A Formal Approach to Component Based Development of Embedded Systems

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
Mumbai - 400 076, India
ramesh@cse.iitb.ernet.in

UNSW-CSE-TR-0004-May 2000

Contents

1 Introduction **1**

2 Component Matching **4**

2.1 Existing Solutions 5

2.2 The Coffee Brewer Example 6

3 Forced Simulation: a formal approach to component matching **11**

3.1 The Interface Process 12

3.2 Interaction between the Interface and the Device 12

3.3 Component Matching 13

3.3.1 Forced Simulation 14

4 Component Matching Algorithm **19**

4.1 Illustration 20

4.2 Complexity 23

5 Implementation of the Results **25**

5.1 Coffee Brewer 25

5.2 Mapping of lathe controller port to Intel 8255 in mode 2 25

5.3 Mapping of interrupt on terminal count function to Intel 8254 in mode 0 . . . 26

5.4 Result Summary 28

6 Related Work **33**

7 Concluding Remarks **36**

A Theorem 1 **42**

B Theorem 2 **45**

List of Figures

2.1 Coffee Brewer Example 7
2.2 Specification of a Coffee Brewer 8
2.3 Interface for adapting the Coffee Brewer 8
2.4 Forcing for initializing D 9

3.1 conventional || operator 13
3.2 Example of Strongly Bisimilar Processes 14
3.3 Example of Weakly Bisimilar Processes 15
3.4 example to illustrate new simulation 16
3.5 f-simulation example 17

4.1 component matching algorithm: main program 21
4.2 Simple Illustration 22
4.3 Example of Reduce(B1, a, B2) 22
4.4 Simple Illustrations Interface 23

5.1 The port of a Lathe Controller 26
5.2 Intel 8255 and mode 0, mode 1 behaviour 27
5.3 Intel 8255 in mode2 and the Interface Process 28
5.4 Intel 8254 abstract behaviour and also mode 0 and mode 1 behaviours 29
5.5 Intel 8254 in mode 2 and mode 3 30
5.6 Intel 8254 mode 4 and mode 5 behaviours 31
5.7 Interrupt on Terminal Count Function 31
5.8 Interface Process for matching interrupt on TC to Intel 8254 in mode 0 . . . 32

Abstract

Component reuse techniques have been the recent focus of research as they are seen as the next generation techniques to handle increasing system complexities. However, there are several unresolved issues to be addressed and prominent among them is the issue of *component matching*. As the number of reusable components in a component database grows, the task of manually matching a component to the user requirements will be infeasible. Automating this matching can help in rapid system prototyping, improve quality and reduce cost. In addition, if the matching algorithm is sound, this approach can also reduce precious validation effort.

In this paper, we propose an algorithm for automatic matching of a design function to a device from a component database. The distinguishing feature of the algorithm is that when successful, it generates an *interface* which can automatically adapt the device to behave as the function. The algorithm is based on a new simulation relation called *forced simulation* which is shown to be a necessary and sufficient condition for component matching to be possible for a given pair of function and device. We demonstrate the application of the algorithm by reusing two system level Intel chips.

Chapter 1

Introduction

Component reuse methodologies have been the recent focus of industry and academia alike, mainly driven by the increasing complexities of modern systems. Other major factors influencing this revolution are immense competition from competing vendors and consequently less time to market, the need for more *open* (generic) solutions of the Internet era, as opposed to the more *closed* solutions of the pre-Internet era and most importantly the need for developing solutions that can be easily verified-often referred to as *design for verifiability* [10, 31]. The ostensible advantages of component reuse are improved productivity, better performance and more significantly better quality products.

In the domain of software engineering, Object Oriented (OO) techniques [5, 41] emerged in the early 80's to handle the increasing software complexity and were touted to be the solution to the "software crisis". The primary reuse concept in OO is that of reusing prevalidated software libraries by applying techniques such as *method forwarding* and *inheritance* to produce higher quality software. Also, middleware such as CORBA [36, 26] now appears to be an industry standard for providing a transparent object reuse framework in a distributed environment.

In parallel to these developments in the software domain, intellectual property (IP) reuse in System on a Chip (SoC) [11, 24] design is an emerging trend in the VLSI domain, which emphasises reusing IPs from various vendors in order to put entire systems on silicon. SoCs have been shown to provide better performance at reduced costs in comparison to the PCB based products of the 90's.

Even though component-based development is the need of the hour and has several advantages over existing methods, many unresolved issues remain to be addressed. Some of the more important ones are:

- developmental issues: how to identify and develop generic products that are easily

reusable

- database issues: how to store, index and retrieve the components. Also, can generic, domain specific and open databases be created
- matching issues: how to decide if a component matches some requirements
- compositional issues: how to compose a set of matched components

The focus of this paper is the third issue of component matching. As the number of reusable blocks in a component database increases, the task of manually matching the requirements to a component will be extremely hard. Also deciding on the best match for the given requirements will be an issue. Finally, manual matching may introduce some inadvertent errors that are difficult to detect and may prove to be costly. In this paper, we propose an algorithm for automatically matching embedded components and cores. Such an algorithm to be useful must satisfy the following criteria:

1. must be computationally tractable
2. must be sound

The proposed algorithm has a polynomial time complexity and is based on a formal notion called *forced simulation* which is proved to be sound. If the component library is developed in such a way that it stores only prevalidated components, then by employing a sound mapping algorithm we can ensure that the resulting design will be correct and save precious validation effort.

Mitra et. al [33] proposed an informal mapping algorithm to automatically map a design function to a system level component. However, since the algorithm was informal, it can not guarantee that the mapping will be devoid of errors. Also, the complexity of the algorithm was exponential in the worst case. Smith and de-Micheli [48] proposed methods for component matching and verification of circuits using polynomials. Also, low level components such as ALUs have been successfully reused [4, 22]. However, these techniques though suitable for low level components are unsuitable for matching system level components for embedded applications which requires matching reactive behaviours.

The main contributions of the paper are:

1. a new polynomial time component matching algorithm for system level components is proposed. To the best of our knowledge, this is a pioneering algorithm for such a task.

2. the algorithm is based on a new simulation relation called *forced simulation* proposed in this paper. Forced simulation is shown to be the necessary and sufficient condition for component matching.

3. the paper illustrates the algorithm by reusing peripheral components from Intel.

This paper is organized as follows: In chapter 2, we provide the problem definition and motivate an informal solution via an interesting embedded application (a coffee brewer). We also show why the existing methods are not directly applicable to the component matching problem. In chapter 3, we formalize the ideas of the previous chapter and propose a new simulation relation called forced simulation and show that such a simulation is a necessary as well as sufficient condition for our component matching algorithm. Chapter 4 presents the component matching algorithm and illustrates it via a simple example. Chapter 5 is the results chapter which illustrates the reuse of two Intel family chips using our matching algorithm. Chapter 6 briefly reviews some related literature. The final cahpter is devoted to some concluding remarks.

Chapter 2

Component Matching

In this chapter, at the outset we define the problem and discuss why existing solutions are inadequate. We then describe a solution to the problem via an example from the domain. In the rest of the paper, F is used to denote the specification of a design *function* (to be implemented) and D is used to denote the specification of a *device* from a library of system level components.

Given F and D, component matching tries to address the question "does D implement F ?"

Having posed the problem, let us consider the issues associated with component matching:

- How to identify the design functions (F) from a system specification ?
- How to represent the F and D ?
- What existing methods may be applicable to solve component matching ?

To address the first issue, we have to consider how system specifications are refined and subsequently partitioned. The refinement process often constitutes the extraction of a control and data flow graph (CDFG) over which functional decomposition is attempted until primitive functions are identified such that these cannot be refined any further. Often following this is a step called hardware software partitioning that determines which of these functions are to be allocated to hardware and which are to be allocated to software. All these tasks are part of a larger problem called codesign [25]. Many codesign environments such as the POLIS system from Berkeley [6] are publicly available and can be used for the above task.

The second issues of representation of F and D is extremely important to the component matching problem. The system components for embedded systems are, in general, *reactive systems* [16, 40] in the sense that they are in continuous interaction with their environment, repeatedly reacting to external signals with certain response signals or actions on the internal

registers, etc. A well-known formalism for modelling reactive systems is Milner's calculus of communicating systems [32]. In this framework, reactive systems are modelled as *processes* that evolve by communicating with other systems and hence the algebraic framework to specify processes and their interactions are also in general known as *process algebras*. Milner's CCS [32] and Hoare's Communicating Sequential Processes (CSP) [18] are examples of process algebras.

In process algebras, *labelled transition systems* (LTSs) are used as the underlying semantics of processes. Informally, LTSs are similar to state transition diagrams where the transition between the states are labelled by events that occur in the environment. We shall use LTSs as our models of F and D. LTSs are formally defined in chapter 3.

Many high-level specification languages [49, 35, 20, 8, 16, 29] can be used to model reactive systems such as embedded systems. In this paper, we selected LTSs rather than any of these languages to model F and D since LTSs have often been used as the underlying semantics of many of these languages and they often might compile to LTSs or automata as in the case of Esterel [8]. Thus, the choice of LTSs entails that we have not tied our component matching to any language and is a generic approach which can be easily extended to be useful to any particular specification formalism. As an exercise in this direction, we have recently extended the matching algorithm to Esterel [43]. However, this is not the focus of the current paper.

2.1 Existing Solutions

For component matching, some simple solutions such as the following may be considered:

- Is D behaviourally equivalent to F ?
- Is D a *refinement* of F or vice versa ?

Bisimulation [32] is a well known technique in process algebras to check for process equivalence. Bisimulation is a stricter notion than trace inclusion and automata equivalence [19] (it identifies fewer behaviours as equivalent in comparison to automata equivalence and trace inclusion) and very efficient algorithms [39] have been developed for checking bisimulation equivalence.

Several techniques [1, 27, 15] based on the notion of refinement have been proposed to test if a low level implementation $\mathcal{I}$ is a *simulation* of a high level specification $\mathcal{S}$. The main idea is that, $\mathcal{I}$ is a simulation of $\mathcal{S}$ if all traces of $\mathcal{I}$ are included in $\mathcal{S}$.

Though both bisimulation equivalence and refinement based techniques have been widely applied to the verification of both hardware [50] as well as software [38, 14], protocol verification [17, 15] and also to check process equivalence in process algebras [32, 23], they are not

directly applicable to our problem, since the implementation $\mathcal{I}$ is a refinement of the specification $\mathcal{S}$, and is not arrived at by *adapting* a general implementation to a given specification, which is the essence of reuse. We illustrate this by the following example.

2.2 The Coffee Brewer Example

We motivate the component matching problem and the need for adaptation by the following coffee brewer example developed for a testing application at B-Tree systems, Inc. [13]. We have slightly modified the example by ignoring the error cases. Here is a abstract description of the coffee brewer.

