A Formal Approach to Component Based Development of Synchronous Programs

P. S. Roop, A. Sowmya

School of Computer Science and Engineering
University of New South Wales
Sydney, 2052
Tel: (602) 9385 3980, x3936
Fax: (602) 9385 1814
e-mail: proop,sowmya@cse.unsw.edu.au

S. Ramesh

Department of Computer Science and Engineering
Indian Institute of Technology
Bombay 400 076
Tel: (91-22) 576 7722
Fax: (91-22) 572 0290
email: ramesh@cse.iitb.ernet.in

Abstract— Synchronous languages may be used for specification and design of embedded systems. Assuming the availability of a library of synchronous programs, we propose a technique to enable reuse of these programs, via an algorithm for automatic matching of a design function to a program from the library. The algorithm, when successful, generates an *interface* which automatically adapts the program. The algorithm is based on a new simulation relation called *synchronous forced simulation*, which is shown to be necessary and sufficient for matching a given pair of function and program.

I. 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 the need for developing solutions that can be easily verified-often referred to as *design for verifiability*. Intellectual property (IP) reuse in System on a Chip (SoC) [3] design is an enabling trend for component reuse.

Though component-based development has several advantages over existing methods, many unresolved issues remain to be addressed. Some of the more important ones are *developmental issues* which try to identify and develop generic products that are easily reusable, *database issues* which address how to store, index and retrieve the components, *matching issues* which decide if a component matches some requirements, and *compositional issues* which try to compose a set of matched components.

The focus of this paper is the issue of component matching for embedded systems which are application specific reactive systems. Synchronous languages such as Esterel [1] are very popular for describing embedded systems behaviourally.

During embedded system design, a system designer spends substantial amount of effort on developing and verifying synchronous programs. Once the specifications are developed standard design tools may be employed to design the target system. If these programs are suitably archived in a database, there is immense scope of reuse.

Consider a library of such synchronous programs which have been successfully designed. We use the term device D to denote a component of this library. Let F denote the specification of a target function which is to be designed.

In this paper we propose a polynomial time algorithm for reusing a D to match the requirements in a given F. The basis of the algorithm is a novel simulation relation called *synchronous forced simulation* which is also proposed by us.

This paper is organized as follows: In section 2, we provide the problem definition and motivate an informal solution. In section 3, we propose a new simulation relation called synchronous forced simulation and show that it is a necessary as well as sufficient condition for our component matching algorithm. In section 4, we present the component matching algorithm and illustrate it via a simple example. The fifth and final section makes concluding remarks.

II. Component Matching

Given F and D, component matching tries to address the question "can D be used to realise F ?"

Several techniques [5] 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 $\mathcal{S}$. Equivalence checking techniques such as bisimulation [4] have also been used for checking process equivalence.

Though both bisimulation equivalence and refinement based techniques have been widely applied to the verification of hardware, 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.

The Car Controller Example

Consider the specification of a car controller, as shown in Figure 1, which is implemented in many automatic cars. The controller waits for *ignition* to be turned on and also for the gear to be in the *parking* position. It then generates a *chk-belt* signal to check if all seat belts of seated passengers are fastened. It also starts the engine (*start_engine*) and sets the mode of operation to *manual_mode* (where the speed of the car depends on the value of throttle pressed).

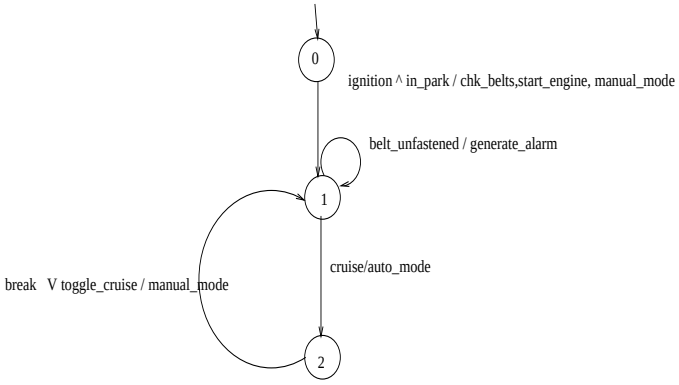

Fig. 1. complex car controller: D

If the seat belts are unfastened then an alarm is generated. The driver can set the cruise mode by setting the *cruise* control button and as a result the car drives in *auto-mode* (where the speed is determined by the speed set for cruise mode). Any time the driver presses the break or toggles the cruise control button, the car goes back to the manual mode of operation. Let this synchronous program be one of the Ds.

Consider the specification of a different car controller in which there is no checking during ignition to see if the gear is in a parked position. Also, this controller is being developed for cars in countries where seat-belts are not mandatory. As a result no checking of seat-belts are required. This car after starting drives in manual mode until the driver selects the cruise mode. It reverts back to manual mode when the driver applies the breaks. Let this be the new car controller specification F, that needs to be implemented. Behaviour of F can be described as in Figure 2.

