Copyright © 1999, by the author(s).
All rights reserved.

Permission to make digital or hard copies 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 and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

SYNCHRONOUS EQUIVALENCE FOR EMBEDDED SYSTEMS: A TOOL FOR DESIGN EXPLORATION

by

Harry Hsieh, Felice Balarin and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M99/1

15 January 1999

SYNCHRONOUS EQUIVALENCE FOR EMBEDDED SYSTEMS: A TOOL FOR DESIGN EXPLORATION

by

Harry Hsieh, Felice Balarin and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M99/1

15 January 1999

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Synchronous Equivalence for Embedded Systems: A Tool for Design Exploration

Harry Hsieh, University of California at Berkeley*
Felice Balarin, Cadence Berkeley Laboratories
Alberto Sangiovanni-Vincentelli, University of California at Berkeley

Abstract

Design exploration consists of analyzing several alternative implementations of the "same" function to determine the most desirable one. A fundamental question is whether an "implementation" is consistent with the high-level specification or whether two implementations are "equivalent". In this paper, we define synchronous equivalence for embedded systems that strongly resembles the concept of functional equivalence for sequential circuits. We then present equivalence analysis algorithms that are of low polynomial complexity. We show an example of application of the algorithms to a real-life design (a shock absorber controller) and demonstrate that synchronous equivalence opens design exploration avenues uncharted before.

1 Introduction

Current embedded system design practice is quite informal and application specific. Designers often start with an informal requirement written in plain English, use "intuition" to pick a particular interpretation of this requirement, and write a so called reference (or golden) model in VHDL, Verilog, C or any other language that has an operational semantic and can be executed. The golden model is executed on a computer to investigate whether it satisfies a set of requirements including a match with the original informal specification. A (candidate) implementation[1] is then generated through a combination of manual labor and often poorly connected tools. The correctness and optimality of the (candidate) implementations are assessed with filtered simulation traces obtained from the reference model and from the candidate implementation. This contorted and highly informal design flow is very error-prone and does not promote efficient design space exploration since the set of "correct" implementations cannot be precisely identified.

*This author is supported by SRC contract DC-324-028

[1] An "implementation" may only be considered a candidate because it may not be correct. The implementation in this context is not generated through formal refinement. Some *ad hoc* manual procedures are involved.

A fundamental point of clarification to improve the design methodology is the formal definition of correctness. We advocate the principle of "separation of concerns" in verification, that is, we would like to verify functional correctness and timing independently. This principle is the basis of the synchronous design methodology for sequential circuits, where latches decompose the circuit into combinational islands. Signal are propagated from island to island when an enabling input is given to the latches. The enabling input is periodic where the period is generated by a "clock". Any design of the combinational islands ensuring that the combinational circuits stabilize before the enabling signal arrives at the latches, can be verified for equivalence paying attention only to the Boolean functions computed by the circuits irrespective of the propagation time. Timing can then be verified independently by performing a worst-case timing analysis and making sure that this bound is within the clock cycle. This powerful approach can be extended to higher level of abstraction as demonstrated by synchronous languages [3]. Synchronous languages describe complex systems consisting of interconnected modules each represented by a Finite-State Machine model. Communication among modules is synchronous. Both communication and computation take zero time to perform. While very powerful, synchronous languages support a model of computation that restricts the design space considerably because of the synchronous communication hypothesis.

In this paper, we would like to relax the "synchronous hypothesis" by adopting a more general model of computation (the one supported by Co-design Finite State Machines(CFSM) [1]), while retaining the fundamental idea of separation between timing and functionality. We will establish a "functional" equivalence among a set of candidate implementations of embedded system specifications. Just as sequential circuit methodology abstracts away different gate delays among different implementations and enable speed/area trade-off among functionally equivalent implementations, we will abstract away the delays of embedded system computational resources. We call this equivalence relation *Synchronous Equivalence.*

While it is possible that some other equivalence relation identifies a larger design space of functionally equivalent circuits, *Synchronous Equivalence* lends itself nicely to simple analysis procedures. We want to be able to figure out quickly whether two implementations are synchronous equivalent to each other. Synchronous equivalence relation divides the design space into synchronous equivalence classes. Within an equivalence class, different implementations represent different speed/cost trade-off. Equivalence analysis can be done

precisely through reachable state methods, or conservatively (but more efficiently) through structural method. We will derive efficient structural algorithms that can be used to explore the design space effectively.

