An Efficient IP Matching Tool using Forced Simulation

P. S. Roop	A. Sowmya	S. Ramesh	Haifeng Guo
Department of EEE	School of CSE	Department of CSE	Department of CS
University of Auckland	University of New South Wales	Indian Institute of Technology	State University of New York
Auckland - 1	Sydney, 2052	Bombay 400 076	Stony Brook, NY 11794-4400
p.roop@auckland.ac.nz	sowmya@cse.unsw.edu.au	ramesh@cse.iitb.ernet.in	haifeng@cs.sunysb.edu

UNSW-CSE-TR-0112-December 2001

December 10, 2001

Contents

1 Introduction **3**

1.1 Motivation . . . 3

1.2 Related Work . . . 4

2 Forced Simulation: Formal Framework for Matching **5**

2.1 Reusing a general purpose port, Intel 8255 . . . 5

2.1.1 Reuse via Interface . . . 6

2.2 Formalization . . . 7

2.3 The IP Matching Problem . . . 8

2.4 Condition for Matching: Forced Simulation . . . 8

2.5 Results . . . 9

3 Logic Programming Based IP Matching Tool **10**

3.1 Encoding of forced simulation in XSB . . . 10

3.2 Complexity . . . 12

4 Interface Generation using the XSB Justifier **13**

5 Results of using the MatchMaker Tool **14**

6 Conclusions **15**

List of Figures

2.1 Abstract behaviour of Intel 8255 5
2.2 Intel 8255 in mode 2 6
2.3 The port of a lathe controller 7
2.4 An Interface Process 8

4.1 Justification evidence for (a) external move (matching) (b) forced move (forcing) 13

Abstract

Automatic IP (Intellectual Property) matching is a key to reuse of IP cores. This report presents an efficient IP matching algorithm which can check if a given programmable IP can be adapted *to match a given specification. When such adaptation is possible, the algorithm also generates a device driver (an interface) to adapt the IP. Though simulation, refinement and bisimulation based algorithms exist, they cannot be used to check the adaptability of an IP, which is the essence of reuse. The IP matching algorithm is based on a formal verification technique called* forced simulation. *A forced simulation based matching algorithm is implemented using a logic programming environment, which provides distinct advantages for encoding such an algorithm.The prototype tool, MatchMaker, has been used to reuse several programmable IPs achieving on an average 12 times speedup and 64 % reduction in code size in comparison to previously published algorithm.*

Chapter 1

Introduction

1.1 Motivation

Design reuse has been the focus of research [7, 10] 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, and most importantly the need for developing solutions that can be easily verified-often referred to as *design for verifiability* [3]. The advantages of design reuse are improved productivity, better performance and, more significantly, better quality products.

In a recent survey on digital design reuse [7] several key factors for successful reuse of Intellectual Property (IP) cores have been identified. These include, among others, the need for synthesis techniques to support the reuse of predesigned IP blocks. The authors also identify a need for enabling reuse of a set of prevalidated IPs from a library. We believe that, these features are not appropriately addressed in current synthesis tools.

To automatically reuse IPs during synthesis, a subset of IPs is normally indexed from a database. Subsequent to indexing, matching is essential to identify the exact IP (a device D) that is functionally equivalent to the design function (F). Zhang et al. [14] have proposed a fuzzy logic based scoring and aggregation technique for indexing IPs from a distributed database that may possibly match a given F. After indexing, precise functionality matching is not proposed as part of this work and the authors suggest simulation as a means for precise IP matching.

Simulation can be a tedious and time consuming activity. In addition, exhaustive simulation may not guarantee that the selected IP is the desired one. This problem is even more severe with programmable or parameterisable IPs which may not exactly match F but can be programmed via a device driver (an interface) to match F. In this report, we propose an algorithmic technique for automatically deciding whether a given D matches F. The algorithm, when successful, automatically generates an interface I that can *adapt* D to match F. The basis of this algorithm is a *formal verification* technique developed by us called *forced simulation* [11] and hence is guaranteed to produce correct matches. In this report, we recast forced simulation into a logic programming framework called XSB [1] for the following reasons:

1. Ease of implementation: the resulting XSB code is extremely compact and readable. In contrast to about 1500 lines of Java code spanning 5 classes to compute forced simulation using partition-refinement based approach [11], we used just 82 lines of XSB code.

