Forced Simulation: A Formal Approach to Component-Based Synthesis

Partha S. Roop A. Sowmya
School of Computer Science and Engineering
The University of New South Wales
Sydney, NSW 2052, AUSTRALIA
Fax: +612-9385-1814
proop,sowmya@cse.unsw.edu.au

S. Ramesh
Department of Computer Science and Engineering
Indian Institute of Technology
Bombay, India - 400 076
ramesh@cse.iitb.ernet.in

UNSW-CSE-TR-9901-March 1999

March 10, 1999

Contents

List of Figures

Abstract

Embedded systems are application-specific digital systems which are normally designed using a microprocessor along with a set of programmable hardware and software components. Component-based synthesis of these systems will lead to the reuse of a vast library of hardware and software components and also facilitate rapid prototyping. However component based synthesis is still low key, a primary reason being the lack of any systematic attempt at the development of automatic component identification algorithms.

In [21] an algorithm to map a design *function* to a *device* from a library of system-level components was proposed. However, it was not based on a formal setting and no proof of correctness was presented. In this paper, we propose a novel notion of simulation called *forced simulation* to formalize the correspondence between a function and a device. What distinguishes forced simulation from other techniques is the idea of *forcing* via an external interface, which can be automatically synthesized, and is useful for adapting the system level component to the given design functionality. We have proposed two different types of forced simulation depending on the handling of internal events.

Chapter 1

Introduction

Embedded systems are *application-specific digital systems* having wide ranging usage from small home appliances such as automatic ovens and washing machines to very sophisticated controllers for aircrafts and submarines. The core of these systems is a micro-controller or a microprocessor which is suitably interfaced to a set of programmable hardware and software components to achieve a specific task. Considering their widespread usage, automatic synthesis of these systems has been a focus of research [5, 9, 20].

Current synthesis tools for embedded systems partition the specification into primitive hardware and software functionalities. Following this, the hardware functions are realized as ASICs using logic synthesis and the software functions by software synthesis. Current approaches to synthesis thus completely ignore a huge set of programmable hardware as well as software components available in this domain. These include components such as ports, timers, I/O controllers, Analog to Digital Converters(ADCs) and Digital to Analog Converters(DACs) which have been designed, developed and tested by various vendors. Also, recently there has been a trend within software engineering towards component-based software development [17]. So, there is vast potential for the reuse of both programmable hardware as well as predesigned software programs. Component-based design, though well known in other engineering domains [16], is still very low key in embedded system synthesis. Component-based synthesis has not been uncommon, however, for lower level components such as ALUs [14].

A major reason for the lack of component reuse during synthesis is the paucity of a systematic attempt towards the development of suitable algorithms that can map a design *function* ($\mathcal{F}$) to a suitable *device* ($\mathcal{D}$) from a library of such components. Recently, Mitra et. al [21] proposed an algorithm for such a mapping. They proposed an automata-based language for describing the behaviours of $\mathcal{F}$ and $\mathcal{D}$ and a mapping algorithm based on language containment. However, this work suffers from several limitations. Firstly, it is not based on a formal setting and hence lacks any means of proving the correctness of the mapping algorithm. Secondly, the use of automata with terminal states to model nonterminating reactive behaviour is inappropriate and hence the corresponding choice of language containment as the basis of the mapping algorithm is also inappropriate. Finally, this algorithm completely ignores possible divergence via transitions triggered by internal events.

In this paper we formalize the problem of mapping $\mathcal{F}$ to $\mathcal{D}$ by a new simulation relation called *forced simulation* and propose a new component identification algorithm based on forced simulation. Formalization provides a theoretical handle for studying the problem in a much more general and complex setting involving internal as well as nondeterministic behaviours and also helps in establishing that the transformation from $\mathcal{F}$ to $\mathcal{D}$ is correct since the proposed algorithm exactly mimics a mathematical relation.

Several simulation techniques [19, 15, 1, 18, 8] have been proposed in the past to check if a low level implementation I is a simulation of a high-level specification S. Though they have been widely applied to the verification of both hardware [26] as well as software [23, 7], protocol verification [10, 8] and also to check process equivalence in process algebras [19, 15], they are not directly applicable to our problem, since the implementation I is a refinement of the specification S, and is not arrived from *adapting* a general implementation to a given specification, which is the essence of reuse.

Adaptation is a major requirement since the device may not correspond exactly to the specification. We propose an adaptation step to be performed on the device, and call it *forcing.* Forcing occurs, when the device has extra control signals compared to the function and these control signals need to be generated externally at appropriate points by an external *interface*, or when the device is multi-functional and an interface has to guide the device along the appropriate path that matches the specification. Both these cases are quite frequent because often the specification is under-specified compared to the device (since it is not expected that the specifier knows all internal details of the device and only gives an abstract specification); also, many system components are programmable and, by appropriate mode selection, may be programmed to behave differently. We now give an example that illustrates the idea of forcing.

1.1 Overview

In this section, we shall give a brief overview of our approach. We also motivate forcing with external labels alone via the following example.
Example 1:

Let us consider a multi functional component which can behave as a *down counter*, a *rate generator* or a *square wave generator*. This device, after initialisation, can be programmed to behave in one of the three modes by loading an appropriate mode word. An abstract behavioural description of such a device is shown in Figure 1.1 where we have the full behaviour of the *down counter*. A practical device very similar to the one described here is the *Intel programmable interval timer (8253)*. Also note that all signals involved in this example are assumed to be external. Note that we have used a labelled transition system (LTS) to represent the behaviours. The LTS is defined formally in the subsequent chapter. We shall use $\mathcal{F}$ and $\mathcal{D}$ to denote abstract behaviours of the function and the device and F and D to denote their respective LTSs.

Suppose, $\mathcal{F}$ is a *down counter* whose behaviour F is as shown in Figure 1.1. F decrements the value of a counter register *val* with every clock (*clk*) input until it receives a *stop* command to output the count value.