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 synchronous equivalence relation and associated definitions. In section 4, we show how synchronous equivalence can be checked by structural methods. In section 5, we show some results of applying this methodology to a real-life industrial example. In section 6, we discuss future directions.

2 Design Representation: Network of CFSMs

Embedded systems can be represented as networks of interacting Codesign Finite State Machines [1]. CFSMs are Finite State Machines extended with side-effect-free computation on the transition edge. Like traditional FSM, the transition is considered atomic: once a transition starts, it must eventually carry on to the end before the next transition of the same CFSM can start. The communication entities between CFSMs are events, which may or may not carry values. Events and their values are considered atomic also. A CFSM can transition only when an input event has "occurred".

Individual CFSM operates in a "locally synchronous" fashion with its own clock. There is no *a priori* synchronization between CFSMs and all the local clocks are completely unsynchronized. This feature is necessary because different resources can operate at widely different speeds. In order for such "globally asynchronous" objects to communicate, buffering is needed. We deal only with minimally required one-deep buffer. Any extra buffering can be designed into the CFSMs themselves. The local clock for each CFSM does not have to be periodic. It is purely a signal for the completion of previous transition and hence, the enabling of the current transition. There is no *a priori* relations between the local clocks and physical time.

With this model of computation, the designers can specify their designs with minimal implied implementation attributes. At the "specification" level, designers specifies only the structure of the design (i.e. number of CFSMs and I/O of these CFSMs) and the local functions of the design (i.e. transition and output relation of individual CFSM, along with extended computation at the transition edges). Tools exists to verify the design at this level. Since the representation is highly abstract, the tools are either in the flavor of syntax checker for the structure of the design or formal verification tools for the properties of the design.

Implementing the specification involves allocating individual CFSMs to computation resources and assigning scheduling policy to shared resources. We call this high-level implementation process *architectural mapping*. Architectural mapping has the consequences of refining the relationship between the local clock and the physical time. If the CFSM is to be implemented on a VLSI synchronous hardware, its local clock will become periodic and the clock period will equal to the synchronous hardware clocking. If the CFSM is to be implemented on a processor, the local clock will not have fixed period and will run at some multiple of the processor clock, depending on the execution delay of the transition on that processor, which in turns depend on software synthesis and scheduling. CFSMs on the same processor resource need to be executed in mutual exclusion (possibly with preemptive scheduling) while different processors and dedicated hardware resources can be executed in a concurrent manner.

Since local CFSM clocks are unsynchronized, a network of CFSMs is inherently non-deterministic: for a fixed input sequence, many system responses are possible (and they are all equally valid). However, a mapped network is deterministic: its response is unique for any fixed input sequence. In fact, we extend the notion of architectural mapping to include any set of rules that resolve non-deterministic choices in a CFSM network (making it deterministic). To resolve non-determinism, a mapping needs to specify two things:

- delays for potentially parallel activities: for two activities happening at the same time, we need to know which one will finish first,
- scheduling: if two activities are enabled, we need to know whether they will be executed in a particular order, or perhaps in parallel.

For example, simulating a CFSM network (which necessarily involves resolving non-determinism) is considered an architectural mapping.

For simplicity, we refer to all mapped specification as implementations (thus a mapping to a simulator is also considered an implementation). Therefore, checking two implementations for equivalence may be used to verify that some manual design optimizations did not alter the behavior, or it may be used to verify a physical implementation versus the "golden" (simulation) model.

3 Synchronous Assumption and Synchronous Equivalence

In this work we are making a key assumption (a restriction on the class of "acceptable" specifications) in order to make efficient analysis possible. We believe that this class includes many interesting examples, and that other specifications may be reduced to it by appropriate (manual for now) partitioning techniques.

