Forced Simulation and Lock-Step Interface: A Formal Approach to Automatic Component Matching

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-9903-June 1999

June 28, 1999

Contents

List of Figures

Abstract

Component-based synthesis of embedded systems will lead to the reuse of a vast library of hardware and software components and also facilitate rapid prototyping. However it is still low key, a primary reason being the lack of a systematic attempt at the development of automatic component identification algorithms. The main task of such an algorithm is to to map a design *function* to a *device* from a library of system-level components. 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 propose a new component matching algorithm based on forced simulation and also propose a technique for the automatic generation of the interface. Finally, a proof of soundness of the approach is presented, based on reducing the synchronous parallel composition of the interface and the device to Milner's weak bisimulation.

Chapter 1

Introduction

Several simulation techniques [1, 8, 4] have been proposed in the past to check if a low level implementation I is a simulation of a high-level specification S. The basic idea of simulation is that of trace inclusion; I is a simulation of S provided all traces of I are included in the traces of the S. Bisimulation [9, 7] is stronger than trace inclusion and also supports a notion of observational equivalence. Simulation techniques have been widely applied to the verification of hardware [17], software [13, 3] and protocol verification [6, 4] and also to check process equivalence in process algebras [9, 7].

In this paper, we extend simulation techniques for reusing existing components during embedded system synthesis. Though several techniques have been proposed for automated synthesis of these systems [10, 5, 2], component reuse during synthesis is still low key. A major reason for this is the lack of suitable algorithms to map a design *function* ($\mathcal{F}$) to a suitable *device* ($\mathcal{D}$) from a library of such components. Mitra et. al [11] proposed a restricted algorithm for such a mapping based on language containment. However, it is not based on a formal model and therefore lacks any means of proving the correctness of the mapping algorithm. Also, this algorithm completely ignores possible divergence of the device 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 new component identification algorithms 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.

Though the current simulation techniques have been successful in verifying that an implementation I satisfies a specification S, they are not directly applicable to our problem, since the implementation I is usually 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.

1.1 Overview

In this section, we shall give a brief overview of our approach and provide the motivation via an example. Firstly, we define labelled transition systems similar to [9] which will be used to represent $\mathcal{F}$ and $\mathcal{D}$ behaviourally. From now on we shall use symbols F and D to denote the LTSs of $\mathcal{F}$ and $\mathcal{D}$ respectively.

Definition 1:

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

1. S is a set of states,
2. $s_0 \in S$ is a unique start state,
3. 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,
4. $\stackrel{l}{\rightarrow} \subseteq S \times S$ for each $l \in L$.

We shall use symbols a, b, c .. to range over external labels, symbols $i, i_1, i_2, ..$ to range over internal labels and symbols $l, l_1, l2, ..$ to range over L. 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 $\mathcal{F}$).

Example 1: Consider F and D in Figure 1.1. It is obvious that the start states of F and D do not possess isomorphic behaviours. However, since transitions $0 \rightarrow 2$ and $2 \rightarrow 3$ are externally triggered in D, an external interface can generate the inputs c and d at appropriate points to guide the device to state 3. This is defined as forcing the device to state 3. As a result, the transition $3 \rightarrow 4$ in D is directly equivalent to the transition $0 \rightarrow 1$ in F. However, the transition $3 \rightarrow 5$ in D is not directly equivalent to the transition $0 \rightarrow 2$ in F since from state 5 of D, no transition labelled by c exists. However, by applying the notion of forcing again, the interface can again guide the device D to successor state 7 which has the appropriate transition. The avid reader can immediately note that this alone is not enough to ensure the implementation of F by D because of the presence of an internal transition from state 7 of D. Since internal transitions are autonomous, an external interface cannot have any control over them. Therefore, to ensure the implementation of F by D, we must additionally ensure that from the destination states of any such extra internal transitions, the interface can eventually place the device in a suitable successor state which is isomorphic to the function state in question and also has no further internal divergence. For example in the above figure, the interface can easily force the transition $9 \rightarrow 10$ in D to place the device in state 10 where the simulation is guaranteed.

