Let $\sigma \in \mathcal{S}_C$, $m \subseteq \mathcal{V}_C$. $\sigma \downarrow m$ is the state containing variables in m only, where the values of the variables in m are the same in σ and $\sigma \downarrow m$. For example

$$\{x = 5, y = 2, s = 0\} \downarrow \{x, y\} \;=\; \{x = 5, y = 2\}$$

$(\mathcal{S}_C)^\omega$ denotes the set of infinite sequences of states of C. Let t denote the sequence $< \sigma_0, \sigma_1, \ldots, \sigma_i, \ldots >$. Element i in t, is denoted $t[i]$, i.e. $t[i] = \sigma_i$.

Projection is extended to apply to sequences and sets of states, i.e.

$$< \sigma_0, \sigma_1, \ldots > \downarrow m \;=\; < \sigma_0 \downarrow m, \sigma_1 \downarrow m, \ldots >$$

$$\{\sigma_0, \sigma_1, \ldots\} \downarrow m \;=\; \{\sigma_0 \downarrow m, \sigma_1 \downarrow m, \ldots\}$$

A cell, C, defines a set of initial states called $\mathcal{I}_C$, and a set of state transitions, called $\mathcal{T}_C$. Each state transition is a pair (σ, σ') meaning that C can perform a state change from σ to σ'. Returning to the multiplier, the pair $(\{x = 3, y = 2, s = 0\}, \{x = 3, y = 2, s = 6\})$ belongs to $\mathcal{T}$.

A cell determines a computation, represented by a trace $< \sigma_0, \sigma_1, \sigma_2, \ldots >$, i.e., an infinite sequence of states. Each state belongs to $\mathcal{S}_C$. σ_0 belongs to $\mathcal{I}_C$, and for each pair of succeeding states (σ_i, σ_{i+1}),

$$(\sigma_i, \sigma_{i+1}) \in \mathcal{T}_C \;\vee\; \sigma_i = \sigma_{i+1}$$

In section 4.1 it is explained why repetitions $(\sigma_i = \sigma_{i+1})$ are allowed. The set of traces, $\mathcal{B}_C$, that a computation of a cell, C, can determine is defined by

$$\mathcal{B}_C = \{s \in (\mathcal{S}_C)^\omega | s[0] \in \mathcal{I}_C \wedge \forall i \geq 0 : (s[i], s[i+1]) \in \mathcal{T}_C \;\vee\; s[i] = s[i+1]\}$$

The states present in any trace of $\mathcal{B}_C$ are referred to as the reachable states.

2.1 Composition

Above, the set of traces of one cell is defined. Often designs are composed of several cells. Below, composition of cells is defined.

For two cells, C and D, composition is denoted $C||D$. In order to avoid renaming, it is assumed that interface variables of C and D are identically named and typed, and that local variable names of C and D are not overlapping. This means that $\mathcal{S}_{C||D}$ is identified by $\mathcal{V}_C \cup \mathcal{V}_D$ as explained above. $\mathcal{I}_{C||D}$ is the set of states from $\mathcal{S}_{C||D}$ such that the variables of C and D are initialized according to the states in $\mathcal{I}_C$ and $\mathcal{I}_D$, respectively. $\mathcal{T}_{C||D}$ is the set of pairs of states from $\mathcal{S}_{C||D} \times \mathcal{S}_{C||D}$ such that for each pair (σ, σ'), the variables of $\mathcal{V}_C$ are changed according to $\mathcal{T}_C$ and the variables of $\mathcal{L}_D$ are unchanged, or vice versa.

3 Interface Protocols

The interface protocol is an important part of the interface description (together with the types of the interface variables, etc.). The interface protocol documents the communication pattern of the cell and the environment. If the communication pattern is formalized, it can be used for reasoning about cells, for example, for refinement purposes.

Informally, the information about the communication is stated as a condition for each interface variable. The condition indicates whether the value of the variable can be used by the environment in a given state. For each state, this defines a subset of the interface variables containing variables that the environment may use. This subset is referred to as the *observable* part of the interface.

Instead of focusing on the whole interface, the notion of refinement defined below is only concerned with the observable part of the interface. Recall, for example, the multiplier where the aim is to disregard s until s has reached the value of $x * y$. Following the outline given here, s is kept out of the observable part of the interface until the correct value $(x * y)$ has been reached, whereupon s is included in the observable part of the interface.

Below, interface protocols are defined, and it is described what it means for an environment to respect an interface protocol.

3.1 Definition

A condition, $V(x)$, is associated with each interface variable x. The condition is an assertion on the interface variables. The collection of interface conditions constitute the interface protocol. The variable x is defined to be in the observable part of the interface in a state σ iff $V(x)$ evaluates to true in σ, written $V(x).\sigma$.

