Efficient Methods for Embedded System Design Space Exploration *

Harry Hsieh, Dpt. of EECS, University of California at Berkeley, USA
Felice Balarin, Cadence Berkeley Laboratories, USA
Luciano Lavagno, Politecnico di Torino, Italy
Alberto Sangiovanni-Vincentelli, Dpt. of EECS, University of California at Berkeley, USA

Abstract

Design space exploration is the process of analyzing several functionally equivalent alternatives to determine the most suitable one. The synchronous assumption has made it possible to develop efficient procedures for establishing functional equivalence between different implementations in the domain of synchronous circuits as well as in the domain of synchronous reactive systems. We extend this notion to embedded systems that *do not satisfy the synchronous assumption inside their boundaries but only at the interface with the environment*. Leveraging this property, we developed efficient synchronous equivalence analysis algorithms for embedded systems with loops and architectures with multiple computational units. We demonstrate our method on an ATM switch containing many interacting components.

1 Introduction

Design space exploration is the process of analyzing several "correct" implementation alternatives to determine the most suitable one. Current methods for design space exploration of embedded system implementations tend to be quite informal and error prone. The designer often starts with an informal specification, writes a reference (golden) model in some executable language (e.g. VHDL, Verilog, or C), and executes it on a computer to make sure it is "correct" according to the original specification. One or more implementation alternatives are then generated manually or semi-automatically. Their correctness is established via "intuition" and by comparing the simulation output with the reference model. The best implementation is chosen based on issues such as performance, cost, reliability, and flexibility.

In order to formalize and clarify the design procedure for embedded system, we advocate the principle of "separation of concerns". The notions of correctness and performance should be separately defined. An example of this separation is the synchronous design methodology for sequential circuits [1] where functional correctness and timing performance can be analyzed separately. This powerful approach has been extended to higher level of abstraction by synchronous languages [2], and further extended to more general models of computation by the definition of synchronous assumption and synchronous equivalence for embedded system implementations [3].

Under the *synchronous assumption* (also known as generalized fundamental mode in the asynchronous circuit domain), the design is assumed to operate synchronously with the environment. Interaction with the environment is allowed only when the design itself has "stabilized". A design has *stabilized* when no more execution of the components is possible without processing more inputs. The operation of the design is therefore split into non-overlapping alternating phases of interaction with the environment and computation within the design[1]. The components within the design are allowed to interact asynchronously. Two implementations are *synchronously equivalent* if the output traces gathered at the end of the computation phases are the same for the same input traces from the environment. Synchronous equivalence extends the notion of functional equivalence of synchronous circuits in to the domain of embedded systems.

The synchronous assumption and the synchronous equivalence enable us to separately reason about the functionality and the timing of the design. Timing properties of embedded systems, such as worst case execution time, are treated in [4], and are beyond the scope of this paper. One key operation in reasoning about functionality is determining whether two implementations are synchronously equivalent, or whether an implementation is synchronously equivalent to a specification (golden model).

Equivalence of sequential systems is an extremely complex problem in general. It can be done precisely, by searching the state space (e.g. formal verification tools [5, 6]), or conservatively (but more efficiently), through structural algorithms. Efficient structural algorithms exist for the analysis of synchronous equivalence of loop-less implementations [3]. Unfortunately, these algorithms cannot be applied to designs with loops. In this paper, we introduce an efficient synchronous equivalence analysis algorithm that is able to deal with some designs with loops, and is less conservative than previously known algorithms. We also extend the reach

*

[1] Even for systems that require more computation time to process some events than is allowed between the arrival of other, more frequent, events, it is often possible to define a "quantum" of the slower computation that can be executed within the time frame allowed by the more frequent events.

Permission to make digital/hardcopy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2000, Los Angeles, California
(c) 2000 ACM 1-58113-188-7/00/0006..$5.00

of analysis to deal with implementations based on multiprocessor architectures.

In the next section we briefly review a formal model for control dominated embedded system design, CFSMs, that provides a convenient representation of the design space. In section 3, we present the abstract communication analysis based on the monotone abstraction, that can deal with some systems with loops. In section 4, we show how synchronous equivalence analysis techniques can be applied to heterogeneous architectures. In section 5, we demonstrate our method with a case study of an industrial design. We conclude by giving some future directions in section 6.