The coffee brewer (here after referred to as the brewer) allows the user to brew 5 or 10 cups of coffee without having to measure coffee grounds or water. The brewer draws its water supply directly from the building's plumbing, using the proper amounts based on the number of cups to be brewed. Similarly, coffee grounds are automatically measured from an internal supply bin. The brewer is controlled by the following switches:

1. ON/OFF_SW for power ON/OFF
2. STRENGTH_SW that determines the brew strength. When ON the brew is STRONG and when OFF the strength is MEDIUM
3. #_CUPS_SW that sets the number of cups to 10 when ON and to 5 when OFF.
4. BREW_CYCLE_SW that initiates the brew cycle based on the parameters set by the STRENGTH_SW and the #_CUPS_SW

The operator may select brew strength and number of cups in any order. Obviously, the other three switches have no function when the ON/OFF_SW is not ON. The behavioural description of this coffee brewer is provided in Figure 2.1 as an LTS. Note that based on the values of the two input switches STRENGTH_SW, #_CUPS_SW the brewer will be in one of the 4 brew states 2, 3, 4, 5, prior to start of the brew cycle.

Suppose that such a brewer has been implemented and is available (call it D). To implement a brewer that provides only five cups of coffee (F as in Figure 2.2), we would like to reuse the above implementation. In F, it is assumed that the brewer automatically enters one of two possible brewing states based on the value of the STRENGTH_SW, after system ON/OFF_SW is turned ON.

There are a number of constraints while reusing D. We cannot access the internals of D nor can D be modified. However, the sequence of events consumed by D can be observed to

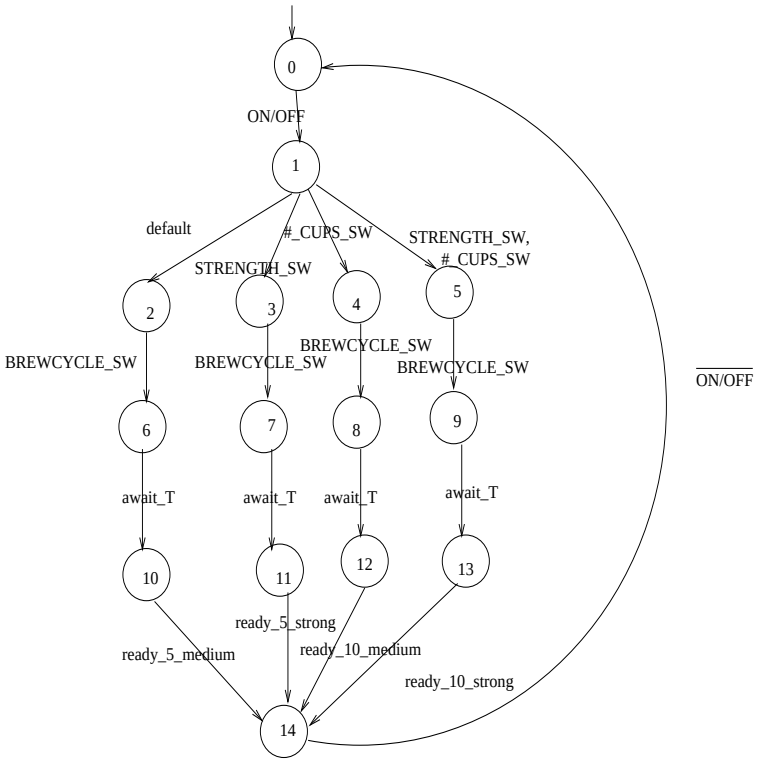


Figure 2.1: Coffee Brewer Example

determine the state of a deterministic D. Note that, in the above example, F is not in any way directly equivalent to D. Also, there is no refinement relation between the two.

The main idea behind matching a generic implementation to a new specification is to have an external process which moves in lock step with D (observing the sequence of inputs consumed by D, to determine D's state) to *adapt* D so that D can match F. Such a process is termed as an *interface* process. The task of the interface process is to suppress extraneous behaviours in D in certain states (since D is capable of more behaviours than F) and generate extra inputs to D (since D is more detailed than F) in some other states.

In our brewer example, for D to match F the interface must perform the following:

1. after system ON when D is in state 1, it must disable the #_CUPS_SW thus preventing D from entering states 3 and 4. Such actions of the interface are termed as *disabling*.

2. irrespective of whether the STRENGTH_SW is ON or OFF, when D is in either of the states 2, 3, it should set the BREWCYCLE_SW to ON. Such actions of the interface are termed as *forcing*. Since, forcing actions are autonomous interface actions, we enclose these actions within [ ] to differentiate them from environment generated actions.

The interface process for the above example is shown in Figure 2.3. The interface together

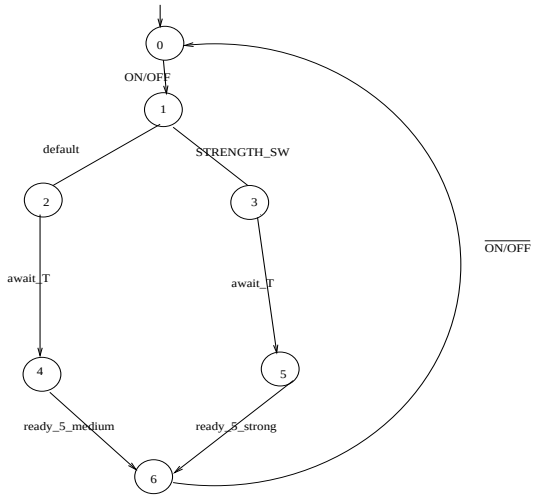


Figure 2.2: Specification of a Coffee Brewer

with D must give the same behaviour as F (i.e, be equivalent to F). Note that, the interface essentially moves in lock step with D either constraining its behaviour in some states (disabling) or generating extra inputs in D, which are not part of F (known as forcing) in some other states. When the interface moves in lock-step with D, it consumes the same inputs as received by D from the environment. However, when the interface forces an input to D, it must generate that input itself. Hence, this kind of actions of the interface are different from all other actions which are environment generated. In order to distinguish such actions, forcing actions are denoted as $[a]$ in contrast to environment actions which are denoted a.

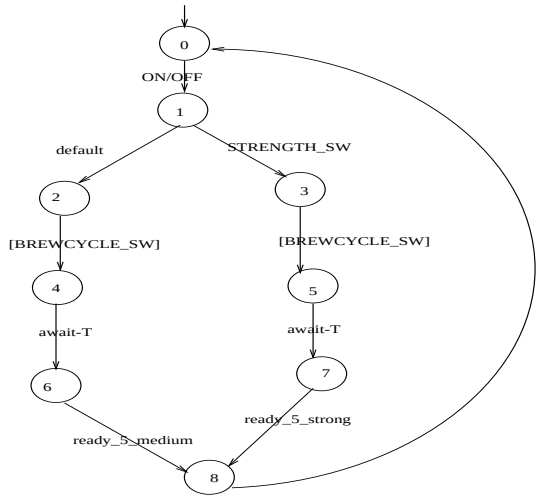


Figure 2.3: Interface for adapting the Coffee Brewer

The need for disabling some transitions in D is intuitive, since D in general is capable of

more behaviour. In contrast to disabling, forcing occurs due to any of the following:

1. initialization sequence for mode selection of D
2. D is more detailed compared to F

The first type of forcing is often required to select an appropriate mode of a multi-functional D to match F. Often, system level components are programmable and by appropriate mode selection can be programmed to behave in any one of a set of modes. For instance, a DMA controller can be programmed to be in either the *block DMA* mode or the *cycle stealing* mode or a general purpose port can be programmed to be a *simple I/O* port or a *handshaking* port. Programming a device consists of supplying a sequence of mode words and command words. After a device has been programmed, it behaves in the appropriate mode until reprogrammed again. Hence, when F is a abstract behaviour of only one of the modes of D, the interface has to generate the programming information to drive D to the appropriate mode. Figure 2.4 illustrates this idea.

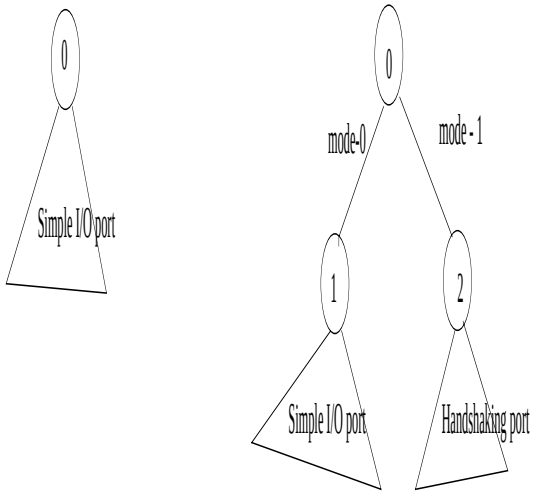


Figure 2.4: Forcing for initializing D

The second type of forcing occurs since D is an implementation and is normally more detailed compared to the specification F. Hence, D might have more control inputs compared to F which need to be generated by the interface. Our brewer example illustrates this type of forcing.

The brewer example illustrates that we cannot directly use existing simulation techniques since D is not a refinement of F and we need to adapt a general implementation to a given specification. As observed, such adaptation is to be performed by an external interface. However, this example also illustrates that the following issues need also to be addressed:

1. Given arbitrary pairs of F and D how do we decide if an interface exists

2. Also, given an interface for a known pair, how can we be sure that D implements all behaviours in F. In other words, how do we know if the interface is the correct interface.

As a solution to these questions, we propose a new simulation relation, called *forced simulation*. In the next chapter, we formalize the notions of F and D and then define this new simulation relation. We then show that forced simulation between F and D is a necessary as well as sufficient condition for the existence of a correct interface(i.e, the composition of the interface and D is indistinguishable from F).

Chapter 3

Forced Simulation: a formal approach to component matching

In this chapter, we shall formalize the various notions like devices, functions, interfaces and interface generation. We first formalize how these are represented. Then, we define the interface process and also show how to compose the interface with the device. Finally, we define a new simulation relation called forced simulation and show that it is a necessary and sufficient condition for component matching.

Definition 1: A process is described by a labelled transition system (LTS) which is a tuple of the form $< S, s_0, \Sigma, \rightarrow >$, where:

1. S is a finite set of states,
2. $s_0 \in S$ is a unique start state,
3. Σ is a finite set of events or signal.
4. $\rightarrow \subseteq S \times \Sigma \times S$ denotes the transition relation.

The functions, devices and interfaces that we mentioned in the previous chapter will all be modelled as processes.

Given a pair of states, $s, s' \in S$ and an event $a \in \Sigma$, we shall use the standard infix notation $s \stackrel{a}{\rightarrow} s'$ rather than $(s, a, s') \in \stackrel{a}{\rightarrow}$. Given a state s of any process we use $Lab(s)$ to denote the set of events a such that $s \stackrel{a}{\rightarrow} s'$ for some s'.

In this paper, we assume that all processes are deterministic.