Since *dev-init* and *in-mode0* in D are external signals, the interface can generate them to place the device in state 2. A difference between F and the *down counter* in D is that the latter requires an extra control signal called *gate* to be set to low. This is an example where F is under specified (which occurs frequently as the specifier is not expected to know every internal detail of the device). Hence, for F to be simulated by D the interface must force all transitions of D from state 0 through to state 3. Once the interface has placed the device in state 3, simulation of F is guaranteed. A typical interface for the above example is also illustrated in Figure 1.1. The interface is a *state-based interface* and moves in *lock-step* with the function and device states and performs forcing when required.

To capture the correspondence between F and D none of the existing simulation relations such as Milner's bisimulation [19], Lynch and Vaandrager's forward and backward simulations [18], Abadi and Lamport's refinement [1] or the more recent ones such as [8] suffices. Hence, we propose a novel notion called forced simulation to capture the correspondence between $\mathcal{F}$ and $\mathcal{D}$. Two different types of forced simulations have been proposed in this paper. One of the relations assumes that both F and D are capable only of external actions and proposes a *state-based interface* which moves in lock-step with the states of F and D to perform forcing as well as signal

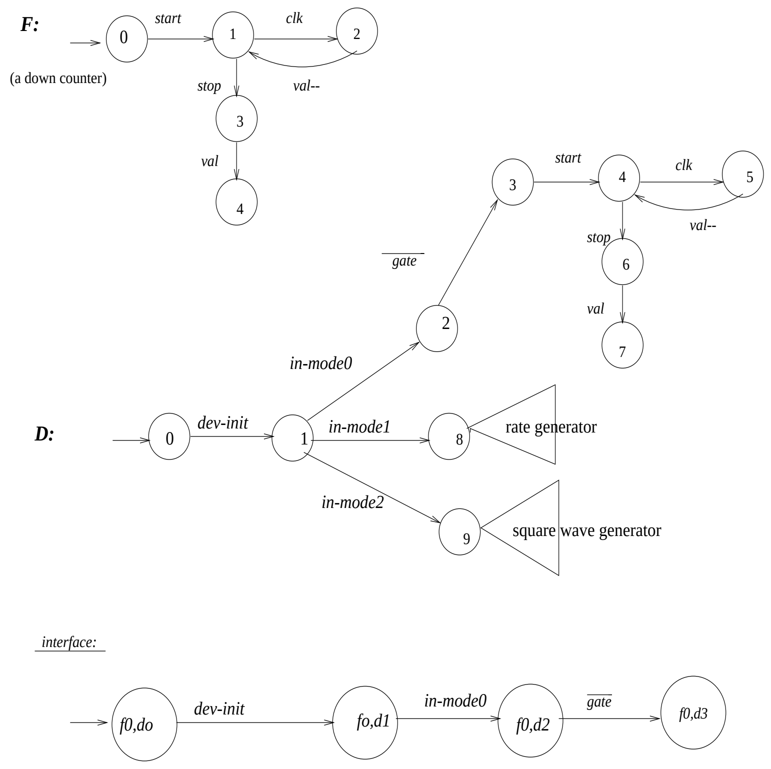

Figure 1.1: Multi-functional Device and simple forcing

mappings. The second one is a more general forced simulation that can handle both internal [19] as well as nondeterministic transitions in F and D in addition to the usual externally triggered transitions. Internal transitions are common in system level components since they are capable of various types of data operations which may trigger internal events such as *overflow, underflow, divide-by-zero error, time-outs*. An external interface cannot directly observe these events and thus has no control over them. Thus in the second and more general type of forced simulation, the interface proposed is a buffered interface that is based on the idea of *observational equivalence* [19], to be explained in the next chapter.

We first give the formal definitions for both these types of forced simulations. Then we propose an automatic component identification algorithm based on the second more general type of forced simulation. This algorithm, given the function specification F and device specification D, can say if a forced simulation relation exists between them. If such a forced simulation relation exists, then an interface can be automatically synthesized such that the device, together with the interface, simulates the function specification.

This report is organized as follows: in Chapter 2,we shall give the motivation behind forced simulation via another example and then formally define two types of forced simulation relations. In Chapter 3 we present an algorithm based on the more general forced simulation definition, which can be used for automatic component identification.The fourth Chapter reviews related literature and the final Chapter is devoted to some concluding remarks.

Chapter 2

Forced Simulation

This chapter is devoted to the presentation of two forced simulation relations, starting from a very simple one that defines simulation in the presence of external inputs only to the more general one where forced simulation is possible in the presence of arbitrary internal inputs of the device. In the previous section, we motivated forcing in the presence of external labels alone. At the outset, before presenting the definitions, we give an example with a more general notion of forcing in the presence of extra internal transitions.
Example: 2 This example illustrates a more complex type of forcing using a buffered interface in the presence of internal symbols. Let us now revisit the previous example with additional internal events in the down counter component as shown in Figure 2.1. The previous example ignored any *underflow* occurring as the count down progressed. In Figure 2.1 both the function and the device can have internal *underflow* when the counter underflows while decrementing its data register. However, this internal signal causes no problem to the simulation as it is common to both the function and the device.

Note also that while a mode word is being loaded in D, an internal *mode-err* might occur. So, at device state 1, there are two possibilities: either the mode word for *mode0* is loaded successfully and the device makes a transition to state 2 or an error is detected by the device, leading to the internal *mode-err* event happening. However, since from either of these states namely 1 and 8 there are paths triggered by the same sequence of external actions *in-mode0,gate* (a single forcing sequence) leading to states 3 and 11 of the device, simulation of F is guaranteed.

Nondeterministic transitions add another level of complexity to the simulation algorithm when they are triggered by a forced signal. In such a situation, the simulation algorithm must check to see that all the destination states out of any such transitions lead to convergent states where simulation is guaranteed. A detailed example later explains the handling of both internal and nondeterministic transitions.

The interface introduced in this example is a *buffered interface*, which has no knowledge of the triggering of internal or nondeterministic events. It has knowledge only of the current function state and the corresponding forcing sequence as a buffered input.