2 Network of CFSMs

Embedded systems can be specified as networks of Co-Design Finite State Machines [7]. A CFSM network is characterized by globally asynchronous communication among the extended FSM components and locally synchronous execution within extended FSM components. A CFSM network specification consists of the structure of the design (i.e. I/O connectivity of the components) and the functions of the components (i.e. transition and output functions of individual CFSMs).

A CFSM network specification by itself is non-deterministic. More than one behavior is possible and many implementations can be consistent with a CFSM network specification. The designer implements the CFSM network by assigning components to computational resources and specifying a scheduling for components that can be enabled at the same time. This process is called *architectural mapping* of the CFSM network. An architectural mapping makes the CFSM network more deterministic by additionally specifying:

- Delays of component executions.
- Scheduling policy, i.e. the conditions for a component to be enabled for execution, as well as the conditions for an enabled component to actually be executed. The assignment of components to computational resources such as hardware, software, or a given processor in a multiprocessor environment, can also be seen (as explained better in Section 4) as a choice of scheduling policies.

A CFSM network can be simulated only if it is mapped. We refer to all mapped specifications as implementations. Checking two implementations for equivalence may be used to verify that some manual design optimization did not alter the functional behavior, or it may be used to verify an implementation versus a "golden" (simulation) model.

3 Communication Analysis Based on Monotone Abstraction

The general equivalence checking problem for sequential systems is very complex. The goal is then to find efficient, but conservative heuristics for synchronous equivalence checking of embedded system implementations. The algorithm conservatively decides the synchronous equivalence between two given implementations from the same specification, assuming that both satisfy the synchronous assumption. A separate worst case timing analysis [4] will be needed to verify this assumption.

In the next section, we present some preliminary definitions. In section 3.2, a new analysis algorithm, *Monotone Abstract Communication Analysis*, is then presented. It is less conservative than the previous algorithm and can be applied to some systems with loops.

3.1 Preliminaries

During a computation phase, there is no interaction between the design and its environment. Within the design, however, components receive events, perform executions in some order, and send out events which can trigger other executions. During a computation phase, any given component may be executed many times.

Definition 1 (Execution Trace) *The sequence of input event patterns consumed by a (CFSM) component is the* **local execution trace** *of the component. Given an input trace, the* **execution trace** *of the implementation is obtained by grouping together all local execution traces of all components and concatenating them across computation phases.*

The execution trace has the following important property [3].

Lemma 1 *For every legal input trace, if the execution traces from two implementations are identical, then the two implementations are synchronously equivalent to each other.*

Definition 2 (Scheduling Policy) *A scheduling policy for a CFSM network can be defined by two functions:* **Enabled** *which chooses a subset of CFSMs to be considered for execution, and* **Select** *which chooses a subset of the enabled tasks to be executed.*

A component is usually considered enabled for execution if one or more of its inputs have arrived. However, a scheduling policy may enable a component even when no input events are present. In this case, the execution should consume no input, produce no output and leave the component in the same state. This is known as *empty execution.*

Definition 3 (Delay Insensitive Scheduling Policy) *A scheduling policy for a given CFSM network is* **delay insensitive** *if different delay assignments to the (CFSM) component executions will result in implementations that are guaranteed to be synchronously equivalent to each other.*

It has been shown in [3] that many common scheduling policies, such as static priority serial (e.g. static priority scheduler on a single processor), cyclic executive serial (e.g. cyclic executive scheduler on a single processor), and unit delay parallel (e.g. synchronous circuit), are delay insensitive. A delay insensitive scheduling policy has the advantage that the designer is free to optimize individual components and the resulting implementation is guaranteed to be synchronously equivalent to the original one. Furthermore, design with a delay-insensitive scheduling policy is functionally robust with respect to uncertainty in timing delays, such as those that may be caused by cache misses. Functional simulation of an implementation with a delay insensitive scheduling policy can be done without costly and often inaccurate timing estimation.