2. Efficiency: the complexity of the XSB based algorithm was less by a factor of NS_f, the size of the function specification. As a result we achieved an average speed up of about 12 times compared to the implementation in [11].

3. Interface generation as a side effect: XSB justifier was used to obtain an interface without a second pass over the constructed forced simulation relation. Also, when the matching fails, the justifier can be used to discover the reasons for failure.

1.2 Related Work

The task of IP matching is similar to simulation/refinement based verification in the sense that both need to establish whether a given IP (an implementation) meets all requirements of a specification. Several simulation/refinement techniques [2, 8], have been proposed in literature. During IP matching, however, a generic implementation often needs to be matched to a given specification. This is the essence of reuse. None of the existing simulation techniques can be used to *adapt* a generic implementation for a given specification. Forced simulation overcomes these limitations.

Forced simulation is a bisimulation variant and hence standard partition-refinement based bisimulation algorithms [4] may be adapted for computing forced simulation [11]. In this report we demonstrate the advantages of using XSB, a tabled logic programming system, to implement a formal IP matching tool. XSB has already been used for building efficient model checkers and bisimulation checkers [12]. We employed some features such as tabling and constraint propagation to build an efficient matching tool.

This report is organized as follows: In chapter 2 we develop the formal framework for IP matching using forced simulation. In chapter 3 we show how the matching algorithm can be encoded in XSB logic programming environment. In chapter 4 we show how the XSB justifier is used to automatically generate the interface after successful matching and also to explain failed matches. In chapter 5 we present some results of using the matching tool-kit developed in XSB and compare this to a conventional partition-refinement based implementation in Java. The final chapter is devoted to concluding remarks.

Chapter 2

Forced Simulation: Formal Framework for Matching

This chapter presents the formal framework of forced simulation as the basis of the matching algorithm. Since we saw obvious advantages of encoding the rules for computing the forced simulation relation as a logic program, we had to redefine the rules in an equivalent form to facilitate such an encoding. Hence, the theory of forced simulation as presented here is equivalent though different from [11].

At the outset, we present the ideas using an example of a programmable port.

2.1 Reusing a general purpose port, Intel 8255

Intel 8255 [6] is a general purpose 8-bit parallel port. It can be used to interface I/O devices to a CPU. It has three 8-bit ports called PORT-A, PORT-B, and PORT-C, of which PORT-A and PORT-B are I/O ports, whereas PORT-C can be used as both an I/O port and control lines, depending on the mode of operation.

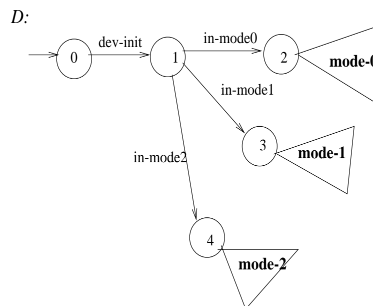

Figure 2.1. Abstract behaviour of Intel 8255

Intel 8255 may be programmed to be in one of the following modes (Figure 2.1 shows the abstract behaviour of the three modes and Figure 2.2 is the detailed behaviour in mode 2, which is the mode of interest to this example):

1. mode 0 (Basic input-output mode): CPU may read contents of a port or write some data to it.

2. mode 1 (Strobed input-output mode): This is mainly for reading or writing from a port using handshake protocol.

3. mode 2 (Bidirectional Bus): This mode is like mode 1 except that it can be used for bidirectional data transfer, while mode 1 is unidirectional.

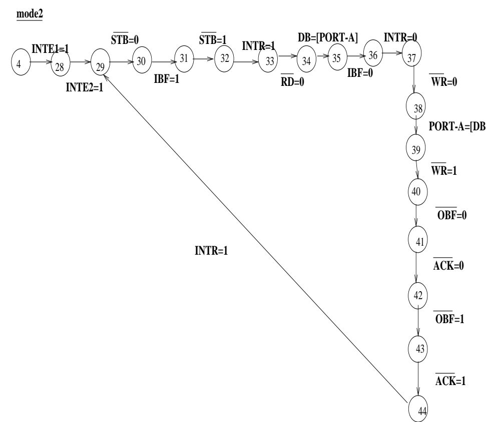

Figure 2.2. Intel 8255 in mode 2

Consider the following specification of a lathe controller port. Figure 2.3 provides an abstract description of a typical port in a lathe controller. 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 writes appropriate lathe instructions to the port, which are read by the lathe to perform the appropriate lathe action. Figure 2.3 gives the abstract description of each transition of the handshaking sequence as comments.