Example. Consider again the multiplier. The interface of the multiplier contains the variables x, y, s and rdy. In the multiplier, rdy is set true to indicate that the computation of s has finished. Therefore the environment should not use the value of s unless rdy is true. This is formalized using the interface protocol

$$V(s) = rdy$$

Consider the state $\sigma = \{x = 4,\ y = 2,\ s = 4,\ rdy = F\}$. The observable part of σ is $\{x = 4,\ y = 2,\ rdy = F\}$. This is also the case for $\sigma' = \{x = 4,\ y = 2,\ s = 0,\ rdy = F\}$. Consequently, σ and σ' can be treated identically when arguing about the states from the environments point of view. This is exploited in the definition of refinement given below.
End of example

For an interface variable x, $V(x)$ is expressed by means of operators and other interface variables. The variable y *appears in* $V(x)$ if y is among the interface variables that are used to express $V(x)$. Since $V(x).\sigma$ is supposed to determine whether the environment is allowed to observe x all the variables that appear in the predicate $V(x)$ should be observable. To avoid circular dependencies, for example

$$V(x) = y \qquad V(y) = x$$

it is sufficient to require that if y appears in $V(x)$, then $V(y)$ must be the constant true. $V(x)$ defaults to true, i.e. leaving out $V(x)$ implies that x always belongs to the observable part of the interface.

The possibility of exploiting the interface protocol relies on the cell and the environment to have the same understanding of which part of the interface is observable in a given state. This is dealt with next.

3.2 Respecting an Interface Protocol

Let $ns(\sigma)$ denote the set of next states that the environment can cause by performing one state transition from $\mathcal{T}_E$ started in σ.

$$ns(\sigma) = \{\sigma' \in \mathcal{S}_E \mid (\sigma, \sigma') \in \mathcal{T}_E\}$$

Informally, the environment *depends on* a variable, y, in a state, σ, if the value of y influences the changes of other variables caused by the environment. This is denoted $dep(y, \sigma)$. Let $\tilde{y}$ denote the set $\mathcal{V}_E \setminus \{y\}$.

$$dep(y, \sigma) \ = \ \exists v \in val(y) : ns(\sigma) \downarrow \tilde{y} \neq ns(\sigma[v/y]) \downarrow \tilde{y}$$

Consider any cell, C, with an interface protocol, V, and environment, E. Let (σ, σ') denote any state transition that E can perform. The environment is not allowed to depend on a variable y in σ when $V(y).\sigma$ is false. This leads to the following condition, **ND** (no dependency)

$$\textbf{ND} \qquad \forall y \in \mathcal{E}_E : dep(y, \sigma) \Rightarrow V(y).\sigma$$

Furthermore, the environment must not turn $V(y)$ true or change the value of unobservable interface variables. This results in the condition **NC** (no change):

$$\textbf{NC} \qquad \forall y \in \mathcal{E}_E : \neg V(y).\sigma \Rightarrow \neg V(y).\sigma' \ \wedge \ \neg V(y).\sigma' \Rightarrow y.\sigma = y.\sigma'$$

When these two conditions are fulfilled, E is said to *respect* V.

4 Refinement

The notion of refinement described below is based on the information about the environment, captured by interface protocols.

4.1 Definition of Refinement

Let C denote the concrete and A the abstract design. Let V denote the interface protocol of A and C, and let E denote an environment that respects V. Refinement as defined below ensures that for any trace of $C||E$ there is a similar trace of $A||E$, where similar means that there are no differences with respect to the observable variables.

The aim is to compare only the observable part of the traces. This is obtained using $\downarrow_V$ that projects away state variables that are not observable. Given a cell C and $\sigma \in \mathcal{S}_C$

$$\sigma \downarrow_V = \sigma \downarrow \{ v \in \mathcal{E}_C \mid V(v).\sigma \}$$

Applying $\downarrow_V$ to a trace corresponds to applying $\downarrow_V$ to each of the states in the trace.

Example: Multiplier (continued). Below, each column denotes a state, and the succeeding columns make up a part of a trace, t. In the presence of the interface protocol $V(s) = rdy$, $t \downarrow_V$ is the part of the trace between the two lines.

s	⋯	2	2	2	0	3	6	9	12	12	⋯
rdy	⋯	T	T	F	F	F	F	F	F	T	⋯
x	⋯	1	4	4	4	4	4	4	4	4	⋯
y	⋯	2	3	3	3	3	3	3	3	3	⋯

End of example

Having projected away the parts of the traces that are not observable, leaves the traces with adjacent duplicates, where the original traces only differ on non observable state variables. Between two observable state changes the abstract and the concrete designs might make a different number of state transitions. In the definition below this is compensated for by allowing traces to have duplicates (section 2). This leads to the following definition of refinement:

Definition 1. Let A and C denote cells with the same interface protocol, V, and E an environment, such that E respects V. C refines A in E iff

$$\forall t' \in \mathcal{B}_{C\|E},\ \exists t \in \mathcal{B}_{A\|E} :\ t' \downarrow_V \, = \, t \downarrow_V$$

Note that this definition is a generalization of trace inclusion, since leaving out interface protocols implies that $\downarrow_V$ only projects away local variables.