Definition 4 (Well-Behaved Scheduling Policy) *A scheduling policy is* **well-behaved** *if:*

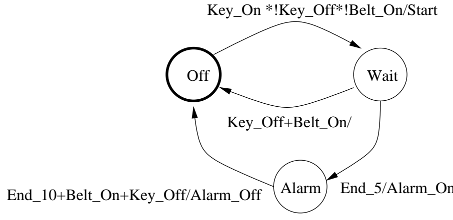

Figure 1: Controller FSM in Seat-belt Example

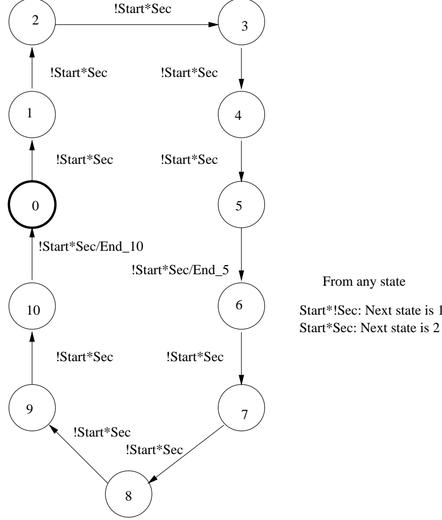

Figure 2: Timer FSM in Seat-belt Example

1. *it is delay-insensitive,*
2. **Enabled** *contains at least all CFSMs with some events on its inputs,*
3. *if* **Enabled** *is replaced with an alternative that enables additional CFSMs then the execution trace remains the same, except possibly for the insertion of some empty executions.*

Whether or not a given scheduling policy is well-behaved needs to be established separately, possibly with the use of formal verification tools. This will only need to be done once since the property is not design dependent. Many common scheduling policies, including static priority serial (SPS), cyclic executive serial (CES), and unit-delay parallel (UDP), are all examples of well-behaved scheduling policies. An example of a policy that is not well-behaved is the following (quite un-natural) *Select* rule: "If CFSMs A, B and C are enabled, execute B followed by C, followed by A. If only B and C are enabled then execute C followed by B."

Consider the seat-belt example consisting of two CFSMs: a controller and a timer. CFSMs representing the controller and the timer are shown in Figures 1 and 2, respectively. Input and output for a state transition are separated by "/". Conjunction is represented by "*", disjunction by "+", and negation by "!". The design roughly performs the function: "Five seconds after the key is turned on, if the seat belt has not been fastened, the alarm will be on for at most five seconds."

Definition 5 (System Graph) *The* **system graph** *(actually, a multi-graph) for a CFSM network is obtained by representing each CFSM, primary input, and primary output by a node. A directed edge is then created for each signal from a sender CFSM or primary input to a receiver CFSM or primary output.*

3.2 Monotone Abstract Communication Analysis

While two implementations with the same delay-insensitive scheduling policy are guaranteed to be synchronously equivalent, two implementations with different delay-insensitive scheduling policies may still be synchronously equivalent to each other, depending on the behavior of the design. Since the corresponding components in the two implementations are guaranteed to have the same functionality, and the connectivity among the components is also guaranteed to be the same, it is possible to deduce the equivalence of two implementations from the behavior of the communication. To this end, we look for a *communication signature* in the flavor of worst case analysis in real-time scheduling [8]. If two implementations have the same communication signatures, they should have the same execution traces. From Lemma 1, it also follows that they are synchronously equivalent to each other. The communication signature that we propose is the *monotone abstract maximal execution trace (MAMET)*. The MAMET of a given CFSM network is a directed acyclic graph. MAMET nodes can be thought of as "containers" representing *possible* events, in the flavor of event graphs representing partially ordered histories [9, 5], but using the additional notion of possibility to further abstract it. Thus, a container can contain either a "0" (event absence) or a "1" (event presence). Each node is labeled with a signal that it corresponds to. The MAMET construction procedure also labels each container with a level, but this label is discarded after the construction is completed. Roughly speaking, edges in the MAMET represent dependencies between input and output events, as well as the ordering among events belonging to the same signal.