2.1.1 Reuse via Interface

Let F be the lathe controller port as shown in Figure 2.3 and let D be Intel 8255 as shown in Figure 2.1 and 2.2. In order for F to match D we need a device driver (an interface, I) which can dynamically adapt D in the following way:

1. When D is in state 0, I must provide the device initialization command *dev-init* followed by required mode word *in-mode2* to bring D to state 4. Such actions of the interface is known as *forcing*. After D reaches state 4, I must continue to force *INTE1=1, INTE2=1*, which are extra control signals of D not present in F. Forcing is not just required from the initial state but may be applied in any other state when required (such as say forcing of *IBF=0* when D is in state 35). In the interface, forcing signals are enclosed in [ ] to clearly distinguish them from other signals.

2. When D reaches state 29 and F is still in state 0, the interface must enable matching of transition $0 \rightarrow 1$ in F with transition $29 \rightarrow 30$ in D. Such actions of the interface is known as *matching*. While matching,

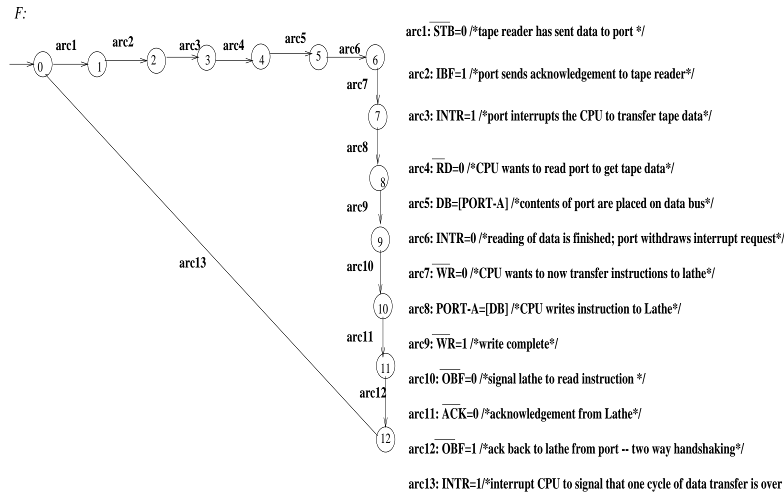

Figure 2.3. The port of a lathe controller

disabling may also be required when D has extra transitions from the matching state, not present in F (not required in this example).

The interface must perform such forcing and matching steps until the desired behaviour is realized. Such an interface is shown in Figure 2.4.

This example raises the following additional questions:

1. Given arbitrary pairs of F and D, how do we decide whether an interface exists or not?
2. Given an interface for known pair of F and D, how can we be sure that D implements all behaviours in F? In other words, how do we know that the interface is correct?

To address these issues, we propose a formalization of the above problem as illustrated in the next section.

2.2 Formalization

In order to be able to answer the questions posed in the previous section, we model F, D and I using labelled transition systems [9] (which are standard models of reactive processes). We then define a new simulation called forced simulation over F and D and show that forced simulation is a necessary and sufficient condition for F to match D.

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: S is a finite set of states, $s_0 \in S$ is a unique start state, Σ is a finite set of events or signals and $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation.*

In this report, we assume that all processes are deterministic. Let LTSs $F =< S_F, s_{f0}, \Sigma, \rightarrow_F >$ and $D =< S_D, s_{d0}, \Sigma, \rightarrow_D >$ stand for the function and device processes respectively.