There are a number of constraints while reusing D. We cannot access the internal states of D nor can D be modified directly. However, the sequence of events consumed by D may be observed to 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 intuition behind using a generic implementation to implement a new specification is to construct an external process which moves synchronously with D and *adapts* D, so that D then matches F. In this paper, such an external process is termed as an *interface* process. The task of the

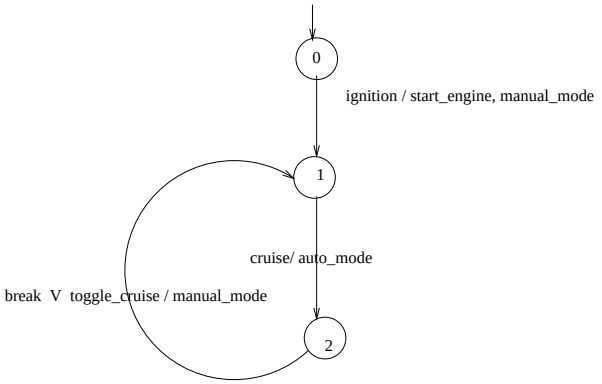

Fig. 2. simple car controller: F

interface process is to generate the inputs to D (termed as *forcing*) whenever the environment satisfies the inputs of F and then to *hide* any extra outputs of D that do not match those of F. Also, if D has any extra behaviours not present in F, then these are *disabled* by the interface.

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

1. In state zero of D when the *ignition* input is given, the interface has to *force* the *in_park* input of D. The interface must also *hide* the extra output *chk_belts* that D generates.
2. In state 1 of D the interface must *suppress* the *belt_unfastened* input of D.

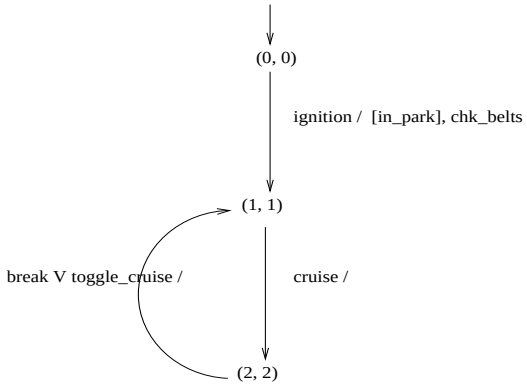

Fig. 3. interface for car controller

The interface for this example is shown in Figure 3. Any transition of the interface has an input identical to the input of the transition in F being simulated by D in that step and can have two kinds of outputs associated with the transition. The first kind of outputs corresponds to generating the inputs of D (forcing) and we enclose these outputs in '[ ]' to denote forcing outputs. The other set of outputs of the interface are the outputs of D that must be hidden. The interface can also disable extra transitions in D not matching F. In this example, the interface disables

the transition triggered by *belt_unfastened* from state 1 of D.

This example illustrates the need for adapting a generic device D to match a function F by constructing an interface and composing it with the device. Thus, given arbitrary pairs F and D which are synchronous programs the main issue is to decide whether such an interface exists and if so, to determine the interface. In the next section, we formalise this matching problem by representing F and D as input-output boolean automata (IOB) [6] (which are models of synchronous programs) and then formalising the interface generation question.

III. Synchronous Forced Simulation: a formal approach to component matching

Definition 1:

An input/output boolean automaton (IOB) [6] is a tuple $< S, s_0, I, O, \rightarrow >$, where: S is a finite set of states, $s_0 \in S$ is a unique start state, I is the set of inputs, O is the set of outputs, and $\rightarrow \subseteq S \times \mathcal{B}(I) - \{ff\} \times 2^O \times S$ denotes the transition relation. $\mathcal{B}(I)$ is a set of boolean formulas with variables in I which is isomorphic to to the set of functions from 2^I to $\{0,1\}$ and ff denotes the identically false formulas.

Let the IOB of the function $F =< S_F, s_{f0}, I_F, O_F, \rightarrow_F >$ and that of the device $D =< S_D, s_{d0}, I_D, O_D, \rightarrow_D >$

Interface Process

The main task of an interface process will be to perform forcing, disabling as well as hiding. The IOB of the interface process $\mathcal{I} =< S_\mathcal{I}, s_{i0}, I_\mathcal{I}, O_\mathcal{I}, \rightarrow_\mathcal{I} >$ where:

$\rightarrow_\mathcal{I} \subseteq S_\mathcal{I} \times \mathcal{B}(I_\mathcal{I}) - \{ff\} \times \mathcal{M}(O_\mathcal{I}) \times 2^{O_\mathcal{I}} \times S_\mathcal{I}$. Here, $\mathcal{M}(O_\mathcal{I})$ denotes the set of complete monomials over $O_\mathcal{I}$. The first set of outputs are the forcing outputs to force any extra inputs in the device guard and the second set of interface outputs are the hidden outputs. Thus, an interface transition is of the form $s_i \overset{b/[b']_{,}o_j}{\longrightarrow} s_i'$ where b denotes the boolean guard and $[b']$ and o_j denote the sets of forcing outputs and hidden outputs respectively. $[b']$ is an operator that given the boolean guard b' of D to be forced generates the appropriate $o' \in \mathcal{M}(O_\mathcal{I})$ such that $o' \Rightarrow b'$.