A MAMET for an implementation with a given scheduling policy can be obtained by "simulating" (or "unfolding") the system graph as follows:

1. Create a container for each primary input and label it with level 0. Set the current level to 0.
2. Determine the set of *active* inputs for each (CFSM) node. A CFSM input is active if there exists a corresponding container with a level that is larger than that used by a previous execution level of that CFSM (i.e. there exists a container that has not been "consumed" by the previous execution).
3. Let all CFSMs with at least one active input be enabled. If no CFSM is enabled, than STOP.
4. Apply the *Select* function of the original scheduling policy to choose the CFSM(s) to be "executed".
5. *Abstractly execute* the selected CFSM by increasing the current level by 1 and repeating the following for each CFSM output: If that output can be emitted in some state, for some input assignment that assigns 1 or 0 to the active inputs and 0 to the other inputs, then:
   - create a new container and label it with the current level and the name of that output,

- for every active input: create an edge from the most recent container corresponding to that input, to the newly created container,
- create an edge from the previous container labeled with the same name (if any exists) to the new container.

6. Go to step 2.

The MAMET generation procedure may not terminate, even if the CFSM network stabilizes for every input pattern[2]. However, if the procedure terminates, then MAMET can be used to decide synchronous equivalence. The abstract nature of MAMET gives it efficiency, but also makes it suffer from the false-negative problem. In section 5 we will show the usefulness of MAMET on an industrial design.

In practice, the abstract execution in step 5 is performed by evaluating the *least non-decreasing cover* of the CFSM output function. To obtain this cover function, we first existentially quantify state variables from the output function, to obtain Boolean function F that depends only on input Boolean variables, say $x_1, \dots x_n$. Next, we compute:

$$\begin{aligned} F^0 &= F \\ F^i &= F^{i-1} + F^{i-1}_{\overline{x_i}} \qquad \text{for } i = 1, \dots, n \ . \end{aligned}$$

It is not hard to see that F^n is non-decreasing (changing some input from 1 to 0 cannot cause the function to change from 0 to 1), it covers F (it is 1 for every minterm of F), and it is the least such function, in the sense that taking away any added minterm would make it decreasing. To abstractly execute a CFSM, we evaluate its least non-decreasing cover with all active inputs set to 1 and all others set to 0.

The output function of the CFSMs in the seat-belt example, after existentially quantifying out state variables, is:

```
Controller: Alarm_On = !Key_Off*!belt_On*End_5
            Alarm_Off = Key_Off+Belt_On+End_10
            Start = Key_On*!Key_Off*!Belt_On
Timer: End_5 = !Start*sec
       End_10 = !start*sec
```

The least non-decreasing cover for this function is:

```
Controller: Alarm_On = End_5
            Alarm_Off = Key_Off+Belt_On+End_10
            Start = Key_On
Timer: End_5 = Sec
       End_10 = Sec
```

The MAMET for the seat-belt example is shown in figure 3 for several different scheduling policies. The ">" symbol shows the priority order in static priority serial scheduling. The component with higher priority will be chosen to be executed if both components are enabled. The "," symbol shows the list ordering for cyclic executive serial scheduling. At every computation cycle, the scheduler always chooses the next component from the list to be executed, if it is enabled. At the end of the list, the scheduler returns to the top. For the sake of readability, MAMET for $SPS : T > C$ (Timer at higher priority than controller) has been compressed so that there is an edge from all (5) inputs to all (3) outputs.

We can (conservatively) check two implementations for synchronous equivalence by comparing their MAMETs, as stated by the following result.

[2]This non-termination is due to looping through the same patterns, and thus can be easily identified in the algorithm implementation, as discussed below.