Example: Multiplier (continued). Let A and C denote the abstract and the concrete multiplier, respectively. Let E denote an environment. Consider the two traces $t \in \mathcal{B}_{A\|E}$ and $t' \in \mathcal{B}_{C\|E}$.

t:	s	⋯	2	2	2	2	2	2	2	2	12	⋯
	rdy	⋯	T	T	F	F	F	F	F	F	T	⋯
	x	⋯	1	4	4	4	4	4	4	4	4	⋯
	y	⋯	2	3	3	3	3	3	3	3	3	⋯

t':	s	⋯	2	2	2	0	3	6	9	12	12	⋯
	rdy	⋯	T	T	F	F	F	F	F	F	T	⋯
	x	⋯	1	4	4	4	4	4	4	4	4	⋯
	y	⋯	2	3	3	3	3	3	3	3	3	⋯

While the computation of s takes place, rdy is false and the interface protocol $V(s) = rdy$ ensures that s is projected away from the traces. This means that when rdy is false, there is no difference between $t' \downarrow_V$ and $t \downarrow_V$. When s has reached the value of $x * y$, rdy is set true, and s is again included in the traces.

Still no difference is found between $t' \downarrow_V$ and $t \downarrow_V$, since now s has the same value in both the concrete and the abstract trace. This illustrates the advantage of considering only the observable part of the interface.
End of example

5 Verification

The definition of refinement given in the previous section is not directly useful for formal verification, since it is concerned with infinite traces. This section provides a verification technique that is suitable for mechanizing the verification of refinement. Below, the verification is carried out using a theorem prover. Often when theorem provers are used, the designs are specified directly in the logic of the theorem prover. In the present work another approach is taken. In order to avoid obscuring the descriptions with theorem prover specific details, SYNCHRONIZED TRANSITIONS [11] is chosen as description language.

5.1 SYNCHRONIZED TRANSITIONS

A subset of SYNCHRONIZED TRANSITIONS is used. In SYNCHRONIZED TRANSITIONS, a cell, C, consists of (1) declarations of the local and interface variables, (2) an initialization, $Init(C)$, (3) a set of transitions, $Tr(C)$, (4) an invariant, $Inv(C)$, and (5) an interface protocol, V.

The variable declarations state the name and type of the variables, thereby identifying the set of states, $\mathcal{S}_C$.

The initialization is a predicate on the state variables. It determines the initial values of state variables. The initialization has the form:

$$v_1 = val_1 \ \wedge \ v_2 = val_2 \ \wedge \ v_3 = val_3 \ \wedge \ \cdots$$

where v_1, v_2, v_3 are variables and val_1, val_2, val_3 are suitably typed values. Any state in $\mathcal{S}_C$ that fulfills this predicate can be an initial state. The initialization is related to the underlying transition system in the following way

$$\mathcal{I}_C = \{ \ \sigma \in \mathcal{S}_C \mid Init(C).\sigma \ \}$$

Transitions describe the state changes that the cell can perform. Each transition consists of a precondition, p, and a multi assignment. A precondition is a boolean typed expression. The multi assignment consists of a list of variables and an equally long list of expressions. If p is true in a state, the multi assignment can be executed in that state. Executing the multi assignment, simultaneously assigns the value of the expressions $e_1, e_2, \ldots, e_n$ to the variables $l_1, l_2, \ldots, l_n$. Syntactically, a transition looks like this:

$$\ll p \ \rightarrow \ \ l_1, \ l_2, \ \ldots, \ l_n \ := \ e_1, \ e_2, \ \ldots, \ e_n \gg$$

Given a transition, t, $exprs(t)$ denotes the list of expressions $< p, e_1, e_2, \ldots, e_n >$, and $exprs(t).\sigma$ denotes the list of values $< p.\sigma, e_1.\sigma, e_2.\sigma, \ldots, e_n.\sigma >$. Given states

σ, σ' and transition t, $\sigma \rightarrow_t \sigma'$ denotes that the precondition evaluates to true in σ, and that executing the multi assignment in σ results in σ'. This is referred to as executing the transition. σ and σ' are referred to as the pre and post states, respectively, for the execution of t. Given a state, σ, several transitions may have a precondition that evaluates to true in σ. In this case, the transition to be executed is selected nondeterministically. This means that the set, $\mathcal{T}_C$, used to define the underlying transition system is defined by

$$\mathcal{T}_C = \{ (\sigma, \sigma') \in (\mathcal{S}_C \times \mathcal{S}_C) \mid \exists t \in Tr(C) : \sigma \rightarrow_t \sigma' \}$$

An invariant, $Inv(C)$, is a predicate on the state variables. It holds in any state of any trace of C. In 5.3 the use of invariants in the verification is explained.