The above example illustrates the following two important features of forced simulation, which to our knowledge, does not exist in any other simulation technique:

- The idea of forcing the device to reach a successor state that possesses behaviour isomorphic to the current function state. Forcing is a generic notion which can be used to initialise the device to a proper functionality and also to filter out any extra transitions.

- The idea of stabilizing the device via forcing in the event of internal transitions taken by the device, until it finally reaches a stable state without internal divergence, that is also isomorphic to the function state.

The novel notion *forced simulation* captures the correspondence between $\mathcal{F}$ and $\mathcal{D}$. We assume that the interface has complete knowledge of the state space of F and D and propose a *state-based interface* which moves in lock-step with the states of F and D to perform forcing as well as signal mappings. We have also developed a more observational version of forced simulation based on the idea of a buffered interface, which is not being

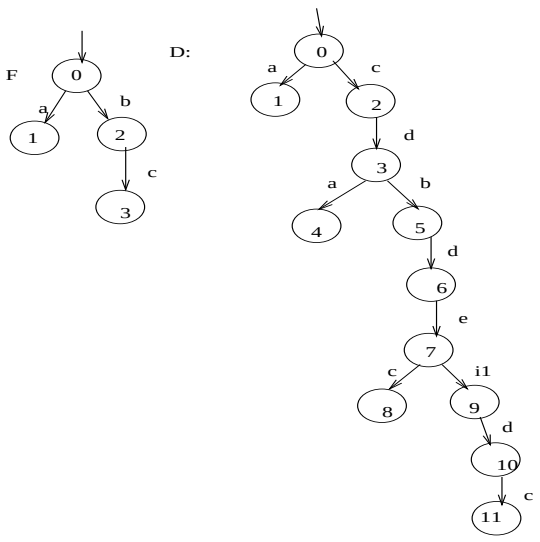

Figure 1.1: simple example

presented for lack of space [15]. However, all techniques presented in this paper have been developed for the buffered interface in parallel.

The paper is organized as follows: in Chapter 2, we shall formally define forced simulation and illustrate it via an example. We also propose rules for generating the interface automatically and prove the correctness of the generated interface. In Chapter 3 we present an algorithm based on forced simulation, which can be used for automatic component mapping. The fourth and final chapter will be devoted to some concluding remarks.

Chapter 2

Forced Simulation

So far the notion of forced simulation, henceforth called *fsimulation*, has been presented informally. In this chapter, fsimulation will be formally developed. Also, we present a technique to derive an interface, I, when F and D are fsimilar. Finally, we show that the derived interface is correct by showing that the synchronous parallel composition of I and D is weakly bisimilar [9] to 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 an 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 the start state of D fsimulates the start state of F (e.g,$s_{f0} \sqsubseteq_\Sigma s_{d0}$). 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 and further this successor state is devoid of internal divergence. 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. Also, if there are extra transitions out of the device state which are triggered by internal symbols, then for fsimulation to hold, the successor states of these transitions must also fsimulate the function state.

Fsimulation via Lockstep Interface: In this definition we assume the following:

1. LTSs may be non-deterministic.

2. Both $\mathcal{F}$ and $\mathcal{D}$ 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 dependent on both the current function as well as the device state and also has full knowledge of triggering transitions of the device. Such an interface will be called a *lock-step interface*.

We now formalize these concepts:

Definition 2:

$$
\begin{aligned}
s_f \sqsubseteq_\Sigma s_d &\stackrel{def}{=} \exists k \geq 0 : s_f \sqsubseteq^k_\Sigma s_d \\
s_f \sqsubseteq^0_\Sigma s_d &\stackrel{def}{=} subtree\_isomorphic(s_f, s_d) \wedge internally\_terminal(s_d, s_f) \\
s_f \sqsubseteq^{k+1}_\Sigma s_d &\stackrel{def}{=} (s_f \sqsubseteq^k_\Sigma s_d) \vee \\
&\quad ((subtree\_isomorphic(s_f, s_d) \wedge \\
&\quad (\forall i \in lL_d, \forall s'_d : s_d \rightarrow_i s'_d \wedge s_f \not\rightarrow_i s'_f : s_f \sqsubseteq^k_\Sigma s'_d)) \vee \\
&\quad ((\exists a \in eL_d, \forall s'_d : s_d \rightarrow_a s'_d : s_f \sqsubseteq^k_\Sigma s'_d) \wedge \\
&\quad (\forall i \in lL_d, \forall s'_d : s_d \rightarrow_i s'_d : s_f \sqsubseteq^k_\Sigma s'_d)) \\
subtree\_isomorphic(s_f, s_d) &\stackrel{def}{=} \forall l, s'_f : (s_f \rightarrow_l s'_f \Rightarrow \\
&\quad \exists s'_d : s_d \rightarrow_l s'_d \wedge s'_f \sqsubseteq_\Sigma s'_d) \\
&\quad \wedge (\forall s''_d : s_d \rightarrow_l s''_d : s'_f \sqsubseteq_\Sigma s''_d) \\
internally\_terminal(s_d, s_f) &\stackrel{def}{=} \text{only internal transitions allowed out of } s_d \\
&\quad \text{are common internal transitions of } s_f
\end{aligned}
$$

The definition asserts that fsimulation of s_f by a state s_d is possible if there exists a positive integer k such that s_d converges to a successor isomorphic state in at most k steps. Moreover, because the device is capable of internal steps, we must guarantee simulation despite the internal transitions. For example in Figure 1.1, D fsimulates F since, by the definition, the simulation of the state 0 of F happens in 2 steps by forcing to the state 3. At this state the simulation of state 1 of F happens in 0 steps by the state 4 of D. Also the simulation of the state 2 of F happens in a maximum of 4 steps either by forcing to state 7 of D when the internal transition from this state does not trigger, or by the internal transition $7 \rightarrow 9$ triggering followed by simulation within one more step.

The base case of the definition (when k=0) says that s_d can simulate s_f provided it is *subtree_isomorphic* (capable of reproducing at least all the behaviours of s_f) as well as *internally_terminal* with respect to s_f (s_d does not have extra internal transitions; all internal transitions of s_d are common

to s_f). The latter condition is included to prevent the device from diverging via an internal path not present in s_f, thus violating the simulation.

The result of applying the fsimulation definition to Figure 1.1 is shown in Figure 2.1 as a Match Graph (MG), which is formally defined later. In this figure, each node of this graph is a triplet, where the first element is the function state, the second element is the device state and the third is the k value. Also note that the transitions on MG are labelled by the following symbols:

- a: an external label to be generated by environment,
- *[a]*: when a is a forced label to be generated by the interface,
- i: when D makes progress via any internal transition
- λ: when the MG has a stuttering step in which neither F nor D make any progress and only the k value is being decreased.

The MG is an abstract description of the interface and will be used to generate the interface process automatically.