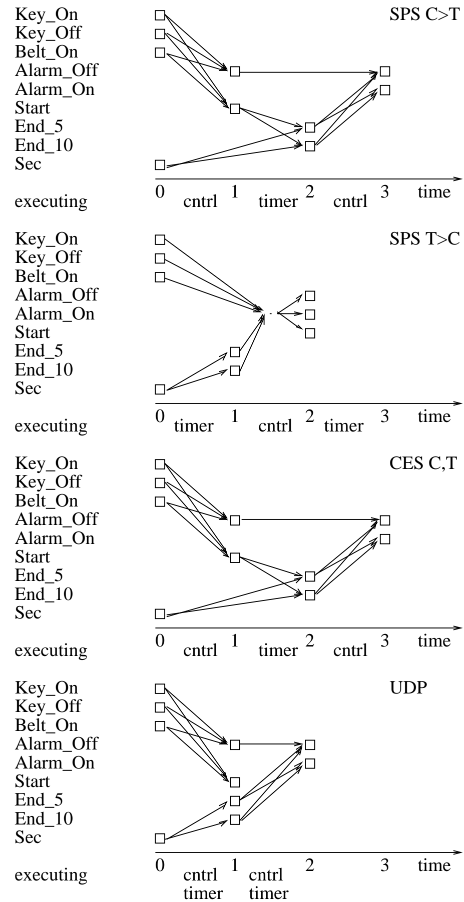

Figure 3: MAMET for Seat-belt Example

Theorem 1 *If two implementations of a given CFSM network with well-behaved scheduling policies have identical MAMETs, then they are synchronously equivalent.*

We prove the theorem in several steps. Given two policies P and Q, we first construct auxiliary policies P^{MAMET} and Q^{MAMET}. The policy P^{MAMET} (Q^{MAMET}) has the same *Select* function as P (Q), but uses a different *Enabled* function, based on the MAMET of P (Q respectively). Then, we prove that P and P^{MAMET} are synchronously equivalent (the same proof applies to equivalence of Q and Q^{MAMET}). Finally, we show that P^{MAMET} and Q^{MAMET} are equivalent, and the desired result follows by transitivity.

Lemma 2 *Consider two scheduling policies for an arbitrary CFSM network. If the two policies are well-behaved and have the same Select function, then they are synchronously equivalent.*

Proof: By definition, the execution traces generated by the two policies will be the same, except for empty executions. Therefore, the value of the last event emitted on some output will be the same for both policies. □

For some well-behaved scheduling policy P we construct a policy P^{MAMET}, by changing the *Enabled* function. The rules for P^{MAMET} follow the outline of the MAMET-generating procedure, except that in step 5, in addition to

abstract execution, the actual selected CFSM is executed with actual inputs. Note that P and P^{MAMET} are synchronously equivalent by Lemma 2.

Lemma 3 *The following holds:*

- *when a CFSM is executed in step 5 of the MAMET algorithm, any input with value 1 is also an active input (as determined in step 2),*
- *if a CFSM emits an output in step 5, then a corresponding container is also created by abstract execution.*

Proof: By induction, using the fact that the least non-decreasing cover evaluates to 1 whenever the actual output function does, even if some of its inputs are changed from 1 to 0. □

In other words P^{MAMET} can be seen as determining the values of the containers in the MAMET, 1 if the event is present and 0 if it is absent. Lemma 3 ensures that for every event that is actually generated, there exists a container to put it in.

Corollary 1 *The content of a container determined by P^{MAMET} depends only on the contents of its immediate predecessors in the MAMET, and the CFSM output function.*

Proof: By definition, all active inputs are immediate predecessors, and by Lemma 3, all other inputs are 0. □

The following result states that the MAMET contains sufficient information to decide synchronous equivalence.

Lemma 4 *If two scheduling policies P and Q have identical MAMETs and if P^{MAMET} and Q^{MAMET} assign to corresponding nodes the same contents for every primary input assignment, them P and Q are synchronously equivalent.*

Proof: We first show that P^{MAMET} and Q^{MAMET} are synchronously equivalent. Indeed, by definition of MAMET, all containers corresponding to some primary output form a chain, and the last container in the chain has the highest level (which may be different for P^{MAMET} and Q^{MAMET}). By assumption, the contents of the last container is the same for P^{MAMET} and Q^{MAMET}, implying that P^{MAMET} and Q^{MAMET} are synchronously equivalent. It follows then by Lemma 2 that P and Q are also synchronously equivalent. □

Now we have all the pieces to prove the theorem.