Definition 2: A process is said to be deterministic if whenever $s \stackrel{a}{\rightarrow} s'$ and $s \stackrel{a}{\rightarrow} s''$ implies $s' = s''$.

Let $F =< S_F, s_{f0}, \Sigma, \rightarrow_F >$ and $D =< S_D, s_{d0}, \Sigma, \rightarrow_D >$ stand for the function and device processes respectively.

3.1 The Interface Process

Recall the informal discussion in the last chapter on the role of the interface process. The interface process adapts, if possible, a given device to match the behaviour of the function to be implemented. Two functions were performed by the interface: disabling and forcing. In general, the adaptation is dependent on the state of the device as well as that of the function. This is complicated by the fact that the state of the device is not accessible to the interface process. However, under the assumption that the states are dependent purely upon the external signals 'consumed', one can design an interface. But such an interface has to run in lock step with the device and be able to observe and enable or disable transitions in the device; further it can force certain transitions in the device.

Modelling such an interface requires a special kind of action to model forcing. So first we define an interface process.

Definition 3: An interface process is a process over $\Sigma \cup \{[a] | a \in \Sigma\}$.

The new set of signals of the form $[a]$ are the special signals that force the transitions labelled a in the device.

According to the above definition, it is possible to have interface processes in which out of a given interface state we can have both forcing events as well as external events. We define well formed interfaces which will never have such behaviours.

Definition 4:

An interface process I is said to be well formed if the following hold:

- $s_i \stackrel{[a]}{\rightarrow} s_i' \in \rightarrow_I \Rightarrow Lab(s_i) = \{[a]\}$.

3.2 Interaction between the Interface and the Device

Given D, I we now need to define the interaction between I and D. The following example in Figure 3.1 illustrates the conventional parallel ($||$) composition of two processes [32, 18]. Note that in this type of composition, processes are allowed to evolve autonomously. Thus, if we choose the conventional parallel operator ($||$) to compose I and D, then D can choose to evolve autonomously ignoring how I is evolving.

Since, out goal is that I should adapt D to behave as F, I must have full control over D. This is very important since, I has to *force* D in some steps and also *disable* D in some other steps. While forcing, D has to consume the input generated by I and while disabling, D will progress with respect to an environment input only if I is also progressing. Keeping these requirements in mind, we define a new operator ($//$) to compose I and D as defined by the following two rules of Definition 5.

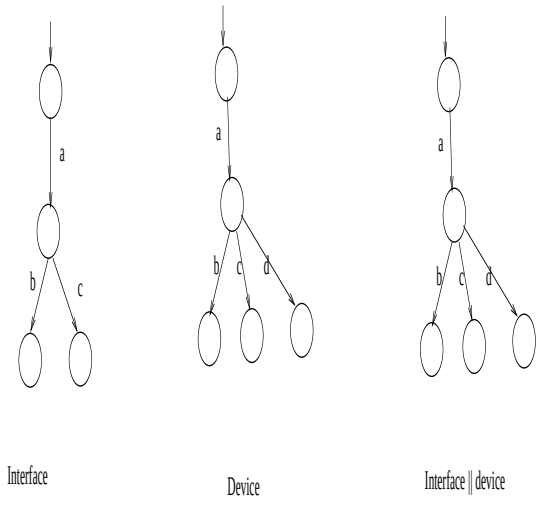


Figure 3.1: conventional || operator

Let $D =< S_D, s_{d0}, \Sigma, \rightarrow_D >$ be a process and I be the interface process $< S_I, s_{i0}, \Sigma_I, \rightarrow_I >$ where $\Sigma_I = \{[a] | a \in \Sigma\} \bigcup \Sigma$.

Now we can define the interaction between D, I by defining a new operator $//$.

Definition 5: Given I, D as above, $I//D$ is defined to be a process described by an LTS, $< S_{(I//D)}, (s_{i0}, s_{d0}), \Sigma_{(I//D)}, \rightarrow_{(I//D)} >$ where $\Sigma_{(I//D)} = \Sigma \bigcup \{\tau\}$ and the transition relation $\rightarrow_{(I//D)}$ is defined by the following rules:

1. Forced Move: $(I//D)$ makes an unobservable τ move, when I 'forces' a transition in D.
   $$\frac{s_d \xrightarrow{a} s_{d1}, s_i \xrightarrow{[a]} s_{i1}}{(s_d, s_i) \xrightarrow{\tau} (s_{d1}, s_{i1})}$$
2. External Move: $(I//D)$ makes an observable move with both the D and I simultaneously responding to the same external signal.
   $$\frac{s_d \xrightarrow{a} s_{d1}, s_i \xrightarrow{a} s_{i1}}{(s_d, s_i) \xrightarrow{a} (s_{d1}, s_{i1})}$$

We have so far formalized the notions of F, D, I and also defined the interaction between I and D. Now, we return to the main questions posed in the previous chapter, given arbitrary pairs of F and D, how do we decide if an interface exists. Also, if one exists, how do we know that this is the desired interface.

3.3 Component Matching

Now we formalize the component matching problem by the following definition:

Definition 6:

A device D can implement a function F (D matches F) if there exists an interface I such that $I//D \approx F$ where $\approx$ is Milner's weak bisimulation [32].

Two notions of bisimulation were proposed by Milner [32] 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 3.2 depicts an example of two strongly bisimilar processes.

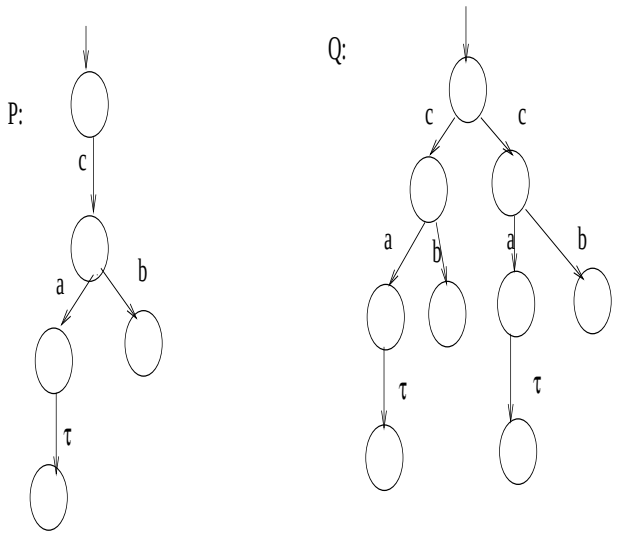


Figure 3.2: Example of Strongly Bisimilar Processes

In *weak bisimulation*, the restriction that each τ action of the processes has 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 3.3 is an example of processes that are not strongly bisimilar but are weakly bisimilar.

For component matching application, weak bisimulation equivalence is the appropriate equivalence since $I//D$ should be observationally inseparable from F, for D to match F.

3.3.1 Forced Simulation

The main task in component matching is to determine if an interface exists between arbitrary pairs of F and D. We propose a new simulation relation called *forced simulation* and then show that it is a necessary as well as a sufficient condition for interface existence. Before presenting the definition, we present the intuition by the following example in Figure 3.4.

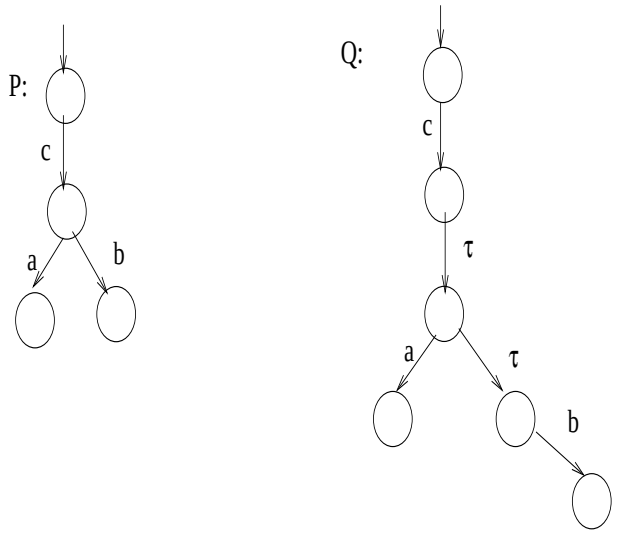


Figure 3.3: Example of Weakly Bisimilar Processes

For this device to implement the function, the start states of both must be matched. In this example, the device start state *0* had different behaviour compared to the function state *0*. However, the device start state has a successor state *3* which has matching behaviour as function states *0*. So, if an external interface forces the device state *0* to state *3*, then the device can match the behaviour of the function start state *0*. Hence, in this case, the two start states $(0, 0)$ match via a *forcing sequence* $b.c$, which is the sequence of inputs that must be applied from device state *0* to make D reach the state *3* which directly matches function state *0*.

Also, since $(0, 0)$ match via forcing sequence $b.c$, function state *0* must match device state *1* (state reached by consuming the b input from *0*) via forcing sequence c and similarly $(0, 3)$ must match directly (e.g, by empty forcing sequence). The states of D which match a state of F via a non-empty forcing sequence are termed as *forcing states*.

When a direct match happens, as in the case of $(0, 3)$ every transition out of function state *0* must be matched by a corresponding transition from device state *3*. Also, the resultant states of these transitions must match. For example, the transition $0 \xrightarrow{a} 1$ is matched by $3 \xrightarrow{a} 5$. Hence, $(1, 5)$ must match. This match can be a direct match or a match via a forcing sequence. In this example, it is a direct match. Also, note the inherent asymmetry in this relation during the direct match step. This is due to the fact that D is a generic component and is capable of more behaviour. However, in order to match F it must be capable of reproducing all behaviours of F. States of D which match a state of F by empty forcing sequence (a direct match) are termed as *matching states*. This example also illustrates that there may be alternatives for forcing and matching states and the example has only depicted

one such matching. For instance, the function state *0* also matches device state *0* via forcing sequence c and function state *1* matches device state *4* via forcing sequence b.

Generalizing the ideas of the above example, for D to match F, the start states must match via some forcing sequence σ (including empty σ). Also, if any two states (s_f, s_d) match via forcing sequence $a.\sigma$, then the a-derivative of s_d (successor reached via a) should match s_f via forcing sequence σ. Finally, if two states (s_f, s_d) directly match (via empty σ) then, not only should they have matching transitions; in addition their respective successors must also match through some arbitrary forcing sequence σ.