Having defined the interface and the device processes formally, we now define their interaction by the following rule.

Definition 2: Given $\mathcal{I}, D$ as above, $\mathcal{I}//_sD$ is defined to be a process described by an IOB, $< S_{(\mathcal{I}//_sD)}, (s_{i0}, s_{d0}), I_{(\mathcal{I}//_sD)}, O_{(\mathcal{I}//_sD)}, \rightarrow_{(\mathcal{I}//_sD)} >$ where the transition relation $\rightarrow_{(I//_sD)}$ is defined by the following rule:

$$\frac{s_i \overset{b/[b_1]_{,}o_j}{\longrightarrow} s_{i1}, s_d \overset{b_1/o}{\longrightarrow} s_{d1}}{(s_i, s_d) \overset{b/o-o_j}{\longrightarrow} (s_{i1}, s_{d1})}$$

This rule asserts that a device consuming inputs b_1 and producing outputs o can be composed with an interface that consumes inputs b producing two sets of outputs $[b_1]$ (forcing outputs) and o_j (hidden outputs). The resultant transition in the composition makes a transition consuming input b and producing outputs $o - o_j$.

Component matching

We can now formalize the matching problem. Intuitively, given F and D, we would like to determine whether there exists an I such that $I//_sD$ has equivalent behaviour to F. To define the equivalence we use the standard notion of *synchronous bisimulation* [6] for synchronous programs.

Definition 3: A device can be adapted to a function provided there exists an interface process $\mathcal{I}$ such that $F \approx_s (\mathrm{I}//_sD)$ where $\approx_s$ is the synchronous bisimulation defined over IOBs [6].

Now we give a necessary and sufficient condition for the existence of such an $\mathcal{I}$.

Definition 4:

Given IOBs F and D, a relation $R \subseteq S_F \times S_D$ is called a synchronous forced simulation relation (in short, an sf−simulation relation) provided the following holds:

1. $s_{f0} R s_{d0}$.
2. $s_f R s_d \Rightarrow (\forall b, o, s_f' : [(s_f \overset{b/o}{\rightarrow} s_f') \Rightarrow \exists s_d', b', o' : (s_d \overset{b'/o'}{\rightarrow} s_d' \wedge o \subseteq o' \wedge s_f' R s_d')])$.

Definition 5:

We say that $F \sqsubseteq_{sfsim} D$ provided there exists an sf−simulation relation between them.

Example:

Consider processes F and D as shown in Figure 4. $F \sqsubseteq_{sfsim} D$ since there exists $R = \{(0,0),(1,1),(2,2)\}$ which can be easily shown to be an sf−simulation relation.

However, R need not be unique as shown by the existence of another relation R' which is also an sf−simulation relation. $R' = \{(0,0),(1,1)\}$.

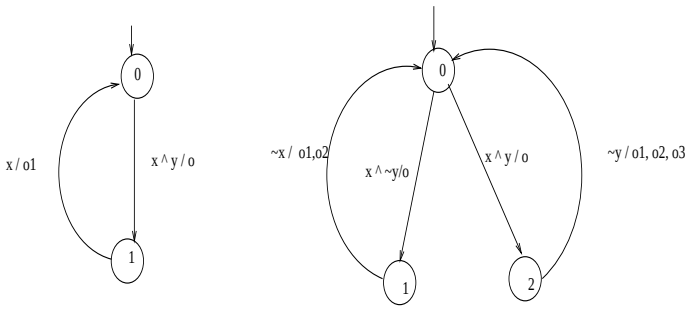

Fig. 4. sf-simulation example

Theorem 1:

Given $F \sqsubseteq_{sfsim} D$ there exists $\mathcal{I}$ such that $F = (\mathrm{I}//_sD)$.

Theorem 2:

Given a deterministic I such that $\mathcal{I}//_sD \approx_s F$, $F \sqsubseteq_{sfsim} D$.

The proofs are omitted due to space constraints; they appear in [7].