Proof of Theorem 1: By Lemma 4 we only need to show that given two different well-behaved policies P and Q, their modifications P^{MAMET} and Q^{MAMET} assign the same values to corresponding containers. This is shown by induction, using Corollary 1, and the fact that the CFSM functions are the same in the two implementations. □

From figure 3, implementations $SPS : C > T$ and $CES : C, T$ have identical MAMETs and are therefore synchronously equivalent. The MAMET analysis result is obtained with negligible computational resources. The synchronous equivalence result has also been independently verified using formal verification tools [6], using around 10 seconds of CPU time. The abstract communication analysis with OR abstraction presented in [3] is related to monotone abstract communication analysis by having a different abstraction function, namely, an OR abstraction. Our proof can actually be adapted for the OR abstraction but there will be many more empty executions in the maximal execution trace, that make the OR abstraction more conservative in general. For loop-less systems, both analysis techniques have a complexity quadratic to the number of components in the system. OR abstraction maximal execution trace (OAMET) will always be infinite for system with loops. Applying OAMET analysis to the example of figure **??** results in the following abstract execution: primary inputs first cause Controller to emit a container at "Start", which causes the Timer to execute and emit "End_5" and "End_10", which in turn cause the Controller to execute and emit "Start" again. The OAMET is infinite and the analysis result inconclusive. MAMET analysis, on the other hand, abstracts away less information so the controller does not emit "Start" when it receives only "End_5" and "End_10" at its input. This is how MAMET analysis can generate finite MAMETs for some systems with loops. Even though it is possible to construct systems with infinite MAMETs, similar to those of infinite OAMET for the seatbelt example, we claim, and later illustrate with the case study in section 5, that MAMET will be effective handling start-end or request-acknowledge loops similar to the one in the seatbelt example.

By storing the presence/absence pattern of signals at each invocation of *Select* function of the scheduling policy, the MAMET generation procedure can detect when the abstract execution of an implementation is caught in an infinite loop. Practically, we generate MAMETs for the two implementations in parallel. The analysis returns inconclusive result if MAMETs differs at any point, or if either of the implementation is caught in an infinite loop. Otherwise, the two implementations are synchronously equivalent.

4 Heterogeneous Architecture Analysis

One of the salient characteristics of embedded systems is that they are often implemented on heterogeneous architectures. Different parts of the system are often better suited for mapping to computation resources with very different performance and cost factors. In this section we consider two very common heterogeneous architectures, the co-processor architecture and the synchronous parallel architecture, and show how they can be modeled for efficient synchronous equivalence analysis.

In a co-processor architecture, the microprocessor acts as a master of the communication to and from the co-processors. Whenever there is some computation to be done on the co-processors, the microprocessor emits the events and associated data to the co-processors, and waits for the operation on the co-processors to be completed. An acknowledge event is sent back to the master, before continuing with its own operation. The operations of the master and the co-processors thus become serialized. A co-processor architecture can be modeled, for the purpose of abstract communication analysis, as a static priority serial architecture with all the components mapped to the co-processors having higher priority than the ones that are mapped to the master.

The synchronous parallel architecture is characterized by processors executing in parallel, but communication is allowed only at a time when all the processing on each processor has been completed. An example is shown in Figure **??**. For the purpose of abstract communication analysis, a synchronous parallel architecture can be hierarchically modeled as having a unit delay parallel scheduling policy on top of the original single processor scheduling policies.

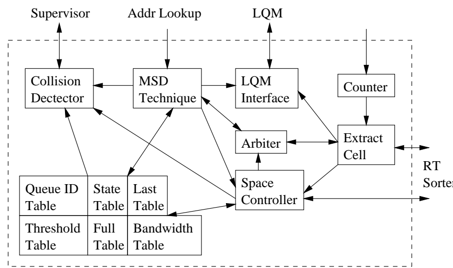

Figure 4: Algorithm block of a ATM server

5 Case Study: An ATM Switch

We applied our monotone abstract communication analysis to a real-life industrial design: The algorithm block of a server that supports ATM-based Virtual Private Networks. The complete server [10, 11] required a design effort of approximately 3 man-years. The algorithm block was re-specified as 1200+ lines of Esterel code separated into 13 different CFSMs. If we were to represent even just the control portion of the system (excluding tables) as a Boolean network, it would have required more than 500 binary latches. Without extensive manual abstraction, verifying this design is clearly beyond the capability of existing formal verification tools [6].