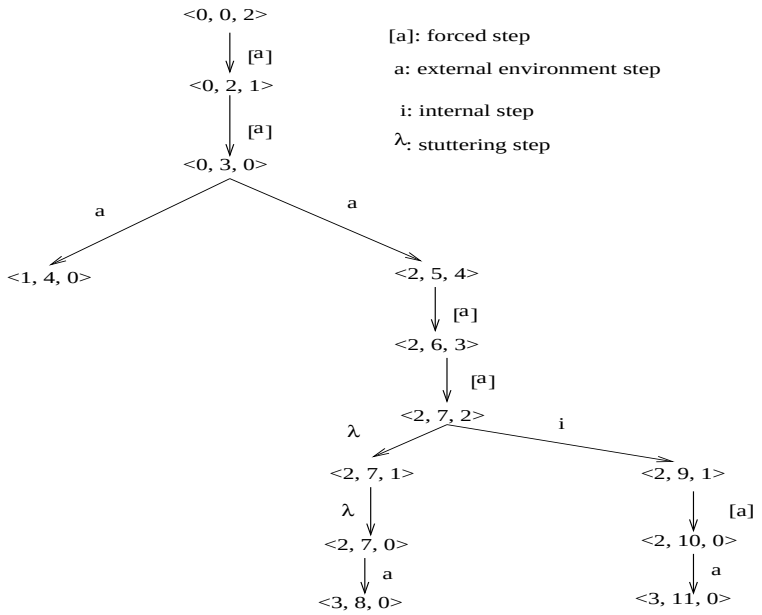

Figure 2.1: Match Graph for example 1

2.1 Interface Generation

In this section, we first formally define the match graph (MG) as an LTS. Following this, we discuss the method to generate the interface I, given the LTS of MG.

Definition 4 An MG is a labelled transition system (LTS) defined to be a tuple $(S_{mg}, L_{mg}, \{\stackrel{l}{\rightarrow}: l \in L_{mg}\})$, where:

1. S_{mg} is a set of states where each $s_{mg} \in S_{mg}$ is of the form $< s_f, s_d, k >$: $s_f \in S_f, s_d \in S_d$ and $k \geq 0$,
2. $< s_{f0}, s_{d0}, k >$ is the unique start state,
3. $L_{mg} = \{a, [a], i, \lambda\}$ is a set of transition labels.
4. $\stackrel{l}{\rightarrow} \subseteq S_{mg} \times S_{mg}$ for each $l \in L_{mg}$.

The basic idea behind interface generation is informally presented below. We assume the following:

- communication between I and D is synchronous.
- C_{start} denotes a unique start state of the interface I.
- We use the following types of transition labels:
    1. a: which is a standard external action. The interface waits for this symbol to be generated by the environment.
    2. $[a]$: the interface generates external action a to perform forcing on the device.
    3. $in[s_d]$: which returns true when the device enters state s_d by taking an internal transition.
    4. $dt_i(s_d)$: (does not trigger) which returns true when none of the internal transitions from the state s_d trigger within a time bound. We assume that there is at most one unit of time delay for an internal transition to trigger after a state is entered in the device. Hence after this time unit the interface can safely deduce whether an internal transition triggered. Such an assumption is consistent with our domain since in many devices the maximum time required before an internal event such as a *timeout*, for example, is known. Also, upper bounds on data computation times can be obtained from the device manuals.

- there is one state in the interface I corresponding to every MG state except for the stuttering states. If the MG state is denoted as $< s_f, s_d, k >$ then the corresponding I state is denoted as $C_{<s_f,s_d>}$.

Informally, from the current MG state $< s_f, s_d, k >$ we check to see if there are any transitions labelled by i. If there is no such transition, and the MG transition from this state is via an *[a]* label, then it can be deduced that this is a forcing step and hence the interface has a transition of the form $C_{<s_f,s_d>} \stackrel{[a]}{\rightarrow} C_{<s'_f,s'_d>}$ (this is defined as a *simple forcing step*). However, if there are transitions out of this MG state labelled by i and also has a transition labelled by $[a]$, then the interface has to first deduce whether any of the internal transitions trigger in the device before resorting to forcing. If none of them trigger, then the interface makes a transition to a state $C'_{<s_f,s_d>}$ from which it performs the appropriate forcing. We term such a forcing step, where the interface has to deduce whether the internal transition in the device triggers or not prior to forcing, as a *branching forcing step*. On the other hand, if the interface finds any of the internal transitions in D triggering, then it makes a transition to an appropriate state without performing forcing (defined as a *simple internal step*). The interface for F and D in Figure 1.1 is shown in Figure 2.2. The formal rules for interface generation appear in Appendix 1 in SOS style [14].