Synchronous Assumption The operation of the design is split into two alternating *non-overlapping phases*. An interaction phase where the environment interacts with the design and a computation phase where the design performs computation.

The interaction phase followed by its associated computation phase is called a "cycle". We will only consider specification that satisfies synchronous assumption. The implementation process must guaranteed to preserve it. This can be done by a separate worst case timing analysis of the flavor of [2]. This analysis is to make sure the implementation obeys synchronous assumption also.

Synchronous Equivalence Under the synchronous assumption, two embedded system implementations are synchronous equivalent if and only if for all possible input traces the outputs of the implementations are the same at the end of every cycle.

As long as the results (outputs of the network of CFSMs) are the same at the end of the cycle, order of execution of single CFSMs or even the parallel/serial nature of the computations do not matter. The former can lead to freedom in scheduler selections while the latter can lead to freedom in processor allocation. Note that this notion of synchronicity is similar to the fundamental mode of operation of asynchronous circuits.

3.1 Related Work

Synchronous languages are a group of languages proposed for automatic synthesis of embedded software [3]. Synchronous languages have a unique notion of "synchronous scheduler", the scheduler that defines correct behavior. This scheduler is the result of the assumption of synchronous communication among modules of the design. Our synchronous assumption is related to the "external" communication of the design with the environment. Hence, there is an intrinsic non-determinism in our specification that results in many possible implementations that are functionally equivalent to the specification.

Another methodology that utilizes synchronous assumption is synchronous data-flow: a powerful formalism for data-dominated embedded systems geared toward simulation and code synthesis for digital signal processors [5]. It exploits the synchronous assumption at the interface between the network and the environoment, while "blocking read" is required of all components in the design so the behaviors are the same (in Kahn's sense) independent of allocation and scheduling.

In our framework, systems described by synchronous languages can be seen as networks of CFSMs with a particular architectural mapping: the one defined by the synchronous scheduler. Similarly, synchronous data-flow systems can be seen as networks of CFSMs with an architectural mapping that respects the blocking read property. While our method can certainly be applied to check equivalence between these and some other architectural mappings, we do not restrict ourselves to them. We use equivalence analysis to tell us whether *any* two architectural mappings are equivalent to each other. Trade-off on timing and cost can then be performed on the "functionally" equivalent implementations. In the next section, we show how synchronous equivalence analysis can be performed efficiently.

4 Analysis of Synchronous Equivalence

The general equivalence checking of sequential systems is very complex. We will devise powerful but conservative heuristics for synchronous equivalence that are of low polynomial time complexity. Our algorithms will decide the synchronous equivalence between two given implementations (network of CFSM plus architectural mapping) from the same specification, assuming both of them satisfy the synchronous assumption. A separate timing analysis will be needed, but we do not address that issue here.

We will first show that for some subsets of implementations, synchronous equivalence will hold regardless of the functions that is being implemented. This type of analysis is "functionality independent". The complexity here is constant. If two implementations do not belong to one such subset, a more complex analysis need to be performed in order to determine equivalence. We introduce "abstract communication analysis" which can be applied to a large set of implementations.

4.1 Functionality-Independent Synchronous Equivalence

The goal here is to identify subsets of implementations such that if two implementations fall within one subset, they are guaranteed to be synchronous equivalent. It is conservative because even if two implementations do not fall within the same subset as defined by these "algorithms", they may still be synchronous equivalent.

Global State Pattern A complete characterization of the state the implementation is in. It will includes state information of all the componenets and values on all the buffers, counters, and any other memory element.

Stabilization An implementation is stabilized if and only if no change in global state pattern or output is possible without the application of a primary input. A system that satisfies the synchronous assumption stabilizes at the end of a cycle.

A single primary input pattern can stimulate the design and "generate" a sequence of global state patterns until stabilization is reached. A sequence of primary input patterns (i.e. a primary input trace) "generates" a sequence of sequence of global state patterns. A primary input trace also generate a sequence of sequence of scheduling points.

Scheduling Point A scheduling point is a point in time where some process finishes computation, or produces some output. It is the point in time some "scheduling decision" need to be made. There are often many scheduling points within a single computation phase.