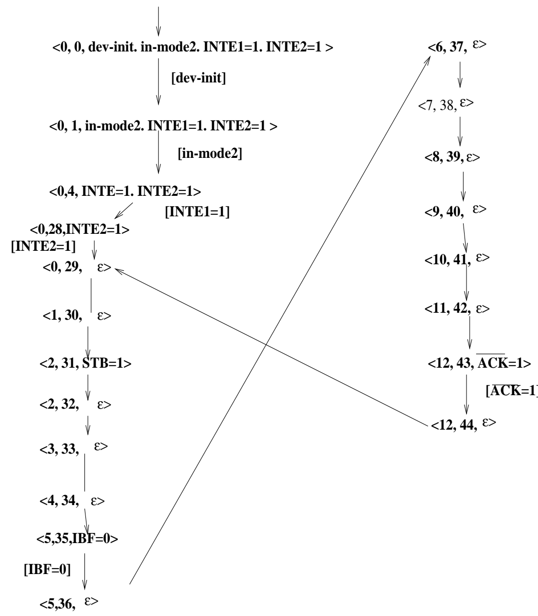

Figure 2.4. An Interface Process

Definition 2 *An interface process is a process whose set of events is* $\Sigma \cup \{[a] | a \in \Sigma\}$.

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

2.3 The IP Matching Problem

The IP matching problem is formalized as follows:

Definition 3 *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 [9] and $//$ is a parallel composition operator [11].*

In the following section we define forced simulation as a relation between the states of F and D and then establish that it is a necessary and sufficient condition for IP matching.

2.4 Condition for Matching: Forced Simulation

A pair F and D are said to be *forced similar* if there exists a relation R relating states of F to some states in D.

Two states s_f and s_d are related over R if either they are *directly related* or *related via a forcing sequence* σ.

Two states s_f and s_d are *directly related* if for every transition $s_f \stackrel{a}{\to} s'_f$ in F, there is a matching transition $s_d \stackrel{a}{\to} s'_d$ in D and further s'_f and s'_d are also related over R. In this case, $(s_f, s_d) \in R_\epsilon$.

Two states s_f and s_d are *related via a forcing sequence* σ if there exists a successor state s'_d in D such that s'_d is reachable from s_d via path σ and further s_f and s'_d are directly related. In this case, $(s_f, s_d) \in R$.

Whenever s_f and s_d are related, we shall say that the states of the tuple (s_f, s_d) *match*.

Consider the example of F and D from section 2.1. State 0 in F (denoted s_{f0}) is not directly related to state 0 in D (denoted s_{d0}) as they do not have matching transitions. However, s_{f0} is directly related to s_{d29} as they have matching transitions and their successors s_{f1} and s_{d30} are directly related. As s_{d29} is reachable from s_{d0} by a path triggered by events $dev - init, in - mode0, INTE1 = 1, INTE2 = 1$, s_{f0} and s_{d0} are related by a forcing sequence. It is easy to examine that s_{f0} is related by forcing sequences to s_{d1}, s_{d4} and s_{d28}. We can continue in this manner to relate states of F and D to build a relation called force simulation, defined below.

Definition 4 *Given LTSs F and D, a relation $R \subseteq S_F \times S_D$ is a forced simulation relation (in short, an $f-$simulation relation where $s_f R s_d$ is a shorthand for $(s_f, s_d) \in R$) provided:*
$\forall s_f \in F, s_d \in D, (s_f R s_d \Rightarrow (s_f R_\epsilon s_d) \vee (\exists \sigma \in \Sigma^*, \|\sigma\| \leq NS_d : s_d \xrightarrow{\sigma} s'_d . s_f R_\epsilon s'_d))$
where $\forall s_f \in F, s_d \in D, (s_f R_\epsilon s_d \Rightarrow \forall s_f \xrightarrow{a} s'_f, \exists s_d \xrightarrow{a} s'_d . s'_f R s'_d)$.

Forced simulation, as originally proposed [11], required a forcing sequence σ to be associated with a device state s_d that will force it to be similar to a function state s_f. Intuitively, the role of the forcing sequence is to indicate the sequence of inputs required to guide s_d to a successor state, s'_d, reachable from s_d such that s'_d has similar behaviour as s_f. However, for logic programming encoding, carrying around explicit σ's is a bottleneck. The above definition overcomes this.

Definition 5 *$F \sqsubseteq_{fsim} D$ whenever there exists an $f-$simulation relation between them.*

2.5 Results

The following two theorems establish that forced simulation is a necessary and sufficient condition for IP matching (where F and D are represented as LTSs).

Theorem 1 *Given $F \sqsubseteq_{fsim} D$ there exists I such that $F \approx (I//D)$.*

Theorem 2 *If there exists a well-formed and deterministic interface I such that $F \approx I//D$ then $F \sqsubseteq_{fsim} D$.*

The proof of both these theorems are constructive and appear in [11]. Theorem 1 and 2 forms the formal basis (necessary and sufficient condition) for IP matching. As is obvious, a key to IP matching is identifying an appropriate interface to adapt the IP. The first step is to identify a forced simulation relation and then construct an interface from this. The next chapter demonstrates how and why a logic programming tool is employed for these tasks.