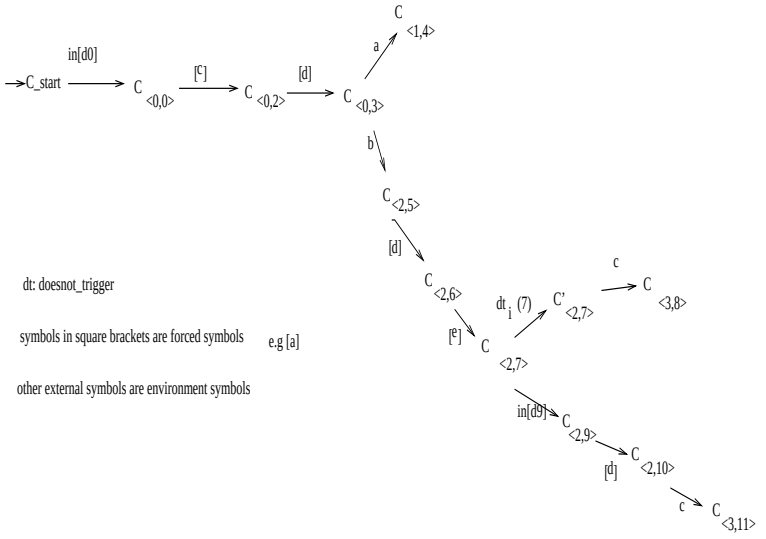

Figure 2.2: Interface generated for Example 1

2.2 Correctness of the generated interface, I

In this section we prove that the generated interface is correct by showing that $[I||D] \approx F$, where $\approx$ is Milner's weak bisimulation and $||$ is a synchronous parallel composition operator defined below in SOS notation [14].
Definition 5: $I||D$ is an LTS whose set of labels $L_{I||D} = L_d \cup \{\tau\}$ where τ denotes a unique unobservable label that occurs in $I||D$ whenever synchronous communication between I and D happens while forcing, and also when the device makes internal moves and the interface moves in lock-step with the device as a result of this move. As before we use subscript d to indicate a device state and use subscript c for an interface state.

1. Initialization: This rule is meant for composing the initial steps of the device and the interface.
$$\frac{in[s_{d0}], s_c \xrightarrow{in[s_{d0}]} s_{c1}}{(s_{d0}, s_c) \xrightarrow{\tau} (s_{d0}, s_{c1})}$$

2. Forced Move: Whenever the device is forced by the interface, this results in an unobservable τ action in the composition.
$$\frac{s_d \xrightarrow{a} s_{d1}, s_c \xrightarrow{[a]} s_{c1}}{(s_d, s_c) \xrightarrow{\tau} (s_{d1}, s_{c1})}$$

3. Internal Move: When the device makes an internal move by taking an extra internal transition not in F and the interface just moves in lock-step with the device, this also results in an unobservable τ move in the composite.
$$\frac{s_d \xrightarrow{i} s_{d1}, s_c \xrightarrow{in[s_{d1}]} s_{c1}}{(s_d, s_c) \xrightarrow{\tau} (s_{d1}, s_{c1})}$$

4. dt Move: Whenever the device is in a state with extra internal transitions out of it and none is taken within a time bound, and the interface progresses by deducing this fact, this also results in an unobservable τ move in the composite.
$$\frac{in[s_d], s_c \xrightarrow{dt_i(s_d)} s_{c1}}{(s_d, s_c) \xrightarrow{\tau} (s_{d1}, s_{c1})}$$

5. Environment Influenced External Move: Whenever, both the device and interface move due to the same environment generated external symbol, this results in an observable transition labelled by the same symbol in the composite.
$$\frac{s_d \xrightarrow{a} s_{d1}, s_c \xrightarrow{a} s_{c1}}{(s_d, s_c) \xrightarrow{a} (s_{d1}, s_{c1})}$$

Theorem 1: $F \approx [I||D]$

Proof:

Note that the interface I is constructed only when $F \sqsubseteq_\Sigma D$. Hence, by the definition of fsimulation, $\exists k : s_{f0} \sqsubseteq^k_\Sigma s_{d0}$. Also, by the definition of fsimulation there exists a successor state of s_{d0} say s_{dk} which can be reached in at most k steps such that $s_{f0} \sqsubseteq^0_\Sigma s_{dk}$. Also, these k steps can be any one of the following:

- a simple forcing step or,
- a branching forcing step or,
- an internal step.