Scheduling points are related to the behavior of the internal modules of the design. When the implementation is stabilized, the computation phase ends. The end of the computation phase corresponds to a scheduling point.

Lemma 1 Given two implementations, A and B, of the same specification, and an arbitrary input trace $i = \{i_1, i_2, ...\}$, i generates a sequence of sequences of scheduling points $\{\{a_1^1, a_1^2, ...\}, \{a_2^1, a_2^2, ...\}, ...\}$ for implementation A, and $\{\{b_1^1, b_1^2, ...\}, \{b_2^1, b_2^2, ...\}, ...\}$ for implementation B.
Let the global state pattern be $\{\{P_1^1, P_1^2, ...\}, \{P_2^1, P_2^2, ...\}, ...\}$ for implementation A, and $\{\{Q_1^1, Q_1^2, ...\}, \{Q_2^1, Q_2^2, ...\}, ...\}$ for implementation B at the scheduling points. If $P_n^m = Q_n^m$, for all integer m,n, A and B are synchronous equivalent.

Proof of Lemma 1 By definition, the implementation stabilizes at some scheduling point. Since $P_n^m = Q_n^m$ for all scheduling points, they must also be the same at all stabilizing points. In addition, an output can only occur at a scheduling point. Since $P_n^m = Q_n^m$, output patterns are the same between stabilizing points also. Therefore, outputs are the same at the end of the cycle (stabilization). Therefore, A and B are synchronous equivalent.

Single processor implementations with the same scheduling decision and preemption policy are synchronous equivalent.

The following two theorems state that having the same scheduling policy is sufficient for synchronous equivalence of two single processor implementations".

Theorem 1 Any two single processor implementations with the same non-preemptive scheduling policy are synchronous equivalent.

Proof of Theorem 1 Let any two single-processor implementations of the same specification with the same non-preemptive scheduling, be A and B. Given an arbitrary input trace, at start up, $P_0^0 = Q_0^0$ because they are specified by the initial state, initial output and the environment. We can now proceed by induction.

- Base Case
  $P_0^0 = Q_0^0$
- Induction Hypothesis
  $P_0^i = Q_0^i$
- Prove: $P_0^{i+1} = Q_0^{i+1}$
  Because $P_0^i = Q_0^i$, the same software scheduler makes the same execution decision and execute the same component, calculate outputs and next state of that component corresponding to the next scheduling point $i+1$. Since the output and transition relation are identical for A and B, $P_0^{i+1} = Q_0^{i+1}$

At stabilization point j, $P_j = Q_j$. The next scheduling point P_{j+1}^0 and Q_{j+1}^0 has the same pattern as the previous scheduling point for all signals except a primary input that has to be identical for both implementations. Therefore $P_{j+1}^0 = Q_{j+1}^0$.

Therefore, $P_n^m = Q_n^m$ for any integer m and n. Due to Lemma 1, all single processor implementations with the same non-preemptive scheduling are synchronous equivalent.

The same theorem can be easily extended to preemptive scheduling.

Theorem 2 Any two single processor implementations with the same preemptive scheduling policy are synchronous equivalent, as long as multiple outputs on the same transition are always emitted in the same order.

Proof of Theorem 2 Follows that of Theorem 1.

Theorem 1 and 2 indicate that once a scheduler is chosen, the designer is free to optimize the individual processes and the resulting implementation will still be synchronous equivalent to the original one. Also, implementation with different processors (hence different delay characteristics) will result in synchronous equivalent implementations as long as the scheduler remains unchanged.

It is clear that, depending on the design specification, two implementations with different scheduling can still be synchronous equivalent. In the next section we try to extend the reach of the analysis by further examining the structure of the system.

4.2 Abstract Communication Analysis

During a computation phase, there is no interaction between the design and its environment. However, components of the design do communicate events among themselves. For a given component during a computation phase, its sequence of executions consume a sequence of events at its input. This is the *local execution trace* of that component. Grouping all the local execution traces together we have the execution trace for that computation phase. More formally:

Execution Trace Given some primary input trace, an execution trace is an ordered list of inputs for the execution of each component of a particular implementation.