Example: Multiplier. An abstract and a concrete multiplier written in SYNCHRONIZED TRANSITIONS are listed below.

Since transitions are selected nondeterministically, no assumptions can be made in advance about the order of the execution of transitions. In case of the multiplier, this is handled in the following way: When the environment has supplied the cell with the arguments x and y, rdy is set false to make the computation start, and when the computation has finished, the cell notifies this by setting rdy true.

```
CELL $mult(x, y, s : \text{INTEGER}\ ; rdy : \text{BOOLEAN}\ )$
INTERFACE PROTOCOL $V(s) = rdy$
INVARIANT $x \geq 0 \land y \geq 0$
BEGIN
  $\ll \neg rdy \rightarrow s, rdy := x * y, \text{TRUE} \gg$
END $mult$
```

This abstract design states that s is assigned the value of $x * y$ in one state transition. To implement multiplication a number of finer grained operations are combined: s is set to 0, and y is added to s x times.

```
CELL $mult(x, y, s : \text{INTEGER}\ ; rdy : \text{BOOLEAN}\ )$
INTERFACE PROTOCOL $V(s) = rdy$
INVARIANT $x \geq 0 \land y \geq 0$
STATE
  $comp : \text{BOOLEAN}$
  $xl, s' : \text{INTEGER}$
INITIALLY $comp = \text{FALSE}$
BEGIN
  $\ll \neg rdy \land \neg comp \rightarrow comp, s, xl, s' := \text{TRUE}\ , 0, x, s \gg$
  $\ll comp \land xl > 0 \rightarrow xl, s := xl - 1, s + y \gg$
  $\ll comp \land xl = 0 \rightarrow rdy, comp := \text{TRUE}\ , \text{FALSE} \gg$
END $mult$
```

If rdy and $comp$ are false, the computation can be started by the first transition: s is set to 0, xl gets the value of x and $comp$ gets true. $comp$ being true denotes

that the computation is in progress. In the second transition, xl is decremented each time s is increased by y. The third transition deals with the situation where the computation of s has finished ($xl = 0$). This is notified by setting rdy true. s' has no influence on the computation. It is included for verification purposes. This is explained in 5.5.
End of example

5.2 Stepwise Verification

As explained previously, the definition of refinement is expressed in terms of infinite traces. In order to avoid arguing about infinite traces, stepwise verification is used. Stepwise verification breaks the verification task into a number of independent steps. This is done by focusing on the parts of the design that determine the traces, namely the initialization and the transitions. Informally, the stepwise verification of refinement consists of

1. Proving that the abstract and the concrete designs have corresponding initializations.
2. Proving that each state change caused by the concrete design can be explained in terms of the abstract design.

Stepwise verification is used in for example [1, 5, 7, 11], however, due to differences in the view of the interface, their proof obligation for each concrete state transition, differ from the proof obligation given below.

Stepwise verification differ from the approach in [6] and the work on temporal abstraction in [9] in the following way: In their work, the concrete state transitions are not considered one at a time; instead sequences of concrete state transitions corresponding to one abstract state transition are considered.

5.3 Exploiting Invariants in Stepwise Verification

Considering transitions one at a time, it is not possible to utilize information about the history of previously executed transitions. This implies that when a transition is considered, the state in which the transition is executed, is only identified by the precondition being true in that state. This means that unreachable states are also considered. Consequently, a verification could fail because the property that is attempted verified could be violated from an unreachable state.

Unreachable states can be excluded from consideration by establishing an invariant that rules out these states. States are only considered in the verification if they fulfill this invariant. Below, it is assumed that given a cell and its environment, it has been verified that neither the cell nor the environment violates the invariant of the cell. In [10] it is described how this is verified.

Example: Multiplier (continued). In the concrete multiplier the following invariant holds:

$$comp \Rightarrow (\neg rdy \wedge s + xl * y = x * y)$$

Consider for example the execution of the second transition, t_2, of the concrete multiplier:

$$\ll comp \wedge xl > 0 \rightarrow xl, s := xl - 1, s + y \gg$$

Looking at t_2 in isolation, it is impossible to tell the value of rdy in states where t_2 can be executed. Taking the invariant into account, it is easy to conclude that rdy is false in states where t_2 can be executed. Below, invariants are used to provide this kind of information.
End of example

5.4 Refinement Mappings

The verification technique ensures that the changes of the observable part of the interface caused by concrete transitions can also be caused by the abstract transitions. When a transition makes changes to the observable part of the interface, the changes might be based on local variables. Consider for example the abstract transition, t_a, $\ll i := la \gg$ and the concrete transition, t_c, $\ll i := lc \gg$. Assume that i is an observable interface variable, and la and lc are local variables. It is not possible to conclude that the execution of t_c corresponds to the execution of t_a unless it is known that la and lc have corresponding values. This implies that a mechanism is needed for relating the variables of the two designs. For this purpose *refinement mappings* [1, 7] are used.