Matching Algorithm
$match(F, D)$
//F and D are the IOBs of the function and device respectively
1. $\rho_I = \{B_{s_f} | s_f \in S_F\}$ where $B_{s_f} = \{\{s_f\} \times S_D\}$
2. $\rho = \rho_I$
3. $waiting = \rho_I$
4. **repeat**
   choose and remove any $B_{s'_f} \in waiting$
   $matchB = \{B_{s_f} \in \rho | s_f \xrightarrow{b/o} s'_f$ for some $b, o\}$;
   for each $B_{s_f} \in matchB$ **do**
      for each transition $s_f \xrightarrow{b/o} s'_f$ out of s_f **do**
         if $|reduceB| < |B_{s_f}|$ **then**
            $reduceB = Reduce(B_{s_f}, o, B_{s'_f})$
            $\rho = \rho - B_{s_f} \bigcup reduceB$
            $waiting = waiting - B_{s_f} \bigcup reduceB$
         endif
      endfor
   endfor
until $waiting = \phi$
if any $(s_f, [\,]) \in \rho$ where $s_f \in S_F$ **then**
   return FALSE;
else
   $Generate\_Interface(\rho)$
endif
//the Reduce() function
$Reduce(B_{s_f}, o, B_{s'_f})$
$reduceB = \{\{s_f\} \bigcup \{s_d \in B_{s_f} | \exists s'_d \in B_{s'_f} \wedge \exists o' \subseteq O_D \wedge s_d \xrightarrow{b'/o'} s'_d \wedge o \subseteq o'\}$
return $reduceB$

Fig. 5. component matching algorithm

IV. Matching Algorithm

We now give an algorithm for matching a function to a device in Figure 5. The inputs to the algorithm are the IOBs of the function and device, F and D respectively. The algorithm is an adaptation of a standard bismulation algorithm [4].

The essential idea of the matching algorithm is to start with a initial set of blocks (ρ_I) such that in each block a function state s_f is paired with all states of D. To start with, we have as many blocks as the number of states in F. After initialization, the refinement process starts.

Once the initial set of blocks ρ_I is computed, a set of blocks *waiting* and another set of blocks ρ are also initialized to ρ_I. *waiting* denotes the set of blocks that are waiting to be picked up as the refining blocks and ρ denotes the set of blocks being refined and will be the final output of the algorithm. The initialization steps are steps one, two and three in Figure 5.

The refinement process is carried out as a set of iterations (step 4) where in each iteration one block is arbitrarily removed from *waiting* as the refining block and all blocks in ρ are refined based on this refining block and an output set o (which is the output on the current transition from s_f to s'_f under consideration).

The refinement process stops when *waiting* is empty. The algorithm outputs ρ which contains a set of blocks where each block contains all device states s_d that are sf-similar to a given s_f.

Interface Generation

Given the output of the matching algorithm ρ, it is easy to construct an abstract interface of the form shown in Figure 6.

This abstract interface has to be finally realized in hardware. The realization involves computation of $[b]$ for each forcing output. This is in general a hard problem. However, efficient algorithms using BDDs [2] can be used for computing $[b]$. We are currently investigating this problem.

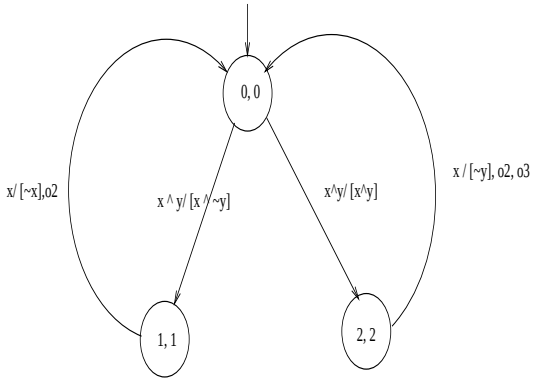

Fig. 6. specification of abstract interface

V. Conclusion

In this paper, we have formalized the component matching problem and proposed a algorithm based on synchronous forced simulation for component matching. Given a library of synchronous programs which have been implemented and a new system to be developed, we can use the matching algorithm to automatically adapt a suitable component from the library to arrive at the new implementation. The proposed algorithm has polynomial time complexity.

There are issues such as indexing of components in the library and also composition of matched components that this paper does not address. We are currently exploring these possibilities.

VI. Acknowledgement

The first two authors thank Australian Research Council for partial support of this project; the third author thanks an Indo-US project.

References

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

[2] R. E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *International Conference on Computer Aided Design (ICCAD)*, 1995.

[3] 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.

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

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

[6] F. Maraninchi and N. Halbwachs. Compositional semantics of nondeterministic synchronous languages. In *Proc. of ESOP'96*, LNCS Vol. 630, 1996.

[7] Parthasarathi Roop. *Forced Simulation: A Formal Approach to Component Based Development of Embedded Systems.* PhD thesis, Computer Science and Engineering, University of New South Wales, 2000. under review.