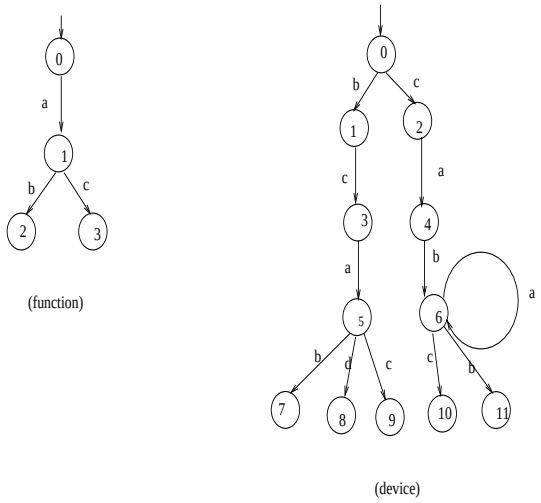


Figure 3.4: example to illustrate new simulation

Definition 7:

Given processes F and D, a relation $R \subseteq S_F \times S_D \times \Sigma^*$ is called a forced simulation relation (in short, an f−simulation relation) provided the following hold ($s_f R^\sigma s_d$ is used as a shorthand for $(s_f, s_d, \sigma) \in R$.):

1. $s_{f0} R^\sigma s_{d0}$ for some σ.
2. $s_f R^{a.\sigma} s_d \Rightarrow (\exists s'_d : s_d \xrightarrow{a} s'_d \wedge s_f R^\sigma s'_d)$.
3. $s_f R^\epsilon s_d \Rightarrow \forall a, \forall s'_f, \exists s'_d, \exists \sigma : s_f \xrightarrow{a} s'_f \Rightarrow (s_d \xrightarrow{a} s'_d \wedge s'_f R^\sigma s'_d)$

The first condition corresponds to matching of the start states of F and D. The second condition captures the matching of a function state to a forcing state in D. Finally, the third condition corresponds to the direct matching case.

Definition 8:

$F \sqsubseteq_{fsim} D$ provided there exists an f−simulation relation between them.

Now we illustrate f−simulation by the following example.

Example:

Consider processes F and D as shown in Figure 3.5. $F \sqsubseteq_{fsim} D$ since there exists

$R = \{(0,0,\epsilon), (1,1,c), (1,2,\epsilon), (2,3,\epsilon)\}$

which can be easily shown to be an f−simulation relation.

Note that $(0,0,\epsilon) \in R$ since according to rule 3 of Definition 7, $0 \xrightarrow{a} 1$ in F and $0 \xrightarrow{a} 1$ in D and $(1,1,c) \in R$. Similarly note that $(1,1,c) \in R$ (by rule 2) since $\exists 2 : 1 \xrightarrow{c} 2$ and $(1,2,\epsilon) \in R$ by applying rule 3.

However, R need not be unique as shown by another relation R' which is also an f−simulation relation on the same pair.

$R' = \{(0,0,a.c.b), (0,1,c.b), (0,2,b), (0,3,\epsilon), (1,0,a.c), (1,1,c), (1,2,\epsilon), (2,3,\epsilon)\}$.

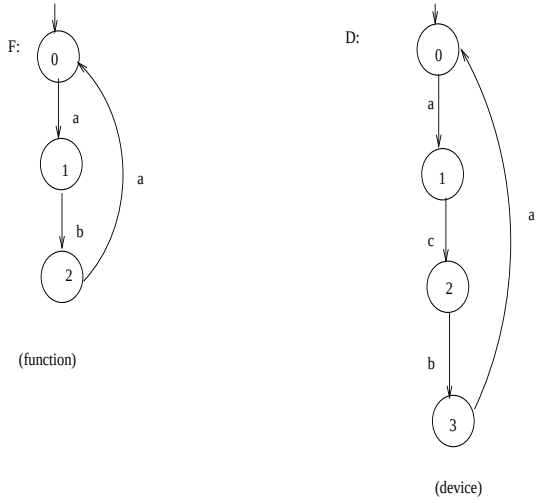


Figure 3.5: f-simulation example

Now we state the theorem which gives the necessary condition for the existence of I.

Theorem 1:

Given $F \sqsubseteq_{fsim} D$ there exists I such that $F \approx (I//D)$. $\approx$ refers to Milner's weak bisimulation [32].

Here we present an informal sketch of the proof. The detailed proof is provided in Appendix A.

The proof of the above theorem is constructive. Given $F \sqsubseteq_{fsim} D$, there exists an f−simulation relation R. For simplicity, let R be a minimal f−simulation relation such that for any pair $s_f \in S_F, s_d \in S_D$ there is at most one $\sigma \in \Sigma^*$ with $(s_f, s_d, \sigma) \in R$. The proof can be carried out, however, even for a non minimal R. Given R the interface process I is constructed. Then $I//D$ is also constructed using the rules of Definition 5. Then, we define a relation between the states of F and that of $I//D$ such that the function state component matches. To prove the theorem we then show that this relation is a weak bisimulation rela-

tion. The essential idea is that when $I//D$ is constructed all the forcing steps in I will be τ steps in $I//D$ and all the matching steps of I will exactly correspond to F. Hence, $I//D$ will be observationally inseparable from F.

Theorem 2:

If there exists a well formed and deterministic interface I such that $F \approx I//D$ then $F \sqsubseteq_{fsim} D$.

This proof is also constructive. Given that $F \approx I//D$ there is a weak bisimulation relation $\mathcal{S}$ over F and $I//D$. Given $\mathcal{S}$ we construct a new relation $R \subseteq S_F \times S_D \times \Sigma^*$ and then show that R is a f−simulation relation. Detailed proof is provided in the Appendix B.

In conclusion, this chapter provides the basis of component matching by formalizing devices (components), interfaces as well as functions as processes and then defining a new simulation relation called forced simulation between arbitrary function and device processes. Finally two important theorems establish forced simulation as the necessary and sufficient condition for component matching. Having established the basis of component matching, the next chapter proposes an algorithm for component matching.

Chapter 4

Component Matching Algorithm

In the previous chapter, we formalized the notion of interface generation via a new simulation relation called forced simulation. We have also shown that forced simulation is a necessary and sufficient condition for component matching. Now, we propose an algorithm to compute the forced simulation relation given F and D. The algorithm when successful in finding a f−simulation relation, will also generate the interface I.

Paige and Tarjan [39] proposed a very efficient algorithm for computing bisimulation relations. Their algorithm is based on the relational coarsest partition problem [2]. The essential idea is to start with an initial partition containing all the states of the processes and then successively refining it until a fixed point is reached. On termination, the algorithm identifies all the equivalence classes. The refinement process is carried out by choosing a splitter partition in each iteration. All other partitions are split based on the splitter partition. We have adapted this algorithm for computing forced simulation relations. Forced simulation computation differs from bisimulation computation in the following ways:

1. bisimulation computation computes the equivalence classes whereas in forced simulation, we have to identify a set of device states that are f−similar to a given function state. Thus, we need to compute the forcing sequences (σ's).

2. bisimulation is a symmetric relationship in contrast to forced simulation which is asymmetric (being a simulation from F to D). Thus, it is possible to have two different s_f's to be related to the same (s_d, ϵ). As a result, we cannot have partitions but rather use blocks.

Hence, the forced simulation algorithm has a pre-computation step, in which it computes all reachable states from every device state and also the path required to reach that state (a set called $RS(D)$). For example, if s_d has reachable states s_{d1} and s_{d2} via paths σ_1 and σ_2,

then the reachability set for s_d would be $\{(s_d, \epsilon), (s_d, \sigma_1), (s_d, \sigma_2)\}$. Note that, the algorithm always finds the minimum length σ to reach a given successor state.

Once this set is computed for all states in D, we use the elements of this set rather than the states in D to perform block refinement. We create the initial set of blocks by pairing every s_f in F with $RS(D)$ (since, initially we assume that each function state can be f-simulated by all elements in this set). After this, these initial set of blocks are refined until a fixed point is reached. The refinement process is carried out by choosing one of the blocks as a refining block and then refining all other blocks based on this. The pseudo code for the algorithm is provided in Figure 4.1. We now explain the algorithm by the following example.

4.1 Illustration

Consider the example of F and D as shown in Figure 4.2. The first step of the algorithm is the call to Compute_Reachable() function which computes all the reachable states from every state of D .

In this example, the states reachable from *0* are *0,1* which written in the (s_d, σ) form will be $\{(0, \epsilon), (0, c)\}$. Similarly, the states reachable from *1* are *1,0*, which is essentially the set $\{(1, \epsilon), (1, a)\}$. Thus, the set $RS(D) = \{(0, \epsilon), (0, c), (1, \epsilon), (1, a)\}$.

The second step of the algorithm is to compute the initial set of blocks ρ_I. This is done by having one block corresponding to every function state. Since, F in this case has only two states, there will be only two blocks in ρ_I. Each block contains one function state and all the elements of $RS(D)$. Hence, the initial set of blocks for this example will be:

$\rho_I = \{ B_1 = \{0, (0, \epsilon), (0, c), (1, \epsilon), (1, a)\},\ B_2 = \{1, (0, \epsilon), (0, c), (1, \epsilon), (1, a)\} \}$

Once we have the initial set of blocks, we initialise the set of waiting blocks, *waiting*, and the set of output blocks, ρ, to ρ_I: $waiting = rho_I$, $rho = rho_I$. Then, we choose one block from waiting as the refining block and refining all the matching blocks in ρ. We illustrate the refinement process again by the example.

Let us assume that initially the refining block chosen from *waiting* is

$B_2 = \{1, (0, \epsilon), (0, c), (1, \epsilon), (1, a)\}$.

In this example, $\Sigma = \{a, b\}$. Suppose we initially choose *'a'*. A matching block is a block of ρ which has a transition via *'a'* from it's s_f to the s_f of the refining block. In this example, the only block that satisfies this is $B_1 = \{0, (0, \epsilon), (0, c), (1, \epsilon), (1, a)\}$ since there is a transition in F from *0* to *1* via a. Now, Reduce(B_1, a, B_2) will yield $\{0, (0, c), (1, \epsilon)\}$ since both $(0, c)$ and $(1, \epsilon)$ have a transition via a to *0* and $(0, \epsilon) \in B_2$. This reduce step is depicted in Figure 4.3. Note that the (s_d, σ)'s which are crossed out are the ones being removed because of the refinement since they don't have any transition via *'a'* to any element of the refining block

Matching Algorithm

$match(F, D)$

//F and D are the LTSs of the function and device respectively

1. Compute_Reachable(D)

//this step computes all the reachable states of every device state and

//returns a set called $RS(D)$ which is a set of (s_d, σ) pairs.