A refinement mapping, R, is a mapping from the state space of the concrete design to the state space of the abstract design.

$$R : S_C \mapsto S_A$$

Refinement mappings as described in [1, 7] are the identity on interface variables. This requirement can be relaxed by using interface protocols: *It is only required that R is the identity on an interface variable, x, in states where $V(x)$ holds.*

The relaxed requirement to R is used to hide changes of unobservable interface variables. For each interface variable, v, with $V(v) \neq$ TRUE a verification variable, v' is introduced. v' is used to hold the last observable value of v while v is unobservable. The part of the refinement mapping concerning v is constructed in the following way.

$$v.R(\sigma) = \text{IF } cond.\sigma \text{ THEN } v'.\sigma \text{ ELSE } v.\sigma$$

cond is a boolean valued expression, and is expressed in terms of local variables of C. The refinement mapping states that the abstract value of v is interpreted as the value of v' if *cond* is true and as the value of the concrete v if *cond* is false. For this to be sound it is required that when the interface protocol for v is true,

R is the identity on v. As explained in 5.3 the requirements to R can be further relaxed by considering only states where the invariant holds. This results in the following requirement to R.

$$\textbf{RC} \qquad \forall v \in \mathcal{E}_C, \forall \sigma \in \mathcal{S}_C : \mathit{Inv}(C).\sigma \land V(v).\sigma \;\Rightarrow\; v.\sigma = v.R(\sigma)$$

This property is exploited in the verification technique stated below.

Example: Multiplier (continued). The refinement mapping for the multiplier is:

$$\begin{aligned} x.R(\sigma) &= x.\sigma \\ y.R(\sigma) &= y.\sigma \\ s.R(\sigma) &= \text{IF } \mathit{comp}.\sigma \text{ THEN } s'.\sigma \text{ ELSE } s.\sigma \\ \mathit{rdy}.R(\sigma) &= \mathit{rdy}.\sigma \end{aligned}$$

When the computation of s starts, *comp* gets true and s' gets the value of s. This causes $s.R(\sigma)$ to remain unchanged. This means that the refinement mapping hides the changes of s until the computation of s has finished. By then, *comp* gets false, and the value of $s.\sigma$ $(= x.\sigma * y.\sigma)$ is mapped to $s.R(\sigma)$.

For x, y, and *rdy*, **RC** holds trivially. In the presence of the interface protocol $V(s) = \mathit{rdy}$ and the invariant $\mathit{comp} \Rightarrow \neg\mathit{rdy}$, the proof obligation for s is

$$((\mathit{comp}.\sigma \Rightarrow \neg\mathit{rdy}.\sigma) \land \mathit{rdy}.\sigma) \Rightarrow s.\sigma = s.R(\sigma)$$

This is easily proved.
End of example

5.5 Verifying Refinement

This section presents a verification technique that ensures the notion of refinement in definition 1. It is important to note that the verification technique is intended for mechanized verification.

The part of the verification technique concerning the initialization is called **INIT**

$$\textbf{INIT} \qquad \forall \sigma \in \mathcal{S}_C : \mathit{Init}(C).\sigma \Rightarrow (\mathit{Init}(A).R(\sigma) \land \forall v \in \mathcal{E}_C : v.\sigma = v.R(\sigma))$$

This ensures that all initial states of the concrete design, when mapped to the state space of the abstract design, fulfill the initialization predicate of the abstract design, and that initially nothing is hidden by the refinement mapping.

The part of the verification technique concerning the transitions is called **STEP**

$$\textbf{STEP} \qquad \begin{array}{l} \forall t_C \in \mathit{Tr}(C), \sigma, \sigma' \in \mathcal{S}_C : \\ \mathit{Inv}(C).\sigma \;\land\; \sigma \rightarrow_{t_C} \sigma' \Rightarrow \\ R(\sigma) = R(\sigma') \lor \;\exists t_A \in \mathit{Tr}(A) : R(\sigma) \rightarrow_{t_A} R(\sigma') \end{array}$$

This ensures that any execution of a concrete transition can be explained either as the execution of an abstract transition $(\exists t_A \ldots)$ or as though no change has taken place ($R(\sigma) = R(\sigma')$). **INIT** and **STEP** can be combined to verify that two cells are in accordance with definition 1 using the following verification technique:

Verification technique: *Let A and C denote cells with the same interface protocol, V, and E any environment, such that E respects V. Verifying* **INIT** *and* **STEP** *ensures that C refines A in E.*

As mentioned in 5.3, it is assumed that both C and E maintain the invariant of C. Consequently this assumption does not appear explicitly in the verification technique.