Hence, by rules 2, 3, 4 (and of course rule 1 for initialisation) we will have the following set of up to $(2 * k)$ τ-transitions in the parallel composition: $(s_{d0}, C_{<0,0>}) \xrightarrow{\tau} .. \xrightarrow{\tau} (s_{dk}, C_{<0,k>})$. The number of τ transitions can be upto $2 * k$ since in the worst case we can have all steps as branching forcing steps.

Also, $s_{f0} \sqsubseteq^0_\Sigma s_{dk} \Rightarrow subtree\_isomorphic(s_{f0}, s_{dk})$

Hence, $\forall a, s_{f1} : (s_{f0} \rightarrow_a s_{f1} \Rightarrow \exists s_{dk1} : s_{dk} \rightarrow_a s_{dk1} \wedge s_{f1} \sqsubseteq_\Sigma s_{dk1})$

Hence, by application of rule 5 above, for every $s_{f0} \rightarrow_a s_{f1}$ in F we will have a transition of the form $(s_{dk}, C_{<0,k>}) \longrightarrow_a (s_{dk1}, C_{<0,k1>})$ in $I||D$. Also, these will be the only transitions out of the state $(s_{dk}, C_{<0,k>})$ due to the fact that s_{dk} is also an *internally_terminal* state (since $s_{f0} \sqsubseteq^0_\Sigma s_{dk}$) which eliminates any possibility of internal divergence from s_{dk}.

Hence, for every $s_{f0} \rightarrow_a s_{f1}$ in F there exists $(s_{dk1}, C_{<0,k1>})$ in $I||D$: $(s_{d0}, C_{<0,0>}) \longrightarrow_{\hat{a}} (s_{dk1}, C_{<0,k1>})$ and conversely ($\hat{a}$ is defined as in the weak bisimulation definition [9]). Moreover, since by the definition of *subtree_isomophic* we know that $s_{f1} \sqsubseteq_\Sigma s_{dk1}$, extending the same argument as above we can show that $s_{f1} \approx (s_{dk1}, C_{<0,k1>})$. Hence, $s_{f0} \approx (s_{d0}, C_{<0,0>})$.

Chapter 3

Component Identification Algorithm

In this chapter we present a component identification algorithm based on fsimulation via lockstep interface. A sketch of the algorithm is provided in Figure 3.1. A version of the full algorithm has been published in [15].

Given F and D as input, the *main*() program checks if an *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 $distinguish(s_f, s_d)$ that checks to see if this pair (s_f, s_d) cannot be distinguished. A pair cannot be distinguished, if $s_f \sqsubseteq_\Sigma s_d$ holds. That is, an fsimulation relations holds between these two states.

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 strategy [9, 7] and thus effectively handles cycles in LTSs. The algorithm also constructs a Match Graph (MG) as the matching of states is achieved by the *distinguish*() function.

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

```
Main Program
main(F, D)
//F and D are the LTSs of the function and device respectively
//the main program starts with an initial partition equal to the
//product of the state space of F and D and successively refines
//this partition until the GFP is reached.
R = state_f × 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 ∈ R then
  return TRUE;
else
  return FALSE
endif
```

Figure 3.1: component identification algorithm

repeat loop in $main()$ can iterate up to a maximum of $NS_f \times NS_d$. During each such iteration, the inner for loop can also iterate up to a maximum of $NS_f \times NS_d$. Hence, in the worst case the complexity of our simulation algorithm is of order $NS_f^2 \times NS_d^2 \times m$.

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 the product of the number of function and 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 will be employed when forcing is necessary and is not deployed every time two states need to be made equivalent. The latter is when no forcing is required to achieve the simulation.

Chapter 4

Concluding Remarks

In this paper, we have proposed a new simulation relation and developed algorithms based on it, 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 that are expected in the device but are missing in the specification. Forced simulation fails in the presence of internal divergence and succeeds only when the interface can force the device after an internal transition to a stable device state devoid of any internal divergence. This is a desirable feature.