$RS(D)$=$\{(s_d, \sigma) | s_d \in S_D \wedge (\exists \sigma \in \Sigma^* : after(s_d, \sigma) = s'_d \wedge s'_d \in S_D) \wedge (\exists \sigma' \in \Sigma^* : after(s_d, \sigma') = s'_d \Rightarrow (length(\sigma) < length(\sigma')))\}$

2. $\rho_I = \{B_{s_f} | s_f \in S_F\}$

$B_{s_f} = \{\{s_f\} \bigcup RS(D)\}$

3. $\rho = \rho_I$

4. $waiting = \rho_I$

5. **repeat**

   choose and remove any $B' \in waiting$

   for each $a \in \Sigma$ **do**

      $matchB = \{B \in \rho | s_f \in B \wedge s'_f \in B' \wedge s_f \xrightarrow{a} s'_f\}$;

      for each $B \in matchB$ **do**

         $reduceB = Reduce(B, a, B')$

         $\rho = \rho - B \bigcup reduceB$

         $waiting = waiting - B \bigcup reduceB$

      endfor

   endfor

until $waiting = \phi$

if any $(s_f, [])\in rho$ where $s_f \in S_F$ **then**

   return FALSE;

else

   Generate_Interface(ρ)

endif

//the Reduce() function

Reduce(B, a, B')

$reduceB = \{\{s_f \in B\} \bigcup \{(s_d, \sigma) \in B | \exists (s'_d, \sigma') \in B' \wedge (s_d, \sigma) \xrightarrow{a} s'_d\}\}$

return $reduceB$

Figure 4.1: component matching algorithm: main program

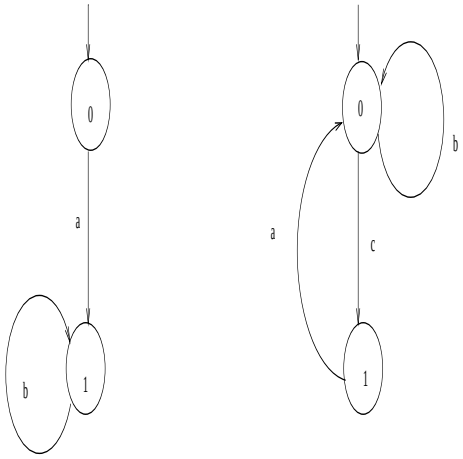


Figure 4.2: Simple Illustration

B_2.

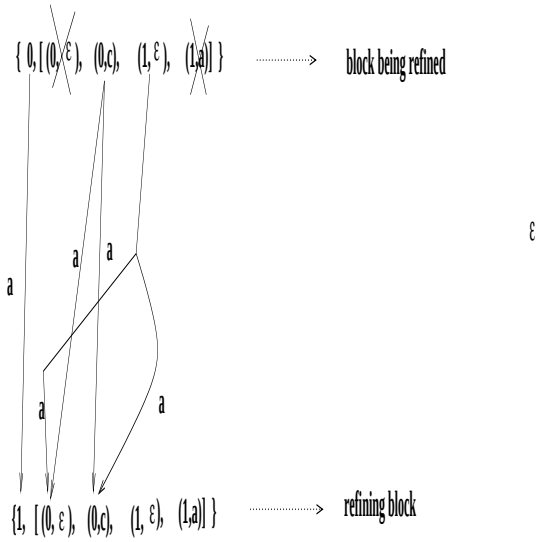


Figure 4.3: Example of Reduce(B1, a, B2)

This process continues until *waiting* has no more blocks and the algorithm finally yields $\rho = \{\,\{0, (0, c), (1, \epsilon)\},\ \{1, (0, \epsilon), (1, a)\}\,\}$

The interface generation process is straight forward, after the set ρ has been computed. The basic idea is to have an algorithm that given a function state s_f, and a device state s_d to be matched, chooses an $(s_d, a.\sigma)$ from the block corresponding to that function state such that the $a.\sigma$ chosen is minimal. It also installs $(s_f, s_d, a.\sigma)$ as the current interface state. Then, it calls a method called *force()* which applies the forcing sequence $a.\sigma$ by installing (s_f, s'_d, σ) as the successor of $(s_f, s_d, a.\sigma)$ where $s_d \xrightarrow{a} s'_d$. This process is repeated until $\sigma = \epsilon$.

At this point a function called *match()* looks at the direct successors of current s_f and s_d and installs them as appropriate successors by repeating the same states as above. The algorithm starts with (s_{f0}, s_{d0}) and terminates when all states of F have been matched.

Note that, the interface generation algorithm is greedy in the sense that it selects a given (s_d, σ) such that σ is minimal. However, such an approach does not necessarily generate an optimal interface (in the sense of the number of forcing steps). Finding an optimal interface is a typical optimization problem that needs to be separately tackled.

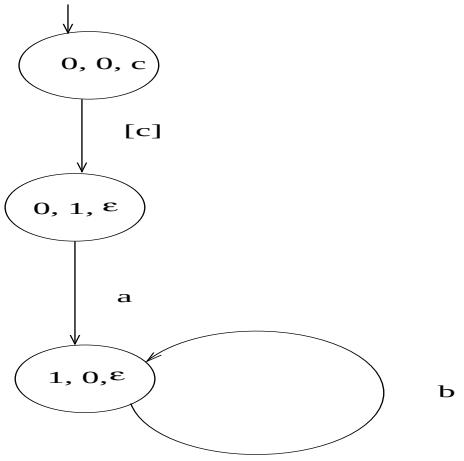


Figure 4.4: Simple Illustrations Interface

In the above example, we start by trying to match $(0, 0)$. We find that in the block corresponding to function states 0, the only state corresponding to device states 0 is $(0, c)$. So, the start state of I is $(0, 0, c)$. Then the function *force()* will install $(0, 1, \epsilon)$ as the successor of the start state via label *[c]*. Now the *match()* function will try to match the only successor of function state *0*, which is *1* to the successor of device state *1* via the same label a, which is *0*. The algorithm then finds from ρ that the function state *1* and the device state $(0, \epsilon)$ are in the same block. Hence, it installs $(1, 0, \epsilon)$ as the successor via label a. Then *match()* is again invoked to match the function state 1 to device state 0 resulting in a self loop being created via label b. The interface is shown in Figure 4.4.

4.2 Complexity

Let NS_f and NS_d denote the number of states of F and D respectively. Let e denote the maximum of NS_d and the number of edges in D. Also, let $m = \|\Sigma\|$.

The complexity of Compute_Reachable() (step 1) is of order $NS_d \times e$. The complexity of forming ρ_I (step 2) is of the order NS_f. Now, let us consider the main complexity which comes from the repeat loop.

1. Total number of blocks in $waiting = NS_f$.

2. size of each block is NS_d^2

Hence, the repeat loop will execute for a maximum of NS_f times and each time the repeat loop performs the following:

1. choose one block from waiting as the refining block

2. refine all matching blocks in ρ by calling $Reduce()$ for each label in σ.

Hence, the worst case complexity of the above two steps will be $NS_f \times NS_d^2 \times m$ since the size of a block is in the worst case NS_d^2 and in the worst case all the NS_f blocks need to be reduced for all the m labels in σ.

Hence, the complexity of the repeat is $NS_f^2 \times NS_d^2 \times m$. Taking into account all the additive steps, the total complexity of the algorithm is: $NS_d \times e + NS_f + NS_f^2 \times NS_d^2 \times m$ which is $O(NS_f^2 \times NS_d^2 \times m)$.

Chapter 5

Implementation of the Results

We have implemented a tool-kit for f-simulation in Java. Using the graphical user interface, specifications for F and D may be created and stored as files. Also, the f−simulation algorithm has been implemented such that it reads the LTSs of F and D as files and then returns the Interface if matching is successful. If matching fails, then the algorithm returns the failure state pairs that the algorithm failed to match. On successful matching however, the Interface is returned both textually as well as graphically in a GUI window. We have tested the algorithm on several examples form the domain, a few of which are presented below.

5.1 Coffee Brewer

Consider the Coffee Brewer Example of chapter 2. F and D for this example are described in Figure 2.2 and Figure 2.1 respectively. Mapping of F to D using the algorithm resulted in the interface process as shown in Figure 2.3.

5.2 Mapping of lathe controller port to Intel 8255 in mode 2

In this section we illustrate the component matching algorithm by matching the port of a lathe controller to Intel 8255 in mode 2.

Figure 5.1 provides an abstract description of a typical port in a lathe controller [45]. The function of this port is to read instructions written in a tape reader and then transfer these to the CPU in a handshaking fashion. The CPU interprets these instructions and then writes appropriate lathe instructions to the port, which are read by the lathe to perform the appropriate lathe action. This figure gives the abstract description of each transition of the handshaking sequence as comments.

The selected device for mapping F is Intel 82C55. The abstract description of this device

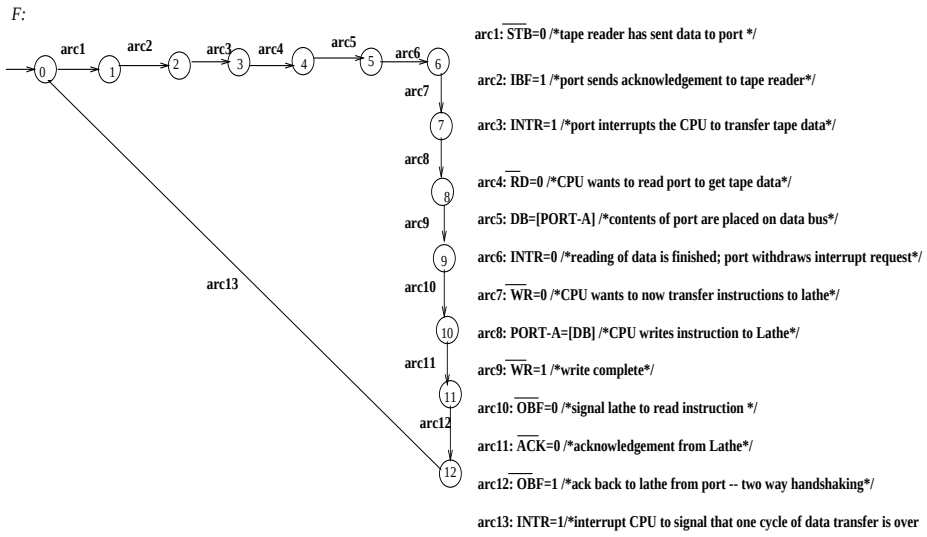


Figure 5.1: The port of a Lathe Controller

is provided in Figure 5.2(a). This figure also contains the detailed behaviour of mode 0. The detailed behaviours of mode 1 and mode 2 are provided in Figure 5.2(b) and Figure 5.3(a). Mode 0 is a simple input-output mode in contrast to modes 1 and 2 which can perform handshaking with peripherals and the CPU, the handshaking being a two way. The primary difference between mode 1 and mode 2 is that the former can be used in one direction of data transfer where as the latter can be used in bidirectional data transfer.