A soundness proof for the verification technique has been carried out, however, it is not given here.

At first glance, the verification technique does not seem only to be concerned with the observable part of the interface. This concern, however, is taken care of by the refinement mapping. When a change of a non observable interface variable is made, the refinement mapping can hide the change. This is illustrated by the following example.

Example: Multiplier (continued). For the multiplier, the verification technique identifies four proof obligations; one for the initialization, and one for each concrete transition.

No initialization is present in the abstract multiplier. This ensures that $Init(C).\sigma \Rightarrow Init(A).R(\sigma)$ in any initial state, σ. For x, y and rdy, the refinement mapping is the identity. In the concrete design *comp* is initially false. This ensures that $s.\sigma = s.R(\sigma)$ in all initial states.

When the first transition, t_1, is executed, s is changed. Since *comp* gets true and s' gets the previous value of s, the change of s is hidden by the refinement mapping from the previous example. Consequently, the execution of t_1 is justified by the clause $R(\sigma) = R(\sigma')$ in **STEP**.

Each time the second transition, t_2, is executed, the change of s is still hidden by the refinement mapping, since *comp* and s' are unchanged. This means that each execution of t_2 is justified by the clause $R(\sigma) = R(\sigma')$ in **STEP**.

When the third transition, t_3, is executed, *comp* gets false. This means that the refinement mapping no longer hides the change of s. The invariant (from the example in 5.3) ensures that when $xl = 0$, s equals $x * y$. Consequently, the execution of t_3 corresponds to the execution of the transition in the abstract design.
End of example

5.6 Respecting the Interface Protocol

In section 3.2, respecting the interface protocol is defined. In this definition the environment is represented by the set of state transitions that it can make. Respecting the interface protocol is stated in terms of this representation. Given the SYNCHRONIZED TRANSITIONS description, a condition is stated below to ensure that the interface protocol is respected.

Considering the execution of a transition, t, $\sigma \rightarrow_t \sigma'$, condition **NC** can be applied directly to σ and σ'. However, condition **ND** must be rephrased.

A transition, t is dependent on a variable y in σ if the value of y influences any expression of t, i.e.

$$dep(t, y, \sigma) = \exists v \in val(y) : exprs(t).\sigma \neq exprs(t).\sigma[v/y]$$

Consequently, the resulting proof obligation is

$$\begin{array}{l} \forall t_E \in Tr(E), \sigma, \sigma' \in \mathcal{S}_E : \sigma \rightarrow_{t_E} \sigma' \Rightarrow \\ \forall v \in \mathcal{E}_E : (dep(t_E, v, \sigma) \vee V(v).\sigma') \Rightarrow V(v).\sigma \ \wedge \ \neg V(v).\sigma' \Rightarrow v.\sigma = v.\sigma' \end{array}$$

Note that respecting the interface protocol is preserved during refinement. This means that if the environment is refined, it still respects the interface protocol. Note also that the verification technique for respecting the interface protocol is independent of the refinement mapping, R. This means that if the concrete design is further refined, probably with a new refinement mapping, R'. If R' does not violate **RC**, the environment still respects the interface protocol. Consequently, it is only necessary to verify that the environment respects the interface protocol once, namely at the most abstract level.

A Modular Approach. In SYNCHRONIZED TRANSITIONS, the designs can be described in a modular way using cells. The modularity can be exploited when proving that the environment maintains the interface protocol. The condition for ensuring that the environment respects an interface protocol quantifies over all transitions. When the environment is extensive, and when several interface protocols are present, it can be laborious to verify that all transitions in the environment respects each of the interface protocols. In [10], an approach is described for verifying certain safety properties. The approach exploits the modularity of the designs to reduce the number of proof obligations. The approach can be generalized to verify that the environment respects an interface protocol.

5.7 Preservation of Safety Properties

In the preceding sections it has been explained how refinement using interface protocols can be used to justify replacing one abstract cell by another concrete cell in an environment. The correctness of this follows from the soundness proof mentioned in section 5.5. Though the correctness focuses on safety properties, the preservation of these is further illustrated below.

Assume that the abstract cell maintains a safety property that is vital for the environment. For example that an arbiter does not grant a privilege to two clients at the same time. If the notion of refinement is trace inclusion on the externally visible behaviors this safety property is also maintained by the concrete design.