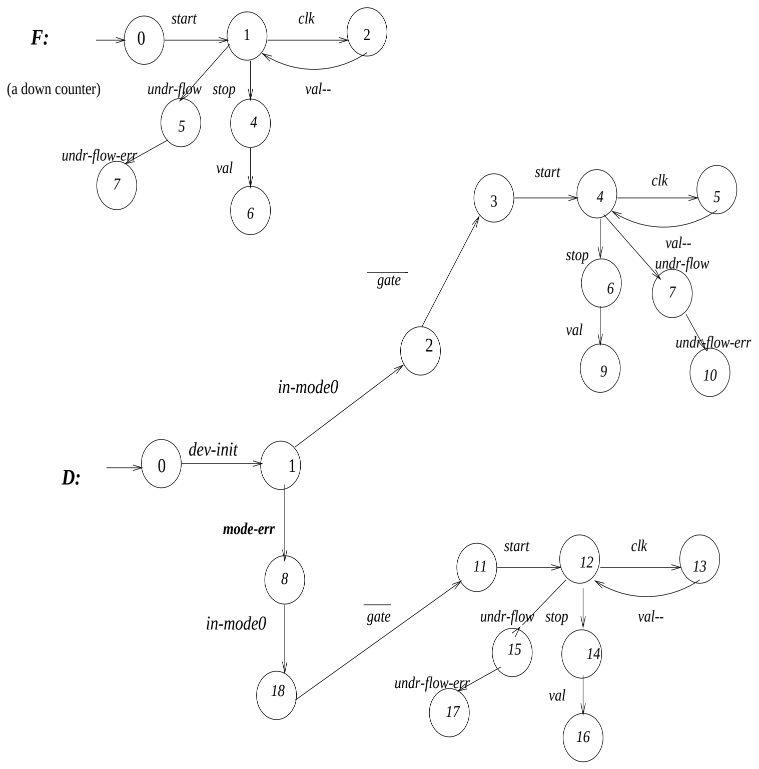

Figure 2.1: Forced Simulation via Buffered Interface

2.1 Formal Definitions

So far the notion of forced simulation, henceforth called *fsimulation*, has been presented informally. In this section, fsimulation will be formally developed.

We assume that all behaviours are represented as labelled transition systems similar to [19]. In our LTS, we assume that there are a set of internal inputs, in contrast to a single τ in CCS.

Definition 1:

A labelled transition system (LTS) is defined to be a tuple $(S, L, \{\xrightarrow{l}: l \in L\})$, where:

1. S is a set of states,
2. L is a set of transition labels, where $L = eL \cup lL$, where eL denotes a set of external labels and lL denotes a set of local (internal) labels,
3. $\xrightarrow{l} \subseteq S \times S$ for each $l \in L$.

We shall use symbols $a,\ b,\ c$ .. to range over external labels and symbols $i_1, i_2, ..$ to range over internal labels. Also, states will be denoted by s_1, s_2. In general, l^e will be used to denote any external label and l^i will indicate internal labels. We add the subscripts f and d to any of these symbols, if necessary, to make the context clear (of device $\mathcal{D}$ or function $\mathcal{F}$).

Problem Definition

Given LTSs F of a design function (a specification) $\mathcal{F}$ and D of a device $\mathcal{D}$, we wish to define a fsimulation relation $F \sqsubseteq_\Sigma D$, when $\mathcal{F}$ can be implemented by $\mathcal{D}$ using the notion of forcing. If an fsimulation relation exists between F and D, then we say that $\mathcal{D}$ can implement $\mathcal{F}$.

Informally, an fsimulation relation exists if for each function state there is a corresponding device state such that the device state *fsimulates* (denoted as $\sqsubseteq_\Sigma$) the function state. A device state fsimulates a function state if there is a successor convergent device state where the same behaviour as the given function state is elicited. A given successor device state is a convergent state of the current state if it can be reached from the device state by a path triggered by external labels alone. This is the simplest type of fsimulation, as formally defined below.

Simple fsimulation

In this definition we assume the following:

1. F and D have only external labels.
2. F and D are also deterministic
3. the interface that performs forcing is a state dependent interface, i.e, has knowledge of the current device as well as the function state.

Definition 2:

$$s_f \sqsubseteq_\Sigma s_d \overset{def}{=} \exists k \geq 0 : s_f \sqsubseteq_\Sigma^k s_d$$
$$s_f \sqsubseteq_\Sigma^0 s_d \overset{def}{=} subtree\_isomorphic(s_f, s_d)$$
$$s_f \sqsubseteq_\Sigma^{k+1} s_d \overset{def}{=} \exists a \in eL_d, \forall s'_d : s_d \rightarrow_a s'_d : s_f \sqsubseteq_\Sigma^k s'_d$$
$$subtree\_isomorphic(s_f, s_d) \overset{def}{=} \forall a, s'_f : (s_f \rightarrow_a s'_f \Rightarrow \exists s'_d : s_d \rightarrow_a s'_d \wedge s'_f \sqsubseteq_\Sigma s'_d)$$

Intuitively, this inductive definition says that a state s_d of D simulates a state s_f of F in $k+1$ steps provided there exists a transition via an external label from state s_d to state s'_d (the forced transition) of D such that s'_d simulates s_f in k steps. The base case is when k is zero and the simulation occurs in zero steps, when the subtrees of s_f and s_d are isomorphic. Thus, in this definition, the iteration index k indicates the maximum number of forcing steps required to reach the isomorphic state of the device, where the simulation of the function state happens directly.

The following example illustrates the idea.

Example 3:

Consider F and D as shown in Figure 2.2. In this example the state s_d converges to the state $s_{d'}$ via path $s_d \longrightarrow s_{d1} \longrightarrow s_{d2} \longrightarrow s_{d3} \longrightarrow s_{d'}$ such that at the state $s_{d'}$ of the device, the same behaviour as the function state s_f may be reproduced. Hence, an fsimulation relation exists between F and D. In Figure 2.2, the transitions marked with dotted lines, in parallel to normal transitions, indicate forced transitions.