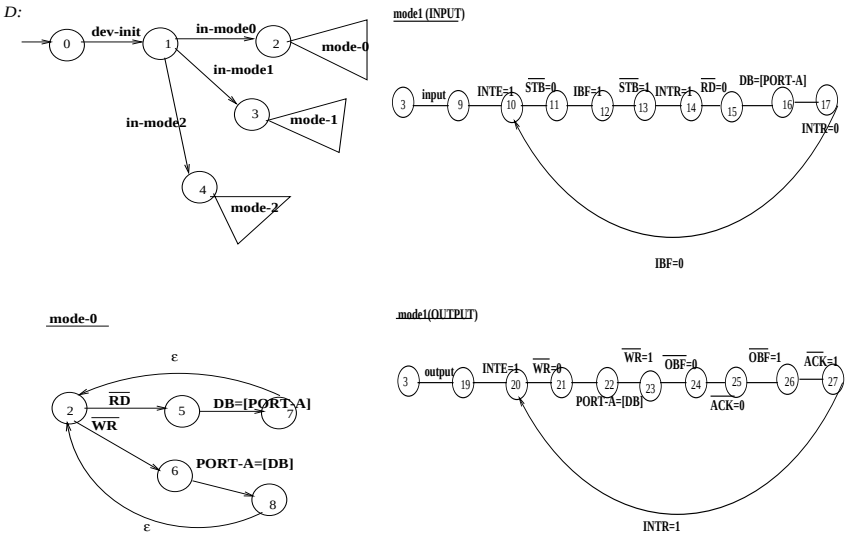
The result of mapping the lathe port (F) to the above device is provided as an Interface in Figure 5.3(b). The algorithm performs device initialization by supplying the *dev-init* command and also selects the appropriate mode (mode-2) that matches F. It also provides intermediate extra control signals required by the device such as the enabling of interrupt enable flip-flops *INTE1* and *INTE2*. Finally, some of the extra handshaking signals needed by D are also identified.

5.3 Mapping of interrupt on terminal count function to Intel 8254 in mode 0

Intel 8254 is a very versatile programmable timer chip which can be programmed to work in one of the following modes:

mode 0 Interrupt on terminal count (in Figure 5.4).

mode 1 Hardware retriggerable one-shot (in Figure 5.4).



(a) Intel 8255 and mode 0 (b) mode 1: input and output

Figure 5.2: Intel 8255 and mode 0, mode 1 behaviour

mode 2 Rate Generator (in Figure 5.5).

mode 3 Square Wave Generator (in Figure 5.5).

mode 4 software triggered strobe (in Figure 5.6).

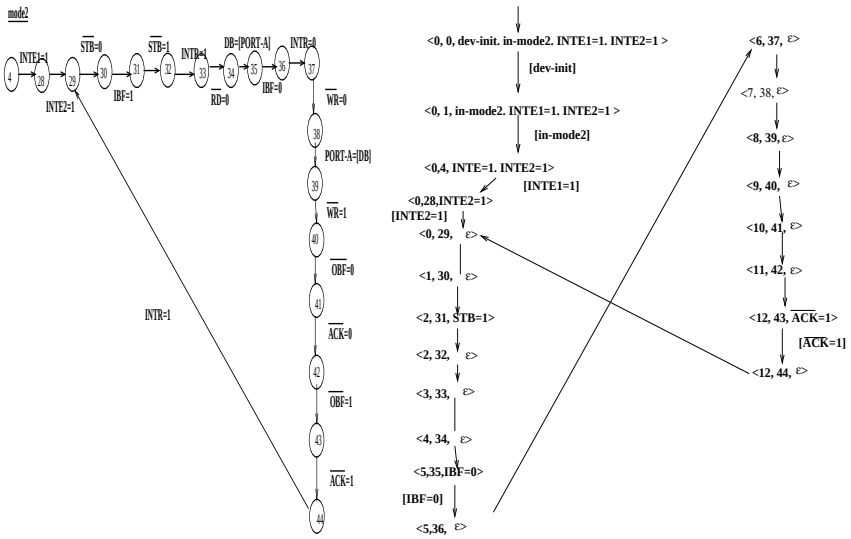
mode 5 Hardware triggered strobe (in Figure 5.6).

In this example we demonstrate the mapping of a interrupt on terminal count function (shown in Figure 5.7) to Intel 8254 in mode 0. The behaviour of the function (F) is as follows:

After initialization, a count value is loaded. Subsequently, this count value is decremented with every clock input (*clk*). Before terminal count is reached (e.g, *cnt ¿ 0*), the count value can be read by setting the *gate* input low. Decrement of count value can be continued by resetting the *gate* input to high again. When terminal count is reached (*cnt==0*), the *out* line is set to high, which can be used to interrupt the CPU if desired.

The behaviour of Intel 8255 in mode 0 is almost similar, except that it has additional control inputs to be initialized such as *out, gate* before the down counting operation can start.

The result of mapping this function to the above device is shown as an Interface in Figure 5.8. The mapping algorithm automatically performs mode selection (by forcing the *in-mode0* input of D). It also performs intermediate extra control signal generations by forcing the device transitions from state 8 to state 9, state 9 to state 10 and also state 10 to state 11.



(a) Intel 8255 and mode 2 (b) Interface

Figure 5.3: Intel 8255 in mode2 and the Interface Process

5.4 Result Summary

We now present a brief summary of the results obtained by the f−simulation tool-kit in Table 5.1. The running times were taken using the IBM JVM on a Pentium II 451 MHz server (time in seconds obtained by taking the user time field of the UNIX *time* command).

Table 5.1: Results of using the f−simulation algorithm

F	D	no of states in F	no of states in D	time in secs
lathe controller port	Intel 8255	13	45	1.20
handshake protocol	Intel 8255	10	45	.77
hardware triggerable one-shot	Intel 8254	7	54	.77
interrupt on TC	Intel 8254	8	54	.56
rate generator	intel 8254	9	54	.67
coffeeF	coffeeD	7	15	.36