If the notion of refinement includes the use of interface protocols as described previously, the concrete cell might cause externally visible behaviors that cannot be caused by the abstract design. However, the refinement using interface protocols implies trace inclusion for the observable part of the interface. For environments that respects the interface protocol, it follows from the proof mentioned in 5.5 that trace inclusion for the observable part of the interface is sufficient. Below, this is illustrated by the multiplier example.

If the multiplier is always provided with non-zero arguments, the safety property $s \neq 0$ holds for the abstract multiplier. For the concrete multiplier, this is not the case. Consider a transition, t, that placed in the environment of the multiplier can detect the difference between the abstract and the concrete multiplier.

$$\ll s = 0 \rightarrow error := \text{TRUE} \gg$$

If t is found in the environment of the abstract multiplier, *error* will never be true, while in the environment of the concrete multiplier, *error* might become true. However, *t does not respect the interface protocol*, so the violation of the safety property does not indicate a lack of soundness in the use of interface protocols.

This illustrates that if a safety property is not preserved, it is too strong for an environment that relies on the safety property to respect the interface protocol. This means that all safety properties that are relevant for the environment are preserved.

5.8 Mechanization

Using the verification technique introduced above, it can be proved that one design refines another. A prototype tool is developed that turns SYNCHRONIZED TRANSITIONS descriptions into an axiomatization in the logic of the theorem prover LP, The Larch Prover [3]. The tool is based on a translator written for proving invariants for SYNCHRONIZED TRANSITIONS [10]. Each of the constituent parts of the designs (i.e. transitions, invariants, initializations, variable declarations, etc.) are translated into corresponding LP constructs. The tool also generates proof obligations for proving refinement. The multiplier used as example in this section and a number of examples from [11] have been verified using the tool and the theorem prover.

The verification technique stated above turns out to be suitable for a theorem prover. The proof is by cases on the concrete transitions. This means that the proof consists of several limited subproofs. Each of these subproofs assumes that one concrete transition has been executed. At any stage of the proof, it is therefore straightforward to relate the current stage of the proof to a specific transition in the concrete design. Consequently, it is easy to interact if manual assistance is needed. Furthermore, changes in a few transitions allow for most of the proof to be reused.

6 Example - The Tamarack Microprocessor

The verification of an implementation of a simple microprocessor is described in [6]. This verification has been redone using the translator and theorem prover described in the previous section.

Below, a part of the verification is described. Partly to illustrate the applicability of the verification technique of the previous section, partly to illustrate the differences between the present approach and the approach in [6].

6.1 Abstract Description

The CPU executes instructions located in a memory (*mem*). A program counter (*pc*) points out the next instruction to be executed. The instructions include addition, subtraction, (conditional) jumping, and loading and storing data in the memory. Below, focus is on the execution of a store instruction.

When *pc* points out a store instruction, the store instruction identifies the location to be changed. The value to be stored in this location is contained in the register *acc*. Let *ST* denote a function that given *pc*, *acc* and *mem* as arguments returns the updated *mem* where the location pointed out by the store instruction contains the value of *acc* and the rest of the locations are unchanged. While *mem* is updated, *pc* is incremented to point out the next instruction to be executed. This means that the abstract description of the store instruction is the following:

$$\ll mem, pc := ST(pc, acc, mem), pc + 1 \gg$$

6.2 Concrete Description

The concrete design describes a microcode implementation of the CPU. In the microcode implementation, it is not possible to update the memory and to increment the program counter in the same state transition. Consequently, *mem* is updated first, and afterwards *pc* is incremented. In order to emphasize on concerns relevant to the use of interface protocols, the part of the concrete design, concerning the store instruction, is less detailed than the original description in [6].

$$\ll mpc = 17 \rightarrow mem, mpc := ST(pc, acc, mem), 18 \gg$$
$$\ll mpc = 18 \rightarrow pc, mpc := pc + 1, 5 \gg$$

mpc denotes the microcode program counter. It ensures the correct order of the update of *mem* and *pc*. The net result of first updating *mem* and afterwards *pc* is the same as executing the abstract transition shown above.

6.3 The Problem

The crucial difference between the abstract and the concrete descriptions is that the changes of *mem* and *pc* in the concrete design do not happen in the same state transition. *mem* and *pc* are both interface variables. This means that the environment might observe different values on the interface of the concrete and the abstract design.

The Solution Using Interface Protocols. In the present approach this is handled by including a flag, *ready*, in the interface of both descriptions. *ready* is set false when *mem* is updated and set true when *pc* is incremented. An interface protocol

$$V(mem) = ready$$

is introduced. This prevents the environment from reading *mem* before *pc* is incremented.