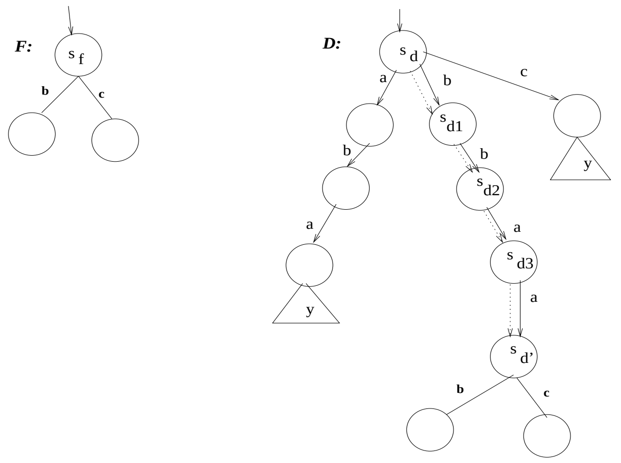

Figure 2.2: fsimulation example

Also note that the simulation relation unfolds in the following manner for the above example: $s_f \sqsubseteq^4_\Sigma s_d \rightarrow s_f \sqsubseteq^3_\Sigma s_d1 \rightarrow s_f \sqsubseteq^2_\Sigma s_d2 \rightarrow s_f \sqsubseteq^1_\Sigma s_d3 \rightarrow s_f \sqsubseteq^0_\Sigma s'_d \rightarrow subtree\_isomorphic(s_f, s'_d)$.

fsimulation via Buffered Interface This definition is a generalisation of the previous definition to incorporate the following:

1. F and D may be non-deterministic
2. F and D are capable of performing internal actions. Each has a set of internal actions and each internal action represents a different internal operation.
3. The interface is based on the concept of *observational equivalence* [19] i.e, the interface has no knowledge of the triggering of internal and nondeterministic transitions.

In this restricted scenario, fsimulation is possible if there exists a single sequence of external labels σ such that irrespective of the internal transitions or non-deterministic transitions of the device, the interface can place the device in a convergent state by using the symbols in σ. We thus obtain a buffered interface a la Example 2. The device can simulate the current function state in any one of these convergent device states. To distinguish between the two different types of fsimulation,the Definition 2 and the current one, we shall use the symbol $s_f \sqsubseteq_{\Sigma_b}$ since the interface assumed with this definition is a buffered interface.

Example 4:

Consider the example of F and D in Figure 2.3. By use of a single forcing sequence σ which is aab, the interface will place the device in one of the convergent states 6, 10, 11, or 12 where the same behaviour as s_f can be guaranteed.

Definition 3:

$$\begin{aligned}
s_f \sqsubseteq_{\Sigma_b} s_d &\overset{def}{=} \exists \sigma \in eL_d^*, \exists k \geq 0 : s_f \sqsubseteq^{\sigma,k}_{\Sigma_b} s_d \\
s_f \sqsubseteq^{a.\sigma,k+1}_{\Sigma_b} s_d &\overset{def}{=} (\exists s'_d : s_d \rightarrow_a s'_d) \wedge \\
&\qquad (\forall s'_d : s_d \rightarrow_a s'_d : s_f \sqsubseteq^{\sigma,k}_{\Sigma_b} s'_d) \wedge \\
&\qquad (\forall b \in lL_d, \forall s'_d : s_d \rightarrow_b s'_d : s_f \sqsubseteq^{a.\sigma,k}_{\Sigma_b} s'_d) \\
s_f \sqsubseteq^{\epsilon,k}_{\Sigma_b} s_d &\overset{def}{=} subtree\_isomorphic(s_f, s_d) \wedge internally\_terminal(s_d, s_f) \\
subtree\_isomorphic(s_f, s_d) &\overset{def}{=} \forall a, s'_f : (s_f \rightarrow_a s'_f \Rightarrow \exists s'_d : s_d \rightarrow_a s'_d \wedge s'_f \sqsubseteq_{\Sigma_b} s'_d) \\
&\qquad \wedge (\forall s''_d : s_d \rightarrow_a s''_d : s'_f \sqsubseteq_{\Sigma_b} s''_d)
\end{aligned}$$

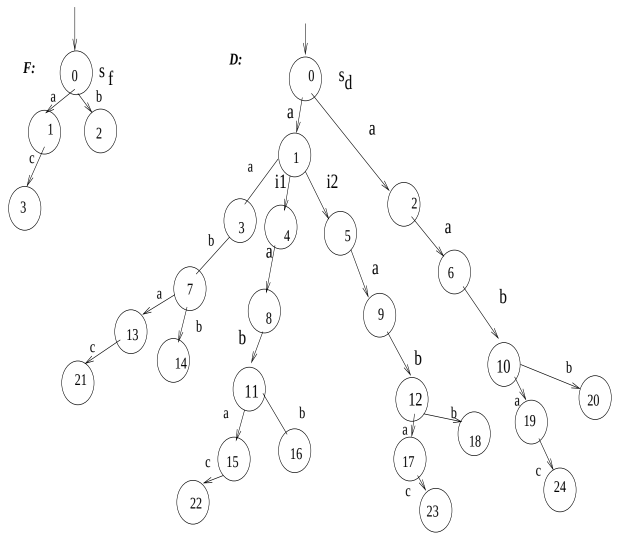

Figure 2.3: fsimulation with buffered interface

$$internally\_terminal(s_d, s_f) \overset{def}{=} \begin{array}{l}\text{the only internal transitions allowed out of } s_d\\ \text{are common internal transitions of } s_f\end{array}$$

In the above definition, a device state *fsimulates* a function state, provided there is a sequence of external actions, σ, and an iteration index, k such that the simulation is possible in at most k steps by using the symbols in σ as the forcing sequence. The base case of this inductive definiton happens when all the symbols in σ are consumed while iterating (i.e, when $\sigma = \epsilon$). The base case is satisfied when the subtrees under the function and device states are isomorphic and also there are no extra internal transitions from the device state (by which it could possibly diverge). The inductive step of this definition ($s_f \sqsubseteq_{\Sigma_b}^{a.\sigma,k+1} s_d$) says that the simulation is possible in $k+1$ steps using symbols in sequence $a.\sigma$ provided there exists a transition from the device state via the symbol a (this is necessary since there is no guarantee of the triggering of internal transitions from this node and this transition denotes a forced transition). Also for all transitions labelled by a, the prefix of σ, the destination states must simulate the function state using all symbols in σ (since the prefix a has been already consumed by the current transition) in at most k steps (note that this also handles nondeterministic transitions via a). Finally, if the device state has any internal transitions out of it, it must be the case that the destination states of all such transitions must simulate the function state using all symbols in $a.\sigma$ (since the prefix has not been consumed yet) in at most k steps.