Recently, in [12] Muller-Olm et. al have proposed a technique for generating *upgrade specifications* from reactive components and their interface languages. Using this approach they can test the downward compatibility of upgrade specifications with respect to existing reactive components. Though similar in spirit, the major distinguishing feature of our work is that the component adaptation step using forced simulation is actually a specialization of the component in comparison to upgrade specifications, which are generalizations of the component. In addition, the handling of internal events and the idea of stabilizing the device via forcing after an internal transition is absent in their work. Both these methods are complementary,

however, and there is scope for combining both.

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 and capable of performing many data and control transformations. We have recently proposed a language specifically tailored to component-based embedded systems [16]. Currently we are working on extending our algorithm to handle behavioural level descriptions in this language.

Bibliography

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

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

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

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

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

[6] 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.

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

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

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

[10] 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.

[11] 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.

[12] M. Muller-Olm, B. Steffen, and R. Cleaveland. On the evolution of reacive components - a process algebraic approach. In J. P. Finance, editor, *Fundamental approach to software engineering*, volume 1577 of *LNCS*, pages 161–175, 1999.

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

[14] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[15] P. S. Roop, A. Sowmya, and S. Ramesh. Forced simulation: a formal approach to component-based synthesis. Technical Report UNSW-CSE-TR-9901, School of CSE, University of New South Wales, March 1999.

[16] 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.

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

Appendix 1: Interface Generation

- INPUT: MG
- OUTPUT: I an LTS of the interface where $L_I = L_d \bigcup [L_d]$.
- Notation: C_{start} denotes an unique start state of the interface. All other states are drawn from the set $C = \{C_{<i,j>} \vee C'_{<i,j>} | i \geq 0, j \geq 0\}$. Also, predicates $in[s_d]$ and $dt_i(s_d)$ are used as labels in the transitions of I. The former label returns true when the device is in state s_d and the latter condition returns true if none of the internal transitions from the state s_d trigger.
- **AXIOMS:**

1. **Initialization:**
$$\frac{start(MG) == <s_{f0}, s_{d0}, k>}{C_{start} \xrightarrow{in[s_{d0}]} C_{<0,0>}}$$
2. **Simple Forcing Step:**
$$\frac{[<s_{fl}, s_{dm}, k> \xrightarrow{[a]} <s_{fl}, s_{de}, k-1>] \wedge [\nexists s_{di} : <s_{fl}, s_{dm}, k> \xrightarrow{i} <s_{fl}, s_{di}, k-1>]}{C_{<l,m>} \xrightarrow{[a]} C_{<l,e>}}$$
3. **Internal Device Step Not Taken:**
$$\frac{[<s_{fl}, s_{dm}, k> \xrightarrow{i} <s_{fl}, s_{di1}, k-1>] \wedge ... \wedge [<s_{fl}, s_{dm}, k> \xrightarrow{i} <s_{fl}, s_{din}, k-1>]}{C_{<l,m>} \xrightarrow{dt_i(s_{dm})} C'_{<l,m>}}$$
4. **Branching Forcing Step:**
$$\frac{[<s_{fl}, s_{dm}, k> \xrightarrow{i} <s_{fl}, s_{di1}, k-1>] \wedge ... \wedge [<s_{fl}, s_{dm}, k> \xrightarrow{i} <s_{fl}, s_{din}, k-1>] \wedge [<s_{fl}, s_{dm}, k> \xrightarrow{[a]} <s_{fl}, s_{de}, k-1>]}{C'_{<l,m>} \xrightarrow{[a]} C_{<l,e>}}$$
5. **Simple Internal Step:**
$$\frac{<s_{fl}, s_{dm}, k> \xrightarrow{i} <s_{fl}, s_{di}, k-1>}{C_{<l,m>} \xrightarrow{in[s_{di}]} C_{<l,i>}}$$
6. **External Step:**
$$\frac{<s_{fl}, s_{dm}, k> \xrightarrow{a} <s_{fp}, s_{dq}, k-1>}{C_{<l,m>} \xrightarrow{a} C_{<p,q>}}$$