A refinement mapping, R, is constructed. It hides the change of *mem* until *pc* is incremented. When *ready* gets false, the value of *mem* is assigned to a variable, *mem'*, and *mpc* is incremented. Using the refinement mapping described below, this makes $mem.R(\sigma)$ unchanged. When *pc* is incremented, *mpc* is set to 5 and *ready* gets true. At the abstract level it appears as though *mem* and *pc* are changed in the same state transition.

$$\begin{aligned}
acc.R(\sigma) &= acc.\sigma \\
ready.R(\sigma) &= ready.\sigma \\
pc.R(\sigma) &= pc.\sigma \\
mem.R(\sigma) &= \text{IF } mpc.\sigma = 18 \text{ THEN } mem'.\sigma \text{ ELSE } mem.\sigma
\end{aligned}$$

Given R and $V(mem) = ready$, the problem degenerates to proving refinement between the following cells:

```
CELL A(pc, acc : INTEGER ; mem : MEMORY; ready : BOOLEAN )
INTERFACE PROTOCOL V(mem) = ready
BEGIN
  ≪mem, pc, ready := ST(pc, acc, mem), pc + 1, TRUE ≫
  ≪ready := FALSE ≫
END A

CELL C(pc, acc : INTEGER ; mem : MEMORY; ready : BOOLEAN )
INTERFACE PROTOCOL V(mem) = ready
STATE
  mem' : MEMORY
  mpc : INTEGER
INVARIANT
  (mpc = 17 ⇒ ready)∧
  (mpc = 18 ⇒ (mem = ST(pc, acc, mem') ∧ ¬ready))
BEGIN
  ≪ mpc = 17 →
      mem, mpc, ready, mem' := ST(pc, acc, mem), 18, FALSE , mem≫
  ≪ mpc = 18 → pc, ready, mpc := pc + 1, TRUE , 5≫
END C
```

The proof is carried out using the tools described in the previous section.

Note that the variable *mem'* is a *verification* variable; after the verification has been completed it can be removed since it has no influence on the design (it is never read).

Solution in [6]. In contrast to the asynchronous descriptions above, the descriptions in [6] are synchronous. For both the abstract and the concrete descriptions, the values of the variables are described as functions of time. In the abstract description, the execution of one instruction takes one time unit, whereas in the concrete design several time units are necessary to complete one instruction. This means that the abstract time scale is more coarse grained than the concrete. This is dealt with using temporal abstraction, which is described in more detail in [9].

The concrete design is augmented with a boolean variable, *ready*. The notion of refinement ensures that at points of concrete time where *ready* is true, the concrete design is in a state that could also be caused by the abstract design. The concrete CPU cannot be used directly instead of the abstract CPU. To be able to replace the abstract CPU with the concrete in an environment, it is necessary to ensure that the environment only depends on the interface variables of the CPU when *ready* is true. This issue of composition is not explicitly dealt with in [6].

7 Conclusion

This paper has reported an attempt to generalize the notion of trace inclusion to a more liberal notion of refinement by taking the environment into account.

A verification technique has been stated for proving refinement for designs written in SYNCHRONIZED TRANSITIONS. A prototype tool has been developed for translating SYNCHRONIZED TRANSITIONS descriptions into the logic of an existing theorem prover, and for generating proof obligations for refinement.

Experience with a number of examples has shown the usefulness of both the liberal notion of refinement and the tools for mechanizing the verification.

Acknowledgements

Thanks to J.Frost, H.H.Løvengreen, A.P.Ravn and J.Staunstrup for comments.

References

1. Martin Abadi and Leslie Lamport. The existence of refinement mappings. Technical Report 29, Digital Systems Research Center, 1988.
2. Randal E. Bryant. Can a simulator verify a circuit? In *Formal Aspects of VLSI Design*, pages 125–136. North-Holland, 1985.
3. Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, Digital Systems Research Center, 1991.
4. Mike Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In *Formal Aspects of VLSI Design*, pages 153–177. North-Holland, 1985.
5. Bengt Jonsson. On decomposing and refining specifications of distributed systems. In *Lecture Notes in Computer Science, 430*, pages 361–385. Springer Verlag, 1990.

6. Jeffrey J. Joyce. Formal verification and implementation of a microprocessor. In Graham Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification and Synthesis*, pages 129–157. Kluwer Academic Publishers, 1988.
7. Leslie Lamport. The temporal logic of actions. Technical Report 79, Digital Systems Research Center, 1991.
8. Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151. ACM, 1987.
9. Thomas F. Melham. Abstraction mechanisms for hardware verification. In Graham Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification and Synthesis*, pages 267–291. Kluwer Academic Publishers, 1988.
10. Niels Mellergaard. *Mechanized Design Verification.* PhD thesis, Department of Computer Science, Technical University of Denmark, 1994.
11. Jørgen Staunstrup. *A Formal Approach to Hardware Design.* Kluwer Academic Publishers, 1994.