Note that the length of the forcing sequence σ is always $\leq k$ since k indicates the maximum path length from s_d to any one of the successor states which satisfy the base case. In the best case, this path is labelled by all the symbols in σ, and $\|\sigma\| = k$. However, this path can also be interleaved with internal labels.

The unfolding of the iteration tree for the example in Figure 2.3 is shown in Figure 2.4. Note that in this figure each node is a four-tuple (s_f, s_d, σ, k), where s_f is the function state, s_d is the device state, σ denotes the forcing sequence and k denotes the iteration index.

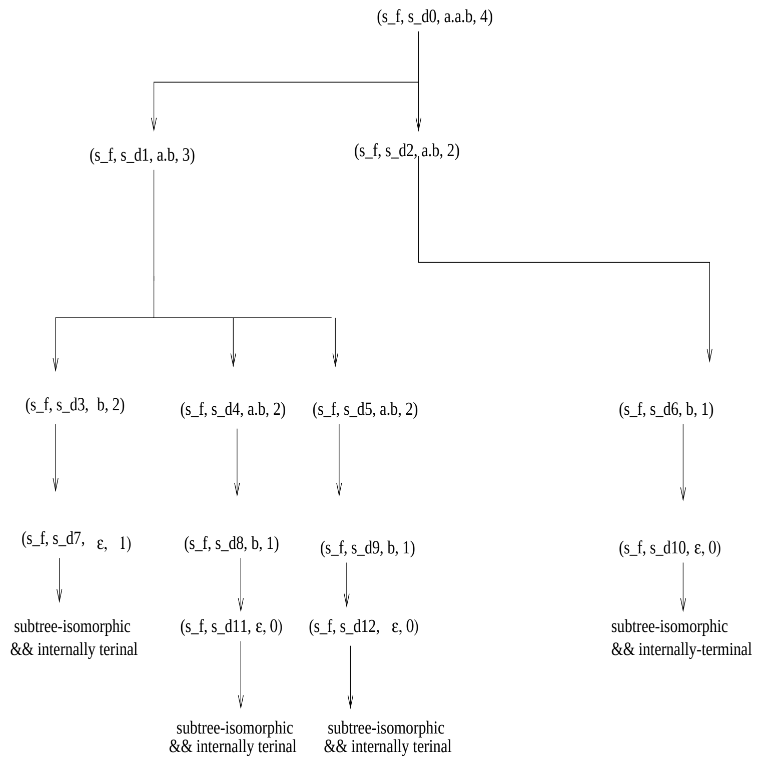

Figure 2.4: Unfolding of buffered forced simulation definition

Chapter 3

Component Identification Algorithm

In the previous chapter we proposed two definitions of *fsimulation*. In this chapter we present a component identification algorithm based on buffered fsimulation (Definition 3). The algorithm for simple fsimulation is a special case of this algorithm.

Given F and D as input, the $main()$ program in Figure 3.1 checks if a *fsimulation* relation exists. The algorithm starts with an universal simulation relation R which is the product of the state space of F and D. It then successively refines this relation until a fixed point is reached. The refinement process is performed via a function called $distinguish(s_f, s_d)$(Figure 3.2) that checks to see if this pair (s_f, s_d) cannot be distinguished. A pair cannot be distinguished, if a path σ exists from the device state s_d such that all successors of s_d via this path simulate s_f. This task of determining a correct σ is performed by traversing F by the $determineSucc()$ function (Figure 3.2). The actual checking of the simulation of s_f by all successors of s_d reached via a σ path is done by the $buffSimulation()$ function (Figure 3.3) which exactly mimics Definition 3. When this function reches the base case, it tests if the device state is isomorphic to the function state (in Figure 3.4) and is also internally terminal with respect to this state (in Figure 3.4).

If a pair of states (s_f, s_d) can be distinguished, then such a pair is removed from R and the repeat loop is reentered with the new R. Otherwise, the next pair in R is checked. This process is repeated until no pair in R can be distinguished. The final R is the desired simulation relation, provided an entry exists in R for each function state. Note that this algorithm is similar to the computation of bisimulation relations by using partition refinement

```
Main Program
$main(F, D)$
//$F$ and $D$ are the LTSs of the function and device respectively
//$s_{f0}$, $s_{d0}$ denotes the start states of $F$ and $D$ respectively
//the main program starts with an initial partition equal to the
//product of the state spece of $F$ and $D$ and successively refines
//this partition until the GFP is reached.
$R = state_f \times state_d$
//cross product of the states of $F$ and $D$
repeat
  change=false;
  for each state $(s_f, s_d)$ in R do
    change=$distinguish(s_f, s_d)$;
    if change then
      R=R-$\{(s_f, s_d)\}$
      exit the for loop;
      // to reevaluate the pairs in $R$
    endif
  endfor
until change=false
if $(s_{f0}, s_{d0}) \in R$ then
  return TRUE;
else
  return FALSE
endif
```

Figure 3.1: the main program for forced simulation using buffered interface

strategy [19, 15] and thus effectively handles cycles in LTSs.

Let NS_f and NS_d denote the number of states of F and D respectively. Also, let m denote the larger of the number of states and the number of arcs in D. The worst case complexity of the *distinguish*() function is of the order of m since it performs complete traversal of D in the worst case. The repeat loop in *main*() can iterate upto a maximum of $NS_f \times NS_d$. During each such iteration, the inner for loop can also iterate upto a maximum of $NS_f \times NS_d$. Hence, in the worst case the complexity of our simulation algorithm is of the order of $NS_f^2 \times NS_d^2 \times m$.