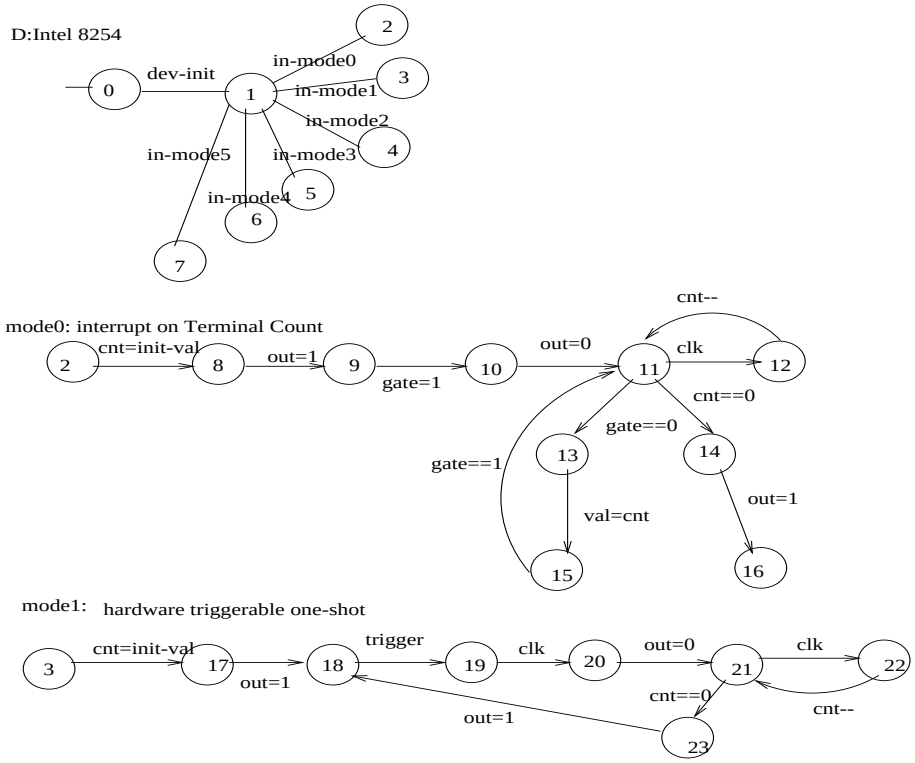


Figure 5.4: Intel 8254 abstract behaviour and also mode 0 and mode 1 behaviours

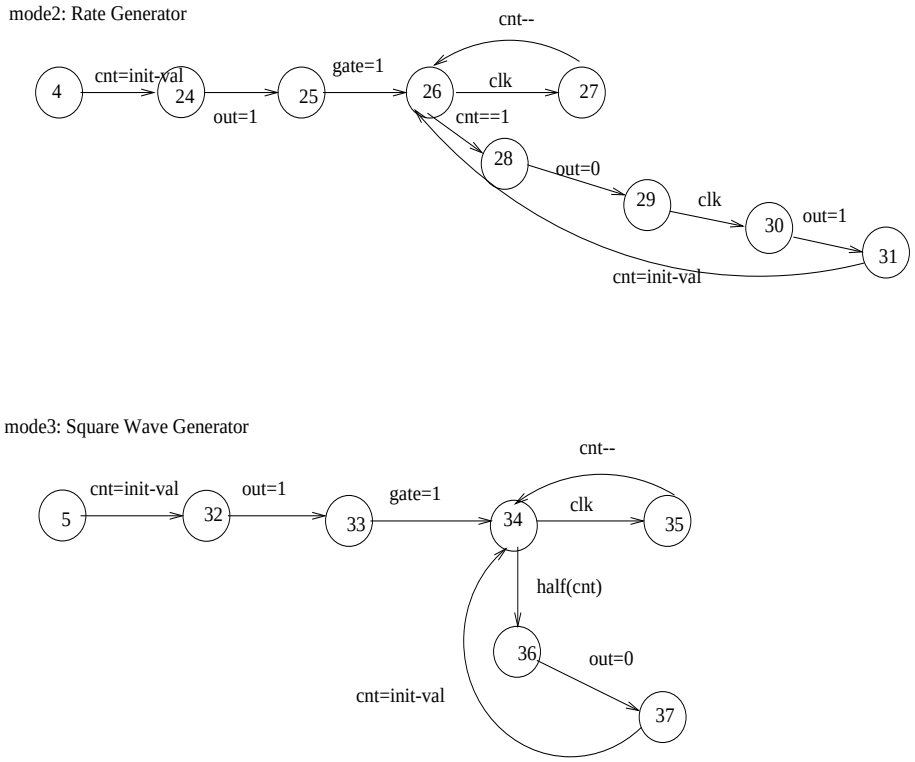


Figure 5.5: Intel 8254 in mode 2 and mode 3

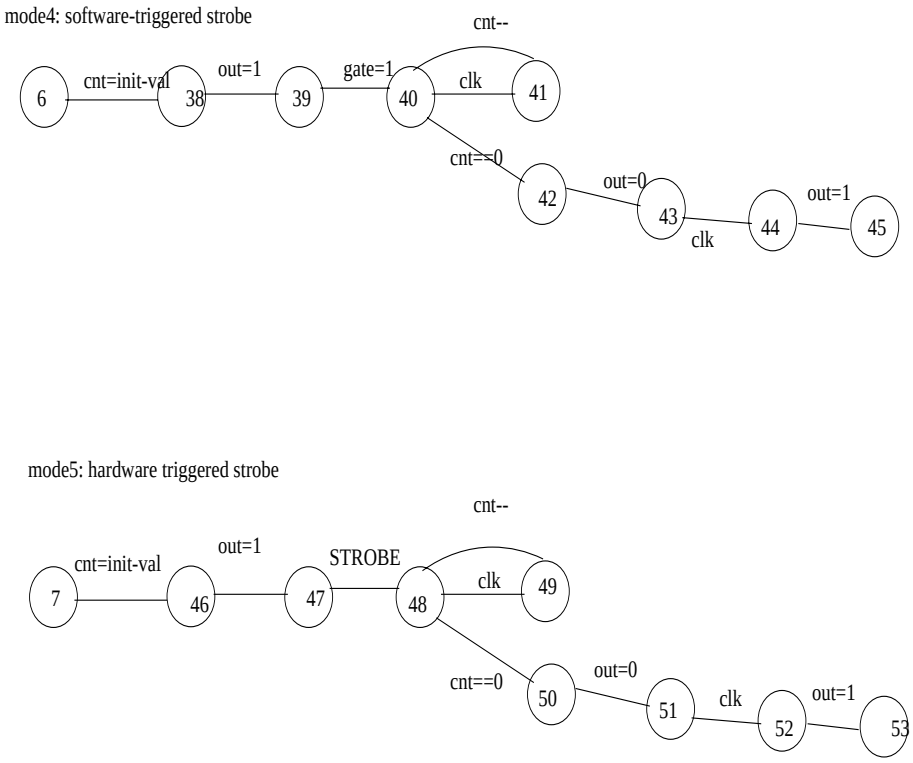


Figure 5.6: Intel 8254 mode 4 and mode 5 behaviours

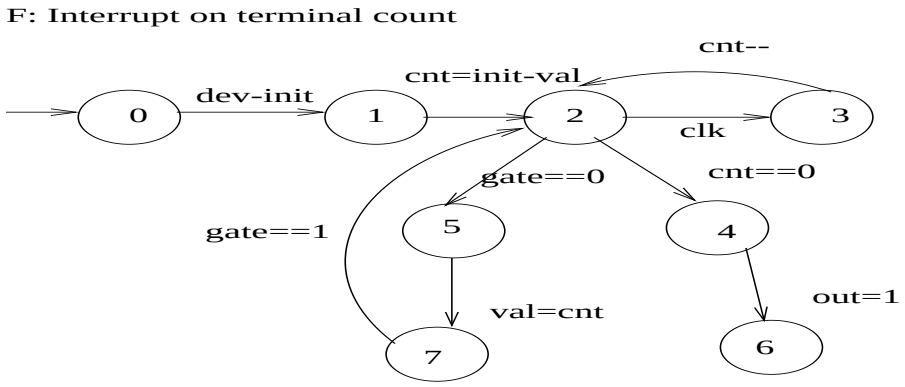


Figure 5.7: Interrupt on Terminal Count Function

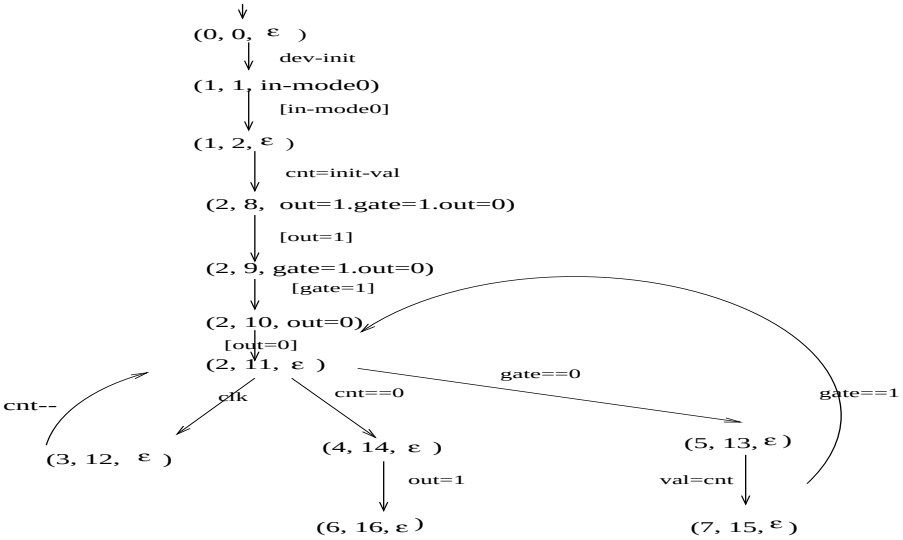


Figure 5.8: Interface Process for matching interrupt on TC to Intel 8254 in mode 0

Chapter 6

Related Work

As a starting point for automated component matching, Mitra et. al [33] proposed an algorithm for mapping a given design function to a system level device. However, this work presented an informal algorithm. Our work makes several improvements to this approach. Firstly, we have proposed a new formal simulation technique which is proven to be correct. This is very important in a component based framework to reduce validation effort when components in the library are already validated. Secondly, we have proposed a generalized notion of forcing using which we can automatically perform mode selection and extra control signal generation. Finally, the proposed interface is a generalization of the one presented in [33] which was heuristically generated.

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 [7, 21, 12, 34, 3, 9] the starting point being the pioneering work by Borriello for automatic transducer synthesis [9]. 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 later work by Narayan and Gajski [34]. In this work the behaviours of the two incompatible blocks were represented in a *hardware description language* [49, 20] 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.

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

- We concentrate on device identification by a new simulation relation, whereas 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.

- 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. Hence, we provide a decision procedure before attempting to generate the interface process.

- 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 [34] data width mismatches can be handled by latching within the interface process. This task is not currently done by our algorithm.

The proposed component matching algorithm is based on a new formal technique called forced simulation, which extends the well known notion of bisimulation proposed by Milner [32]. Bisimulation was proposed in for checking process equivalence. Bisimulation is a stricter notion than trace inclusion, based on which a set of simulation and refinement techniques have been proposed. 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 [27] the notion of refinement was extended to automata. Forward and backward simulations were later defined in [27] by Lynch and Vaandrager which are generalizations of refinement to incorporate more behaviours into the proof system.

All the above simulation relations though interesting, cannot be directly used for component matching since they are refinement based and cannot adapt a general implementation to match a specification

At a broader level, the motivation of the problem of interface synthesis is the same as that of controller synthesis [42], namely to arrive at a desired behaviour from an existing system by controlling or adapting it. But there are two major differences which make the problem and the solution different from the solutions to controller synthesis [42, 37, 28]: the interaction between the interface (or controller) and the device (or plant) are different and the relationship between the function (or specification) and the device with the interface (or controlled plant). These differences probably arise as a result of difference in the application domain.

In controller synthesis, the controller can only disable events whereas our interfaces can also force signals. In the former, the interaction is modelled by synchronous parallel composition whereas we have a novel way of interaction between the interface and device. This interaction is easily implementable in hardware. More importantly, embedded system designers often employ techniques such as writing device drivers to program a generic device to behave in a given way. This task when automatically performed amounts to a set of forcing steps of the interface.

We have a simple model of labelled transition system and the device modified by the interface and the function are related by observation equivalence which is the finest equivalence relation that can be used to identify external behaviour of reactive systems. This suffices for our application. Whereas the relationship between the controlled plant and the specification is quite varied and more complex.

As a consequence of this difference, we have a simple solution of forced simulation which in our opinion is more elegant. Also, this is keeping in trend with the fact that simulation and refinement based techniques have been widely used in the past for hardware design and verification [30, 50].

In addition to these differences, controller synthesis employs the notion of controllable and uncontrollable events and the task of controller is to control the behaviour of an uncontrolled plant (as it is called). In our problem, the device is an already developed component whose internals are not accessible. The only kind of uncontrollable events one can think of here are the internal events. Allowing any simple kind of internal events in the device makes the interface synthesis impossible since the interface has no way of knowing about their occurrence.

Chapter 7

Concluding Remarks

In this paper, we have formalized the component matching problem and proposed a algorithm based on forced simulation for component matching during embedded system synthesis. The proposed algorithm as well as the theory are novel from the perspective of CAD and simulation literature. We have implemented and tested the algorithm on a set of practical examples from the domain. The example include the specifications of two full Intel programmable components namely Intel 8254 and Intel 8255 which could be automatically initialized using our algorithm to appropriate modes for matching various functions. A preliminary version of this paper has appeared in [46].

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.

The current algorithm, though a major starting point, has some limitations. currently, it matches simple LTSs which have very simple synchronizing labels. However, to consider specifications as well as developed components which are represented in a high level specification language, we will require more complex labels in our LTS. We have recently presented in [43] some preliminary results of applications of our technique for component based development of reactive programs in Esterel [8].

The algorithm assumes that the signal names in F and D are identical. However, a specifier might use completely different signal names in comparison to the device. In such a situation, relabelling of terms must be done similar to [33]. Our algorithm can be easily extended to handle this by having a set of state-based bindings rather than the global bindings proposed in [33]. This approach will help in identifying more behaviours, which might possibly

be rejected by a global binding strategy.

Also note that the same functionality might be represented in different ways by different designers, especially while modelling the data operations. In such a situation, the current algorithm fails. We have proposed a new formal specification language called CFSMcharts [44, 47] that uses a set of generic subsidiary machines (SMs) for data operations and the abstract functionality is represented as a primary machine devoid of any explicit data operation. The PM calls suitable SMs to perform operations on data. Such hierarchical handling of data helps in overcoming the above problem by representing the core functionality only and component matching will be attempted at the PM level. We are currently lifting our matching algorithm to the CFSMcharts level.

Bibliography

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

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and analysis of algorithms.* Addison-Wesley, 1974.

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

[4] R. Ang and N. Dutt. An algorithm for the allocation of functional units from realistic RT component librarties. In *7th international symp. on high level synthesis*, May 1994.

[5] K. arnold and J. Gosling. *the Java Programming Language.* Addison Wesley, 1996.

[6] F. Balarin, M. Chiodo, P. Giusto, H. Hsien, J. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach.* Kluwer Academic Press, 1997.

[7] 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.

[8] G. Berry and G. Gonthier. The ESTEREL synchronous programming language. *Sc. Comput. Prog.*, 19:87–152, 1992.

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

[10] P. Camurati, F. Corno, and P. Prinetto. A methodology for system-level design for verifiability. In *CHARME'93.* Springer Verlag, 1993.

[11] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd. *Surviving the SOC revolution: a guide to platform based design.* kluwer academic, 1999.

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

[13] L. Dominik. Incorporating automation into embedded system testing. http://www.btree.com. Doc. no. WP000298.001.

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

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

[16] D. Harel. Statecharts : a visual formalism for complex systems. *Sci. Comput. Prog.*, 8:231–274, 1987.

[17] 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.

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

[19] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* addison Wesley, 1979.

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

[21] 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.

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

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

[24] M. Keating and P. Bricaud. *Reuse methodology manual for System-on-a-chip design.* Kluwer academic, 1999.

[25] S. Kumar. *The codesign of embedded systems: a unified hardware software representation.* Kluwer academic, 1995.

[26] S. M. Lewandowski. Frameworks for component based client erver computing. *ACM computing surveys*, 30(1), 1998.