Chapter 3

Logic Programming Based IP Matching Tool

Forced simulation is an extension of a standard verification technique called bisimulation [9]. Normally, bisimulation algorithms are implemented using bottom-up partition refinement [4]. We have already developed a similar bottom-up algorithm for forced simulation [11]. This algorithm starts by constructing a global relation R which is initialized to $S_f \times S_d \times RS(D)$ where $RS(D)$ is the set of all reachable states in D. Let NS_f, NS_d denote the number of states of F and D respectively. The size of this set R is this $O(NS_f \times NS_d \times NS_d^2)$ since $RS(D)$ is $O(NS_d^2)$. $RS(D)$ is required for computing the forcing sequences. After computing R the algorithm refines it by using partition-refinement like algorithm until a greatest fixed point is reached. The complexity of this implementation has been shown to be $O(NS_f^2 \times NS_d^2 \times m)$ where m denotes the maximum number of transitions in D.

Unlike bottom-up algorithm [11] that constructs the maximal relation and refines it until the greatest fixed point is reached, XSB operates in a top-down manner, i.e., explores only states that are required to prove (or disprove) that a given pair of (s_f, s_d) are *forced similar*. This local decision making results in a reduction in complexity by a factor of NS_f. XSB based encoding is very small, simple and hence readable. This happens because, unlike the top-down implementation which requires classes to be written for creation, manipulation and traversal of graph data structures, XSB uses a set of facts to represent the graphs corresponding to F and D. Also forced simulation is encoded as a set of rules. Additionally, XSB justifier can be used to construct the simulation interface.

3.1 Encoding of forced simulation in XSB

We encode the dual of forced simulation as a tabled logic program using XSB. This is essential since XSB is a least fixed point engine whereas forced simulation is computed as a greatest fixed point (we compute the greatest forced simulation relation between F and D when it exists). The rules for a pair of states to be not-forced-similar are directly encoded as XSB clauses. Input datasets F and D are encoded as a set of facts representing the adjacency matrix of the corresponding LTSs.

In the following, we provide the formal definition and the corresponding encoding in XSB. Suppose R is a forced simulation relation, then $\overline{R}$, the dual of R, is defined as :

$\forall s_f \in F, s_d \in D, (s_f \overline{R} s_d \Leftarrow (s_f \overline{R_\epsilon} s_d) \wedge$
$(\forall \sigma \in \Sigma^*, \|\sigma\| \leq NS_d : s_d \xrightarrow{\sigma} s'_d . s_f \overline{R_\epsilon} s'_d))$

where, $s_f \overline{R_\epsilon} s_d \Leftarrow \exists s_f \xrightarrow{a} s'_f, \forall s_d \xrightarrow{a} s'_d . s'_f \overline{R} s'_d$

Also, note that since D is a deterministic LTS, we can rewrite the above by replacing the $\forall$ quantification to a $\exists$ giving:

$s_f \overline{R_\epsilon} s_d \Leftarrow \exists s_f \xrightarrow{a} s'_f, \exists s_d \xrightarrow{a} s'_d . s'_f \overline{R} s'_d$.

This definition essentially encodes the fact that a pair (s_f, s_d) are not forced similar (denoted $s_f \overline{R} s_d$) when s_f is not directly forced similar to s_d (denoted $s_f \overline{R_\epsilon} s_d$) and all reachable states s'_d from s_d are also not directly forced similar to s_f. A given pair (s_f, s_d) are not directly forced similar whenever s_f has a transition that cannot be matched to any transition from s_d.