The complexity of the interface (its size) is another issue to be studied. We found that in the worst case the interface will also be of the order of

```
distinguish $distinguish(s_f, s_d)$
//$s_f$ denotes a function state
//$s_d$ denotes a device state
//this function returns true when the pair $(s_f, s_d)$ should not belong to R
$s'_d$=null;
$prev = s_d$
repeat
  $s'_d = determineSucc(s_d, prev)$
  if $s'_d$!=null then
    $\sigma = all\_external\_labelsInPath(s_d, s'_d)$
    $k = maxPathLength(D)$
    if $buffSimulation(s_f, s_d, \sigma, k)$ then
      return FALSE;
    else
      $prev = s'_d$;
    endif
  endif
until $s'_d$==null
return TRUE;
$determineSucc(devNode, SearchPoint)$
//tries to determine a successor of $devNode$
// that possibly satisfies the base case
//$SearchPoint$ is the device node from which search has to be continued,
//since $determineSucc()$ may be called a number of times
perform depth first traversal of $D$ with back tracking beginning with
$SearchPoint$
to determine a successor to $rootD$ that matches $rootF$
a match occurs if the successor node can match every transition in $rootF$
if no successor found satisfying above criteria then
  return null;
else
  return the successor node found;
endif
```

Figure 3.2: the *distinguish*() and *determineSucc*() functions

```
Buffered Forced Simulation
$buffSimulation(s_f, s_d, \sigma, k)$
//$k$ is an integer and $\sigma$ is the forcing sequence of external labels
if $\sigma{=}{=}\epsilon$ then
   //base case
   if $subtree\_isomorphic(s_f, s_d) \wedge internally\_terminal(s_d, s_f)$ then
      return TRUE;
   else
      return FALSE;
   endif
else
   //base case not yet reached; iterate further
   //$a$ denotes the first symbol of the forcing sequence
   //$\sigma_1$ is the sequence of remaining symbols
   $a$=head($sigma$);
   $\sigma_1$=tail($\sigma$);
   if $\nexists s_d \rightarrow_a s'_d$ then
      //no forcing path from $s_d$ via $a$; return failure
      return FALSE;
   endif
   for $\forall s'_d : s_d \rightarrow_a s'_d$ do
      //check that all destination states reached via $a$
      // simulate in k-1 steps
      if $buffSimulation(s_f, s'_d, \sigma_1, k-1)$ then
         continue;
      else
         return FALSE;
      endif
   endfor
   for $\forall i \in lL_d, \forall s'_d : s_d \rightarrow_a s'_d$ do
      //check that all internal transitions out of $s_d$ eventually
      // converge to a state where simulation is possible
      if $buffSimulation(s_f, s'_d, \sigma, k-1)$ then
         continue;
      else
         return FALSE;
      endif
   endfor
   return TRUE;
endif
```

Figure 3.3: The $buffSimulation()$ function

```
subtree-isomorphic $subtree\_isomorphic(s_f, s_d)$
for each $s_f \rightarrow_a s'_f$ do
   find $s_d \rightarrow_a s'_d$;
   if found then
      for all $s''_d : s_d \rightarrow_a s''_d$ do
         if $(s'_f, s''_d) \in R$ then
            continue;
         else
            return FALSE;
         endif
      endfor
   else
      return FALSE;
   endif
endfor
return TRUE;
$internally\_terminal(s_d, s_f)$ for each $s_d \rightarrow_i s'_d : i \in lL_d$ do
   find $s_f \rightarrow_i s'_f$
   if !found then
      return FALSE;
   else
      continue;
   endif
endfor
return TRUE;
```

Figure 3.4: The *subtree_isomorphic*() and *internally_terminal*() functions

the product of the number of function and the device states. Less complex interfaces can be designed, which will be specialisations of the above interface. We can think of several types of specialisations ranging from a device state dependent interface to state independent ones. The former is employed when forcing is necessary and is not deployed every time any two states need to be made equivalent. The latter is a specialisation, when no forcing is required to achieve the simulation.

Chapter 4

Prior Research

In this chapter, we review some related research from simulation literature and interface synthesis literature.

4.1 Review of Simulation Techniques

4.1.1 Bisimulation

One of the earliest notions of simulations, called Bisimulation, was proposed by Milner [19] for checking process equivalence. CCS (Calculus of Communicating Systems), a very neat algebraic formalism, was proposed by him for representing process communication and concurrency formally. It has a set of algebraic combinators to capture choice, parallel composition, prefix and hiding (also termed as restriction), which may be used to define complex processes in a bottom-up manner. The notion of bisimulation was developed to automatically check process equivalence. A major assumption was that process communication occurs by asynchronous handshake mechanism. However, in the algebra, such communication between two CCS processes was considered to be an internal action called τ. The usage of this single τ action as the only internal action led to the simplicity of the calculus. CSP, another algebraic formalism by Hoare was also proposed around the same time as CCS [11].

In CCS, two notions of bisimulation were proposed based on whether the internal action τ could be observed or not. In the first and more stronger notion, the internal action τ was treated identically to any other action and this lead to the notion of *strong bisimulation*. Informally speaking, two processes P and Q are strongly bisimilar if each action of the former could be matched by an identical action of the latter and the resultant states are also strongly bisimilar and conversely. Here, the internal action τ is observable and has also to be matched in both processes. Figure 4.1 depicts an example of two strongly bisimilar processes.

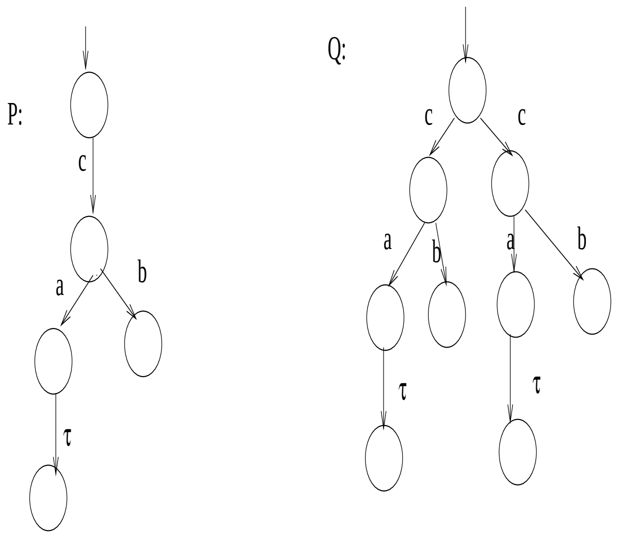

Figure 4.1: Example of Strongly Bisimilar Processes

In *weak bisimulation*, the restriction that each τ action of the processes have to be matched and that they are also observable is relaxed by the requirement that each τ action be matched by zero or more τ actions. This notion is thus based on the idea of observational equivalence where the internal action τ is unobservable. Figure 4.2 is an example of processes that are not strongly bisimilar but are weakly bisimilar.