Examples of execution trace for the design of figure 1 are shown in table 1. Three implementations are being considered, a single processor with A at a higher priority than B, a concurrent hardware implementation where A and B execute in parallel, and a single processor with B at a higher priority than A. A "1" in the table indicates the presence of an event and a "0" indicates the absence of an event. Execution traces have the following property:

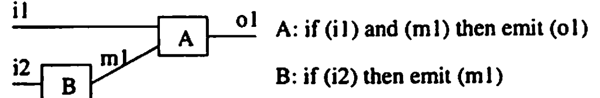

Figure 1: Example for Abstract Communication Analysis

	B<A				B=A				B>A			
	E.T.		M.E.T.		E.T.		M.E.T.		E.T.		M.E.T.	
$i_1 i_2$	B	A	B	A	B	A	B	A	B	A	B	A
11	1	10 01	1	10 01	1	10 01	1	10 01	1	11	1	11
10		10				10				11		
01	1	01			1	01			1	11		

Table 1: Execution Traces and Maximal Execution Traces

Lemma 2 If execution traces from two implementations are identical for all possible input traces, then the two implementation are synchronous equivalent.

Proof of Lemma 2 Since the ordered list of input are the same and the outputs of the components and changes in global state patterns must be the result of the executions of the components, they have to be the same also. If output and global state patterns are the same for all possible input traces, they have to be the same at all stabilization points. The two implementations are therefore synchronous equivalent.

This lemma suggests a simple algorithm for checking synchronous equivalence which is essentially exhaustive simulation. Exhaustive simulation is clearly not practical for all but the most trivial designs. Instead, we generate a "signature" that summarizes the execution traces. A "signature" must have the property that, if two implementations have the same signature, they must be synchronous equivalent. For implementations satisfying the proerty called "order-monotonicity", we can easily obtain one such signature called "maximal execution trace". It is maximal because if an event is possible at some input at some point in time due to some particular execution trace, "event" will be present in the maximal execution trace at that input at that point in time.

Order Monotonicity Given any set of enabled components (i.e., of components that have an input to process), at any scheduling point, the addition of another event can only enable components receiving this event and cannot cause any component to change their priority order.

Single processors with list scheduling or static priority scheduling and synchronous hardware are all order-monotonic. Some dynamic priority scheduling may not be order monotonic (e.g. the priority between A and B depends on whether or not C is enabled). The following simple algorithm computes the maximal execution trace for order monotonic implementations.

Maximal Execution Trace The following procedure is used to obtain maximal execution traces for order monotonic implementations:

1. Existentially quantify both output and transition relation of all components. The functions of the components are effectively replaced by "OR" gates where any input event can cause all output events to be emitted.
2. Simulate with a single pattern of all-primary-input-event-presence to obtain the maximal execution trace.

The correctness of the algorithm hinges on the previous lemma and the following theorem:

Theorem 3 If the maximal execution traces are the same for two order monotonic implementations, their execution traces are identical for all possible input traces.

Outline of Proof of Theorem 3 Due to the order monotonic property, the maximal execution traces can be thought of as actual traces padded with "dummy events" which can not affect the "real" execution. In fact, it can be shown that dummy events only add spurious traces to go on top of the real traces. Therefore, if maximal execution traces are the same, the real execution traces for both implementations have to be identical also.

Table 1 shows all possible execution traces and maximal execution traces. Since the maximal execution traces are the same between implementations A<B and A=B, the two implementations must be synchronous equivalent. If the maximal execution traces are different, as it is between A>B and A=B or A<B and A=B, we cannot conclude that the implementations are not synchronous equivalent.

The complexity of this algorithm is quadratic in the number of components. In fact, each component can be executed no more than n times, where n is the number of components in the design. This algorithm can only be applied to a design with no loops in the connection among processes. We suggest a simple extension to deal with common loop structures in section 6.