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

[28] P. Madhusudhan and P. S. Thiagarajan. Controllers for discrete event systems via morphisms. In *CONCUR 98*, volume LNCS, 1998.

[29] F. Maraninchi. The argos language : graphical representations of automata and descriptions of reactive systems. In *IEEE Workshop on Visual Languages*, Kobe, Japan, 1991.

[30] T. F. Melham. Abstraction mechanisms for hardware verification. In C. Birtwistle and P. Subrahmanyam, editors, *VLSI specification, verification and synthesis*. Kluwer Academic Publishers, 1988.

[31] G. J. Milne. Design for verifiability. In *Hardware Specification, Verification and Synthesis: Mathematical sciences institute workshop proceedings*, pages 1–13. Springer Verlag, 1990.

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

[33] 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.

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

[35] S. Narayan, F. Vahid, and D. D. Gajski. System specification and synthesis with the SpecCharts language. In *Int. Conf. on CAD (ICCAD)*, pages 266–269, 1991.

[36] R. Orfali and D. Harkey. *Client server programming with Java and CORBA*. John Wiley and Sons, 1998.

[37] A. Overkamp. Supervisory control using failure semantics and partial specifications. *IEEE Transactions on automatic control*, 42(4), 1997.

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

[39] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 1987.

[40] A. Pnueli. *Application of temporal logic to the specification and verification of reactive systems : a survey of current trends*, volume Lecture Notes in Computer Science, Vol.224, chapter pp.510-584. Springer Verlag, 1986.

[41] R. Pooley and P. Stevens. *Using UML software engineering with objects and components.* Addison Wesley, 1999.

[42] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proc. IEEE*, 77:81–98, 1989.

[43] S. Ramesh, P. Roop, and A. Sowmya. Component based development of reactive programs. The 1999 Synchronous Workshop, Hyeres, France, Nov 29-Dec 3 1999.

[44] P. Roop and A. Sowmya. CFSMcharts: A new language for microprocessor based system design. In *11th IEEE International Conference on VLSI Design*, pages 342–346, January 1998.

[45] P. S. Roop, A. Sowmya, S. Baldwin, and G. Mormanis. Application of CFSMcharts for modelling real-time and industrial embedded systems. In *15th IFAC workshop on Distributed Computer Control Systems*, September 1998. accepted for presentation.

[46] P. S. Roop, A. Sowmya, and S. Ramesh. Automatic component matching using forced simulation. In *13th International Conference on VLSI Design*, Calcutta, India, January 2000. IEEE Computer Society Press. to appear.

[47] 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.

[48] J. Smith and G. de Micheli. Polynomial methods for component matching and verification. In *IEEE/ACM International Conference on Computer Aided Design*, 1998.

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

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

Appendix A

Theorem 1

Given $F \sqsubseteq_{fsim} D$ there exists I such that $F \approx (I//D)$.

The proof of the theorem involves the generation of an interface given $F \sqsubseteq_{fsim} D$.

Given $F \sqsubseteq_{fsim} D$, there exists an f−simulation R between F and D. For simplicity, let R be a minimal f−simulation relation such that for any pair $s_f \in S_F, s_d \in S_D$ there is at most one $\sigma \in \Sigma^*$ with $(s_f, s_d, \sigma) \in R$. The proof can be carried out, however, even for a non minimal R.

I is given by $< R, (s_{f0}, s_{d0}, \sigma_0), \Sigma_I, \rightarrow_I >$ where:

1. R is the set of states of I,
2. $(s_{f0}, s_{d0}, \sigma_0) \in R$ is the start state of I,
3. $\Sigma_I = \{[a] | a \in \Sigma\} \bigcup \Sigma$ is the set of events,
4. $\rightarrow_I$ is the transition relation defined by the following rules:
   - if $(s_f, s_d, a.\sigma) \in R$ and $s_d \xrightarrow{a} s'_d$ then $(s_f, s_d, a.\sigma) \xrightarrow{[a]} (s_f, s'_d, \sigma)$.
   - if $(s_f, s_d, \epsilon) \in R$ and $(s'_f, s'_d, \sigma') \in R$ and $s_f \xrightarrow{a} s'_f$ and $s_d \xrightarrow{a} s'_d$ then $(s_f, s_d, \epsilon) \xrightarrow{a} (s'_f, s'_d, \sigma')$.

The following set of simple observations and lemmata regarding I follow directly from the definitions of I.

Observation 1: For every $s_f \in S_F$ there exists $(s_f, s_d, \sigma) \in S_I$ for some $s_d \in S_D$ and $\sigma \in \Sigma^*$.

Observation 2: Suppose $s \in S_I$ is $(s_f, s_d, a.\sigma)$ Then, $Lab(s) = \{[a]\}$.

Observation 3: Suppose $s \in S_I$ is (s_f, s_d, ϵ). Then, $Lab(s) = Lab(s_f)$.

Observation 4: I is well formed.

Observation 5: I is deterministic.

Based upon the observations, we can prove a set of lemmas useful for the proof of the theorem.

Consider the composition of I and D, $I//D =< S_{(I//D)}, (s_{i0}, s_{d0}), \Sigma_{(I//D)}, \rightarrow_{(I//D)} >$. Then, the following lemmas state some properties of $I//D$.

Lemma 1: For any state $s = ((s_f, s_d, \sigma), s'_d) \in S_{(I//D)}$ then $s_d = s'_d$.

For the following lemmas let $s, s' \in S_{(I//D)}$ such that s be $((s_f, s_d, \sigma), s_d)$ and s' be $((s'_f, s'_d, \sigma'), s'_d)$.

Lemma 2: $s \xrightarrow{\tau} s'$ iff $s_f = s'_f$, $\sigma = a.\sigma'$ and $s_d \xrightarrow{a} s'_d$ for some $a \in \Sigma$.

Corollary 1: Let $\sigma \neq \epsilon$. Then there exists s''_d such that $s_d \xrightarrow{\sigma} s''_d$ and $s(\xrightarrow{\tau^*})((s_f, s''_d, \epsilon), s''_d)$. Note that $\xrightarrow{\sigma}$ is the usual transitive closure over $\xrightarrow{a}$ and $(\xrightarrow{\tau^*})$ is the reflexive and transitive closure over $\xrightarrow{\tau}$.

Lemma 3: $s \xrightarrow{a} s'$ iff $s_f \xrightarrow{a} s'_f$ and $\sigma = \epsilon$.

Lemma 4: $I//D$ is deterministic.

Now, to prove the main theorem, we define a relation $\mathcal{S}$ over states of F and $I//D$ as follows:

For any $s_f \in S_F$ and $((s'_f, s_d, \sigma), s_d) \in S_{(I//D)}$,

$s_f \mathcal{S}\ ((s'_f, s_d, \sigma), s_d)$ iff $s_f = s'_f$.

Lemma 5:

$\mathcal{S}$ is a weak bisimulation over F and $I//D$

Proof:

To prove that $\mathcal{S}$ is a weak bisimulation, we have to establish that:

1: $s_{f0} \mathcal{S}\ ((s_{f0}, s_{d0}, \sigma_0), s_{d0})$.

Given any $s_f \in S_F$ and $s = ((s_f, s_d, \sigma), s_d) \in S_{(I//D)}$ for some $s_d \in S_D$ and $\sigma \in \Sigma^*$,

2: For any $a \in \Sigma$, if $s_f \xrightarrow{a} s'_f$, then there exists $s' = ((s'_f, s'_d, \sigma'), s'_d)$ such that $s \stackrel{\hat{a}}{\Rightarrow} s'$ and $s'_f\ \mathcal{S}\ s'$

3: For some $\alpha \in \Sigma \bigcup \{\tau\}$, if $s \xrightarrow{\alpha} s' = ((s'_f, s'_d, \sigma'), s'_d) \in S_{(I//D)}$ then there exists s'_f such that $s_f \stackrel{\hat{\alpha}}{\Rightarrow} s'_f$ and $s'_f\ \mathcal{S}\ s'$.

1 follows directly from the definition of $\mathcal{S}$.

To prove 2 assume $s_f \xrightarrow{a} s'_f$, for some $a \in \Sigma$.

If $\sigma = \epsilon$, then by Lemma 3, $s \xrightarrow{a} s' = ((s'_f, s'_d, \sigma'), s'_d)$. Hence $s'_f\ \mathcal{S} s'$.

Otherwise, by Corollary 1 $s(\xrightarrow{\tau^*})s''$ for some $s'' = ((s_f, s''_d, \epsilon), s''_d)$, $s''_d \in S_D$.

¿From Lemma 3 and our assumption we can conclude that $s'' \xrightarrow{a} s' = ((s'_f, s'_d, \sigma'), s'_d)$ for some $s'_d \in S_D$ and $\sigma' \in \Sigma^*$.

This implies that $s \stackrel{a}{\Rightarrow} s'$ and $s'_f \mathcal{S} s'$.

To prove 3 let $s \xrightarrow{\alpha} s'$. Then there are two cases:

- $\alpha = \tau$. $s_f = s'_f$ by Lemma 2, which inturn implies $s_f \stackrel{\hat{\tau}}{\Rightarrow} s_f$ and hence $s_f \; \mathcal{S} \; s'$.
- $\alpha \neq \tau$. By lemma 3, $\sigma = \epsilon$ and $s_f \xrightarrow{\alpha} s'_f$. Hence, $s'_f \; \mathcal{S} \; s'$, which proves Lemma 5 and hence Theorem 1.

We conclude this section with the proof of theorem 2 which showws the existance of $\sqsubseteq_{fsim}$ as a sufficient condition for I.

Appendix B

Theorem 2

If there exists a well formed and deterministic interface I such that $F \approx I//D$ then $F \sqsubseteq_{fsim} D$.

Proof:

Assume that $F \approx I//D$. Therefore, there exists a weak bisimulation relation $\mathcal{S}$ over F and $I//D$. Given $\mathcal{S}$ we construct a new relation $R \subseteq S_F \times S_D \times \Sigma^*$.

By assumption I is well-formed and deterministic. Also D is deterministic. It is easy to show that then $(I//D)$ is deterministic. Further, for any state s in $(I//D)$, $Lab(s)$ is either $\{\tau\}$ or a subset of Σ.

Let R be the smallest relation such that (s_f, s_d, σ) is in R iff there exists $s_i \in I$ and $(s_i, s_d) \in S_{I//D}$ with $(s_f, (s_i, s_d)) \in \mathcal{S}$ and one of the following holding:

1. $\sigma = \epsilon$ and $Lab((s_i, s_d)) = Lab(s_f)$.
2. $s_i \stackrel{[\sigma]}{\rightarrow} s_i'$ and $Lab(s_i')$ is not a forcing symbol and $Lab((s_i, s_d)) = \{\tau\}$ where $[\sigma]$ is the sequence of forcing symbols appearing in σ.

It is easy to establish that R as defined is a forced simulation relation.