Though CCS and bisimulation are interesting from the point of view of our application, they fail to capture two important requirements of our problem domain. Firstly, the components in our domain are capable of performing a set of data operations, each with a different semantics. Each of these operations is essentially an internal action. For example, the operation of incrementing a data register, or loading a register with a value or storing the value in a data input line in a given register represent different data operations and each is essentially an internal action of the device. Hence, a single τ can not be used to represent all of them. The second important lacuna of bisimulation and CCS is the lack of any operator using which forcing could be performed.

CCS has a hiding operator which provides global hiding of a given input

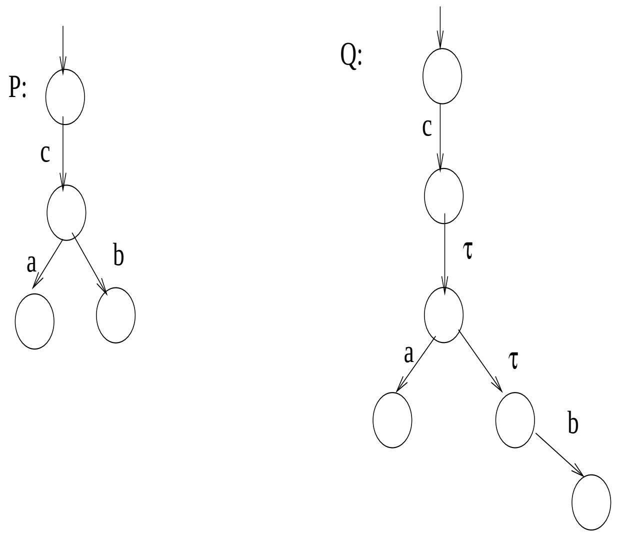

Figure 4.2: Example of Weakly Bisimilar Processes

symbol. Considering our informal example again (Example 1), using either the hiding operator or any other operator of CCS we can not make the two processes equivalent. For example, hiding of the symbol b in $\mathcal{D}$ will lead to the process as shown in Figure 4.3.The closest analogue of the forcing operator will be an operator that can provide state-based hiding and not the global hiding operator of CCS.

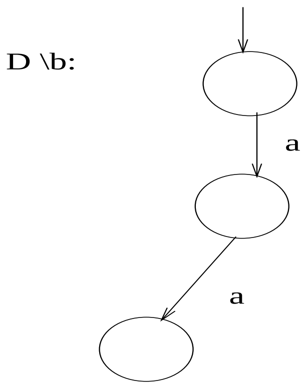

Figure 4.3: Example of Global Hiding

4.1.2 Refinement

The notion of refinement was first proposed by Abadi and Lamport [1] to prove that a lower level specification correctly implements a higher level one. The basic notion adopted was that of trace inclusion.

In [18] the above notion of refinement was extended to automata. A refinement from an automaton A to another automaton B is defined as a mapping from states of A to states of B such that:

1. image of every start state of A is also a start state of B and,
2. every transition in A has a corresponding sequence of transitions in B (including internal transition τ) that begins and ends with the images of the respective beginning and ending states of the given transition, and that has the same external actions.

Forward and backward simulations were later defined in [18] by Lynch and Vaandrager which are generalizations of refinement to incorporate more behaviours into the proof system. A primary difference between refinement and simulation is that refinement is a function from the states of the implementation to the states of the specification, whereas simulation is a relation. So, every refinement is a simulation but not conversely. Forward and backward simulations have been shown to provide a sound and complete proof method for checking trace inclusion between automata. A primary difference between the approach of Abadi and Lamport [1] and the approach taken by Lynch and Vaandrager [18] is that the former is based on the *state-based* approach where as the latter is based on the *action based* approach.

In all these approaches, auxiliary variables similar to those proposed by Owicki and Gries [23], often called history and prophecy variables, are introduced into the specification when simulation is trivially not possible to induce simulation in presence of these variables.

More recently, a new type of simulation, called *normed simulation* [8] has been introduced by Griffioen and Vaandrager by introduction of a *norm function.* In a normed simulation, each transition of a low level system may be simulated by at most one transition in the high level system, for any related pair of states. By introducing norm functions, the authors have proved that the checking for the existence of a normed simulation is decidable, which was not the case for checking the existing simulation relations. A norm function introduces a bound on the maximum number of τ transitions that the high level specification can take before it is forced to take a branch identical to that of the low level implementation.

All the above simulation relations discussed here are unsuitable for our application due to the same basic reasons that bisimulation can not be used directly: lack of multiple types of internal actions and the lack of forcing, both of which are essential to our application.

4.2 Forced Simulation versus Interface Process Generation

Interface process generation also referred to as interface synthesis is the task of automatically generating an interface process between incompatible protocols. There have been several attempts for the automatic generation of interface processes [3, 13, 6, 22, 2, 4] the starting point being the pioneering work by Borriello for automatic transducer synthesis [4]. In this work, timing diagrams of the two custom hardware was presented as input and the system produced the logic specification of the required glue logic automatically. However this approach did not handle data width mismatches between the two incompatible hardware. This limitation was overcome in a later work by Narayan and Gajski [22]. In this work the behaviours of the two incompatible blocks were represented in a *hardware description language* [25, 12] and then the algorithm verified if the two protocols were duals of each other. If they were not exact duals of each other then necessary extra control signals on either side were appropriately generated and the data width mismatches were also bridged by latching data values within local memory of the interface protocol and supplying the combined data values appropriately.

However, interface synthesis has several differences compared to our approach, as detailed below:

- We concentrate on device identification by a new simulation relation, where as interface process generation seeks to make communication possible between incompatible protocols of mapped or already identified system components (by the system designer). Protocols are incompatible if they are not exact duals of each other. In contrast, we are seeking to simulate equivalence (by forcing). So, the tasks are being applied to two different design steps: component identification versus component interconnection.