$\overline{R}$ is the least model of the above formula. We encode $\overline{R}$ as a logic program in XSB as follows:

```
:- table nfsim/2.
nfsim(SF, SD) :-
  nfsim_e(SF, SD),
  findall((SD,SD1),
          (reach(SD,SD1), SD\=SD1), SL),
  all_nfsim_e(SF, SL).
all_nfsim_e(_, []).
all_nfsim_e(SF, [(SD, SD1)|R]) :-
  nfsim_e(SF, SD1),
  all_nfsim_e(SF, R).
nfsim_e(SF, SD) :-
  trans(SF, A, _),
  \+ trans(SD, A, _).
nfsim_e(SF, SD) :-
  trans(SF, A, SF1),
  trans(SD, A, SD1),
  nfsim(SF1, SD1).
```

where nfsim/2, a tabled predicate, denotes the dual relation $\overline{R}$, and nfsim_e/2 the relation $\overline{R_\epsilon}$. `findall(X, Goal, List)` collects all the instance of `X` to `List` such that `Goal` is provable. If `Goal` is not provable, `List` will be an empty list `[]`. Predicate `all_nfsim_e/2` is used to verify the relation $\overline{R_\epsilon}$ between a state in S_F with each state from the given list of states in S_D where `\+` stands for negation. The tabled predicate `reach/2` is used to determine reachable successors and is encoded as:

```
:- table reach/2.
reach(S, S1) :- trans(S, _A, S1).
reach(S, S1) :- trans(S, _A, S2),
                reach(S2, S1).
```

An LTS $< S, s_0, \Sigma, \rightarrow >$ is encoded as a set of facts in a logic program P such that whenever $s \xrightarrow{a} t$, then trans(s, a, t) is in P. Note that since $s, t \in S$ as well as $a \in \Sigma$ are from a finite set, they can be represented in a logic program by ground terms.

Thus, given the coding of forced simulation, and two LTSs F and D with their respective start states s_{f0} and s_{d0}, the query, to check whether $F \sqsubseteq_{fsim} D$ is satisfied or not, is `nfsim(`s_{f0}`, `s_{d0}`).`

3.2 Complexity

The complexity of the XSB implementation is $O(NS_f \times NS_d \times m \times NS_d)$. The first 3 terms come from the standard nfsim_e computation and the extra NS_d term comes from the fact that there are up to NS_d reachable states from every device state. In a realistic implementation with XSB system, the tables are organized using *trie* [13] data structures, giving close to unit-time lookups in practice. This is $O(NS_f)$ times faster than previously published implementation of forced simulation.

Chapter 4

Interface Generation using the XSB Justifier

XSB justifier [5] is to give evidence, in terms of a proof, for the query evaluation result by post-processing the tables created during query evaluation. Based on XSB justifier, we implement a forced simulation interface generator by defining a few rewriting rules from the justification results in logic programming level to the interface in forced simulation problem level. The generated interface is essentially a justification of an LTS-based device D to realize the specified LTS-based function F if there is a f-simulation relation $F \sqsubseteq_{fsim} D$, or a partial simulation that points out from which state no further simulation can be made if there is no f-simulation between F and D.

We use the same example described in section 2.1 to illustrate how to generate an interface I to validate Intel 8255 parallel port D to simulate a lathe controller port F. Two types of moves, *external* move and *forced* move, have to be explicitly shown in generated interface I, where the former is mapped from the justification evidence as shown in Figure 4(a), while the latter from the evidence shown in Figure 4(b). The details of justification evidence can be found in [5]. Thus, an interface (Figure 2.4) for the lathe controller port example can be generated by mapping the justification evidences to their corresponding moves, where the moves in [] are for forced moves, and the rest moves are external moves.

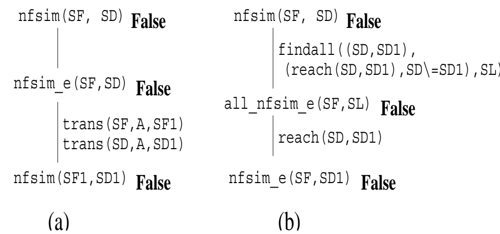

Figure 4.1. Justification evidence for (a) external move (matching) (b) forced move (forcing)

Chapter 5

Results of using the MatchMaker Tool

We have developed several examples to test the IP matching algorithm. We selected certain general purpose programmable devices such as Intel 8254 and Intel 8255. These devices can be easily reused by human designers by writing device drivers which supply appropriate mode and command words to select the desired mode. We wanted to verify if our approach can automate such tasks. We also selected a number of reactive controllers such as vending machines, coffee brewers and automobile cruise controllers to illustrate the reusability of such controllers. All these examples were encoded as labelled transition system specifications as F and D pairs. We then tested these examples using our new XSB based matching tool, MatchMaker, and a published Java based implementation (based on partition-refinement) [11].