5 Experiments

We applied abstract communication analysis to a real-life industrial design: a shock absorber controller [4]. The controller sets the shock absorbers' motors to appropriate absorption levels according to inputs from steering wheel, vertical acceleration sensor, speed sensor, and battery voltage sensor. The system includes over 200 binary latches. The system includes over 200 binary latches, and it is thus not amenable to formal verification without extensive manual abstraction. A conservative analysis was performed making use of the synchronous equivalence checking algorithm in a very short computing time and with negligible memory occupation.

The system graph for this design is shown in figure 2. We use abstract communication analysis algorithm to decide synchronous equivalence among the following five implementations:

1. Synchronous hardware.
2. Single processor with list scheduling: a,b,c,d,e,f,g,h.
3. Single processor with list scheduling: h,g,f,e,d,c,b,a.
4. Single processor with priority: a>b>c>d>e>f>g>h.

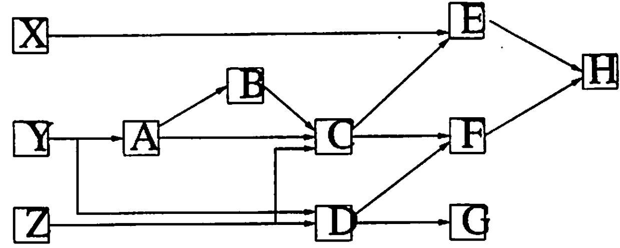

Figure 2: System Graph for Shock Absorber Controller.

impl	execution	a	b	c	d	e	f	g	h
1	1	1	1	001	1	10	11	1	10
	2			010		01	10		10
	3			100		01	10		11
	4					01			11
2	1	1	1	111	1	11	11	1	11
3	1	1	1	001	1	10	11	1	10
	2			010		01	10		10
	3			100		01	10		11
	4					01			11
4	1	1	1	111	1	11	11	1	11
5	1	1	1	001	1	10	01	1	10
	2			010		01	10		01
	3			100		01	10		01
	4					01	10		10
	5								01
	6								10
	7								01
	8								10

Table 2: Maximal Execution Traces.

5. Single processor with priority: h>g>f>e>d>c>b>a.

We obtained the maximal execution traces for all five implementations in table 2. The input patterns are recorded from the top-most input on the graph to the bottom-most one.

From a quick analysis of these traces, we concluded that implementation 1 and 3 are synchronous equivalent. This means that any synchronous hardware implementation and any single processor implementation (with any delay characteristics) with the given list scheduling are synchronous equivalent. If they both satisfy timing constraints, implementation 3 may have a lower cost. Implementation 1 will probably have better performance in terms of timing. We can also conclude that implementation 2 and 4 are synchronous equivalent. This means that any single processor implementation (with any delay characteristics) with the given list scheduling and any single processor implementation with the given static priority scheduling are synchronous equivalent also.

6 Summary and Future Direction

We have proposed a definition of functional equivalence for embedded systems: synchronous equivalence. We also proposed a simple algorithm for evaluating equivalence that can be applied to a large set of implementations.

The calculation of maximal execution traces can easily be made less conservative. Instead of abstracting away all functionality, one can abstract away only the state information of the components and leave everything else intact.

The generated symbolic trace remains correct. Preliminary study has indicated that this type of extension is very useful in dealing with request-acknowledge loops and other loops of similar nature.

Another important direction is to increase the applicability of the abstract communication analysis to include architectures using more than one computational resources. The key is in making the synchronization explicit among resources. A third direction is to deal with timing issues such as the computation of worst-case execution time. This will complement abstract communication analysis and possibly make the synchronous approach as popular in the embedded system domain as it is in the sequential circuit domain.

References

[1] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. 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 Publishers, 1997.

[2] F. Balarin and A. Sangiovanni-Vincentelli. Schedule validation for embedded reactive real-time systems. In *Proceedings of the Design Automation Conference*, June 1997.

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

[4] M. Chiodo, D. Engels, P. Giusto, H. Hsieh, A. Jurecs ka, L. Lavagno, K. Suzuki, and A. Sangiovanni-Vincentelli. A case study in computer-aided codesign of embedded controllers. *Design Automation for Embedded Systems*, 1(1-2), January 1996.

[5] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *IEEE Proceedings*, September 1987.