- In interface synthesis internal signals are of no significance since it is the external communication protocols that are being made compatible. In contrast, internal signals are significant in our application and for each internal operation performed by the function, the device must be capable of performing identical internal operations.

- There is no formal way of determining if in a given situation an interface process can be simulated. In contrast, $\mathcal{F}$ can be implemented by $\mathcal{D}$ whenever a forced simulation relation exists between the two.
- Just as in forcing, in interface synthesis also, extra control signals of any protocol are generated at appropriate times by the interface process. In addition, in the work in [22] data width mismatches can be handled by latching within the interface process. This task is not currently done by our algorithm.

Chapter 5

Conclusions

In this paper, we have proposed new simulation relations and developed algorithms based on them, which can be used for automatic component identification during embedded system synthesis. The proposed theory as well as the algorithm are novel from the perspective of simulation literature as well as in CAD literature.

Forced simulation gives rise to an external interface which together with the appropriate device, can simulate the given function. This is a very important development over existing simulation techniques, considering that many system level devices are multi-functional and hence the interface has to guide the device along its appropriate functionality to match the specification. Also the interface plays a vital role in generating extra control signals present in the device that are missing in the specification. The proposed component identification algorithm in this paper is based on the notion of observational equivalence of Milner [19] which requires that the interface has no knowledge of either internal or nondeterministic transitions of D. We have also developed more powerful algorithms by removing this restriction and assuming that the modified interface has complete state knowledge of both F and D. This new interface is similar to Milner's idea of *strong bisimulation* where even internal actions are considered observable. This algorithm has been developed and is being communicated separately.

The current algorithm, though a major starting point, has some limitations. Firstly it is based on a LTS setting. However, to test on real-life examples we must consider more powerful languages for modelling $\mathcal{F}$ and $\mathcal{D}$ since in practice these devices can be very complex being capable of performing many data and control transformations. We have recently proposed a language specifically tailored to component-based embedded systems [24].

Currently we are working on extending our algorithm to handle behavioural level descriptions in this language. Also, the interface synthesis algorithm, not presented in this paper, is being worked out. Finally, a proof of correctness of the approach is being developed based on the idea that when an fsimulation relation holds between F and D, there exists an interface I such that, F exhibits the same behaviour as $(I \parallel D)$, where $I \parallel D$ represents synchronous parallel composition of I and D.

The proposed techniques may also be applied to several other interesting problems such as re-engineering control software, verification and design adaptation. We have already implemented the above algorithms. Testing them on real life examples is a task to be performed. Also, extension of forced simulation to real-time systems will be considered in future.

Bibliography

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer science*, 82(2):253–284, 1991.

[2] J. Akella. *Input/Output performance modelling and Interface Synthesis in concurrently communicating systems.* PhD thesis, Carnegie Mellon University, 1991.

[3] A. Basu, R. S. Mitra, and P. Marwedel. Interface synthesis for embedded applications in a co-design environment. In *11th IEEE International conference on VLSI design*, pages 85–90, C, 1998.

[4] G. Borriello. *A new interface specification methodology and its application to transducer synthesis.* PhD thesis, University of California, Berkeley, 1988.

[5] P. Chou, R. Ortega, and G. Borriello. Synthesis of hardware/software interface in microcontroller based systems. In *ICCAD-92*, pages 488–495, 1992.

[6] P. Chou, R. B. Ortega, and G. Borriello. Interface co-synthesis techniques for embedded systems. In *ICCAD*, pages 280–287, 1995.

[7] R. Gerth. Foundations of compositional program refinement-safety properties. In *Stepwise refinement of distributed systems*, number 430 in LNCS, pages 777–808, 1989.

[8] D. Griffioen and F. Vaandrager. Normed simulations. In *Computer Aided Verification, CAV*, pages 332–344, 1998.

[9] R. K. Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems.* PhD thesis, Department of Electrical Engineering, Stanford University, 1993.

[10] L. Helmink, M. P. A. Sellink, and F. W. Vaandrager. Proof-checking a data link protocol. In *TYPES*, volume 806 of *LNCS*, pages 127–165, 1993.

[11] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[12] IEEE. *IEEE Standard VHDL Language Reference Manual*, 1988.

[13] T. B. Ismail, J. M. Daveau, K. O'Brien, and A. A. Jerraya. A system level communication approach for hardware/software systems. *Microprocessors and Microsystems*, 20(3):149–157, 1996.

[14] P. K. Jha and N. D. Dutt. High-level library mapping for arithmetic components. *IEEE Tr. on VLSI systems*, 4(2), 1996.

[15] P. C. Kanellakis and S. C. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86:43–68, 1990.

[16] H. Kopetz. Component-based design of large distributed real-time systems. In *14th IFAC Workshop on Distributed Computer Control Systems (DCCS'97)*, pages 171–177, Seoul, Korea, 1997.

[17] D. Krieger and R. Adler. Emergence of distributed component platforms. *Computer*, 31(3), 1998.

[18] N. Lynch and F. Vaandrager. Forward and backward simulations part i: Untimed systems. *Information and Computation*, 121(2):214–233, Sept. 1995.

[19] R. Milner. *Communication and Concurrency.* Prentice Hall International, 1989.

[20] R. S. Mitra, P. S. Roop, and A. Basu. An overview of mickey - a knowledge based hardware-software codesign framework for microprocessor-based systems. *Sadhana-Academy proceedings in Engineering Sciences*, 1996.

[21] R. S. Mitra, P. S Roop, and A. Basu. A new algorithm for implementation of design functions by available devices. *IEEE Transactions on very large scale integration (vlsi) systems*, 4(2):170–180, June 1996.

[22] S. Narayan and D. d. Gajski. Interfacing incompatible protocols using interface process generation. In *32nd Design automation conference*, pages 468–473, 1995.

[23] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[24] Partha S Roop and A. Sowmya. Hidden time model for specification and verification of embedded systems. In *10th Euromicro Workshop on Real-Time Systems*, pages 98–105. IEEE Computer Society Press, 1998.

[25] D. E. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic, 1991.

[26] P. J. Windley. Verifying pipelined microprocessors. Technical report, Laboratory of applied logic, Brigham Young University, 1995.