The XSB based implementation has a total of 82 lines of XSB prolog code for forced simulation computation. In contrast, the JAVA implementation had a total of about 1200 lines of JAVA code for forced simulation computation (excluding the code required for interface generation) using the bottom-up partition-refinement based approach [11]. Since, XSB is prolog based, no extra code needed to be written for setting up the graph data structure and for traversal (which is required in JAVA). This led to considerable time saving in terms of building the prototype.

The performance comparison of the XSB implementation versus the published Java based implementation is summarized in Table 5.1. Both the implementations were tested on a Pentium III 800 MHz workstation. *On an average, the XSB implementation was 12 times faster than the Java implementation.*

Table 5.1. Results of using the matching algorithm

F	D	no of states in F	no of states in D	secs(XSB)	secs(Java)	speedup
lathe controller port	Intel 8255	13	45	0.1800	0.4900	2.72
handshake protocol	Intel 8255	10	45	0.0300	0.4400	14.66
down-counter	Intel 8254	8	54	0.1000	0.4100	4.1
simple coffee brewer	complex coffee brewer	7	15	0.0400	0.3800	9.5
coffee vending machine	beverage vending machine	3	9	0.0190	0.3900	20.52
stamp VM	postal accessory VM	5	7	0.0110	0.2600	23.63
manual car controller	cruise controller	4	9	0.0220	0.2800	12.72

Chapter 6

Conclusions

In this report, we have presented a new formal verification technique called forced simulation as a basis for matching a design function F to a IP D. Whenever a forced simulation relation exists between a pair of F, D an interface I can be constructed to *adapt* D to match F. Forced simulation forms the formal basis for automatic IP matching, which is a key task in automated IP reuse and integration. We believe that this is the first such attempt to provide a formal verification based algorithmic technique for IP matching, which is being currently tackled manually through simulation [14]. If the IP library consists of pre-validated components then the proposed approach entails a considerable saving of verification effort.

We have built a IP matching tool MatchMaker, using the XSB system and tested it by reusing several IPs from the domain of embedded systems. To achieve this, forced simulation had to be adapted for efficient encoding in XSB. The XSB based prototype performs better than conventional implementation of forced simulation [11] in terms of code size (XSB based implementation was 64 % smaller in size measured as LOC) and efficiency (complexity reduction by a factor of size of F and about 12 times average speedup).

Though the proposed approach is a novel technique for IP matching at the functional level, it has several limitations. Firstly, it is entirely non real-time being LTS based and has no mechanism for matching real-time constraints. Secondly, it is well suited to matching control units and data path matching is yet to be solved. Finally, we have to integrate this matching tool with existing IP reuse environments such as JAVA CAD.

Bibliography

[1] The xsb logic programming system v1.7, 1997. Available by anonymous ftp from ftp.cs.sunysb.edu.

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

[3] P. Camurati, F. Corno, and P. Prinetto. A methodology for system-level design for verifiability. In *Conference on Correct Hardware Design and Verification Methods '93*. Springer Verlag, 1993.

[4] J. C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13, 1990.

[5] H-F. Guo, C.R. Ramakrishnan, and I.V. Ramakrishnan. Speculative beats conservative justification. In *In Proc. 17th International Conferencene on Logic Programming (ICLP)*, pages 150–165, 2001.

[6] Intel peripheral datasheets for 82c55 programmable peripheral interface. http://developer.intel.com/designer/datashts, 1995.

[7] J. Jussel. System-on-a-chip reuse platforms can dramatically shorten design cycles. *Electronic Design*, 48(21), 2000.

[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] D. Monjau and M. Sporer. Reuse-oriented design of embedded systems. In *Fourth International Conferene on Knowledge-Based Intelligent Engineering Systems and Allied Technologies*, Brighton, UK, 2000. IEEE.

[11] Author names supressed. Forced simulation: A technique for automating component reuse in embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 2001.

[12] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. W. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *9th International Conference on Computer Aided Verification*, 1997.

[13] I. V. Ramakrishnan, Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. Efficient access mechanisms for tabled logic programs. *JLP*, 38(1):31–54, 1999.

[14] T. Zhang, L. Benini, and G. De Micheli. Component selection and matching for ip based design. In *Design Automation and Test in Europe*, Munich, Germany, 2000. IEEE.