The algorithm block decides which input cells must be accepted or discarded to avoid node congestion, and implements the shaping and bandwidth partition functions among ATM VPCs. Figure 4 provides a functional description of the algorithm block. Upon arrival of a new cell, it receives the Cell ID from the address-lookup module. If the cell is accepted, the Message Selective Discarding Technique sends instructions to the Logic Queue Manager about where (i.e. in which queue) to store the cell in the shared buffer. Communication between the algorithm block and the LQM is handled by the LQM interface, which performs the required protocol adaptations.

We used abstract communication analysis to decide synchronous equivalence among many different implementations, including ones with UDP and various SPS and CES scheduling policies. The MAMETs were generated using Polis [7] on a network where CFSMs were replaced with their least non-decreasing covers. With minimum additional effort, we carried out the same experiment with Esterel [2] and obtained the exact same MAMETs. In all cases, the generation of the MAMET took less than 1 second of CPU time. The MAMET associated with the SPS policy chosen by an expert designer consists of 314 containers. Even with such a complex MAMET, we have found a CES scheduling policy that is synchronously equivalent to the designer-specified SPS, therefore resulting in an implementation with less scheduling overhead.

We have performed abstract communication analysis for several different SPS, CES, and UDP scheduling policies. In each case, the MAMET is finite even though there are many loops in this heavily interacting design. The analysis was performed in a very short computation time and with negligible memory occupation.

6 Conclusion and Future Work

In this paper we have extended and improved synchronous equivalence analysis to cover system with loops and architectures with multiple computation units. At the same time we made the analysis less conservative. We demonstrate our method on an ATM switch and manage to perform synchronous analysis for many implementations with realistic scheduling policies, even though the system contains components that are heavily interacting.

An important direction for future work is to make the analysis even less abstract and reduce false-negative results by considering partial abstraction on the state. Abstract communication analysis may be seen as a special case of symbolic simulation [12], but more work is needed to clearly establish and exploit this relationship.

References

[1] S. H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley Interscience, 1969.

[2] G. Berry, P. Couronné, and G. Gonthier. The synchronous approach to reactive and real-time systems. *IEEE Proceedings*, 79, September 1991.

[3] H. Hsieh, F. Balarin, L. Lavagno, and A. Sangiovanni-Vincentelli. Synchronous equivalence for embedded systems: A tool for design exploration. In *Proceedings of the International Conference on Computer-Aided Design*, November 1999.

[4] F. Balarin. Worst-case analysis of discrete systems. In *Proceedings of the International Conference on Computer-Aided Design*, November 1999.

[5] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[6] R.K. Brayton, et.al. VIS: A system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of Computer Aided Verification: 8th International Conference, CAV'96, Rutgers, NJ, July, 1996.* Springer-Verlag, 1996. LNCS vol. 1102.

[7] F. Balarin, et.al. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach.* Kluwer Academic Publishers, 1997.

[8] C.L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46 – 61, January 1973.

[9] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains. part I. *Theoretical Computer Science*, 13:85–108, 1981.

[10] Coppo, M. D'Ambrosio, and V. Vercellone. The A-VPN server, a solution for atm virtual private networks. *Proceedings of ICCS*, November 1994.

[11] E. Filippi, L. Lavagno, L. Licciardi, A. Montanaro, M. Paolini, R. Passerone, M. Sgroi, and A. Sangiovanni-Vincentelli. Intellectual property re-use in embedded system co-design: an industrial case study. *International Symposium on System Synthesis*, December 1998.

[12] C. Seger and J. Brzozowski. Generalized ternary simulation of sequential circuits. *Informatique Theorique et Applications*, 28(3-4):159–86, 1994.