

Synchronous Equivalence for Embedded Systems: A Tool for Design Exploration

Harry Hsieh*

Alberto Sangiovanni-Vincentelli

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

{hahsieh,alberto}@eecs.berkeley.edu

Felice Balarin

Luciano Lavagno

Cadence Berkeley Laboratories
Cadence Design Systems

{felice,lavagno}@cadence.com

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 a 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, or C. 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) implementation¹ 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.

A fundamental clarification to improve the design methodology is the formal definition of correctness. We advocate the principle of “separation of concerns” in verification. Functional correctness and timing are verified independently. This principle is the basis of the synchronous design methodology for sequential circuits [1], where latches decompose the circuit into combinational islands. Signals are propagated from island to island when an enabling in-

put (clock) is given to the latches. 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 [2]. Synchronous languages describe complex systems consisting of interconnected components each represented by a Finite-State Machine model. 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 relax the “synchronous hypothesis” by adopting a more general model of computation (the one supported by Co-design Finite State Machines(CFSM) [3]), while retaining the fundamental idea of separation between timing and functionality. We establish *synchronous equivalence*, a “functional” equivalence among a set of candidate implementations of embedded system specifications. Equivalence analysis can be done precisely through reachable state methods (e.g. formal verification tools [4, 5]), or conservatively (but more efficiently) through structural methods. We derive efficient structural algorithms for synchronous equivalence analysis 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. 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 Network of CFSMs

Embedded systems can be represented as networks of interacting Codesign Finite State Machines [3]. CFSMs extend Finite State Machines with side-effect-free computation on the transition edge. The communication entities between CFSMs are events, which may or may not carry values. A CFSM can transition only

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

¹An “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.

when an input event has “occurred”.

Individual CFSMs operate in a “locally synchronous” fashion with its own clock. 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. 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).

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. CFSMs implemented in hardware have local clocks coincide with the hardware clocking. CFSM implemented in software has local clock running at some variable period, depending on the execution delay of a particular transition and scheduling.

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 specifications 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 Equivalence

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 specifications

that satisfies synchronous assumption. The implementation process must guarantee to preserve it. This can be done by a separate worst case timing analysis in the flavor of [6].

Synchronous Equivalence Under the synchronous assumption, two embedded system implementations are synchronously 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, the order of execution of 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.

3.1 Related Work

Synchronous languages are a group of languages proposed for automatic synthesis of embedded software [2]. 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 components of the design. Our synchronous assumption is related only 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 “functional” behaviors that are consistent with the specification.

Synchronous data-flow is a powerful formalism geared toward simulation and code synthesis for digital signal processors [7]. It too exploits the synchronous assumption at the interface between the network and the environment, but “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.

Our work does not restrict the implementation choices to those utilizing synchronous scheduler, nor does it require communications to have the “blocking read” property. We use equivalence analysis to tell us whether *any* two implementations are equivalent to each other. In the next section, we show how synchronous equivalence analysis can be performed efficiently.

4 Analysis of Synchronous Equivalence

The general equivalence checking problem for sequential systems is very complex. We devise powerful but conservative heuristics for synchronous equivalence that are of low polynomial time complexity. Our algorithms decide the synchronous equivalence between two given implementations from the same specification, assuming both of them satisfy the synchronous assumption. A separate worst case timing analysis [6] will be needed.

We first show in the next section that for some subsets of implementations, synchronous equivalence will hold regardless of the design that is being implemented. This type of analysis is “design independent” and has constant complexity. If two implementations do not belong to one such subset, a more complex analysis must be performed in order to determine equivalence. We introduce “abstract communication analysis” which can be applied to a large set of implementations.

4.1 Design-Independent Synchronous Equivalence

We first have the following definitions:

Global State Pattern A complete characterization of the state the implementation is in. It includes state information of all the components 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 generates a sequence of sequence of scheduling points.

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

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

Proof of Lemma 1 Output are the same at all stabilizing point because every stabilizing point is a scheduling point. $P_n^m = Q_n^m$ for all scheduling point. Therefore, A and B are synchronously equivalent.

We now present the main theorem of this section.

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

Proof of Theorem 1 Given two such implementations A and B and an arbitrary input trace, $P_0^0 = Q_0^0$ 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 , $P_0^{i+1} = Q_0^{i+1}$

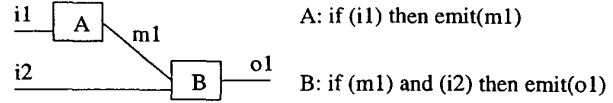


Figure 1: Example for Abstract Communication Analysis

At stabilization point j , $P_j = Q_j$. The next scheduling point $P_{j+1}^0 = Q_{j+1}^0$ because they are the same pattern as at previous scheduling point plus primary input.

Therefore, $P_n^m = Q_n^m$ for any integer m and n . Due to lemma 1, the theorem is proven.

This theorem can be easily extended to preemptive scheduling because preemption always occur at a scheduling point.

Theorem 1 indicates that once a scheduler is chosen, the designer is free to optimize the individual processes and the resulting implementation will still be synchronously equivalent to the original one. Also, implementation with different processors (hence different delay characteristics) will result in synchronously 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 synchronously equivalent. In the next section we try to extend the reach of the analysis by further examining the communication structure of the system.

4.2 Abstract Communication Analysis

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. A given component during a computation phase can be executed many times. The sequence of input event pattern consumed is the *local execution trace* of the component. The *execution trace* of the design is obtained by grouping together all local execution traces of all components and concatenating across phases.

An execution trace does not contain any timing information, except in the form of ordering among the sequential executions of the same component. The ordering of the executions of different components is not kept in the execution trace. This feature allows widely different implementations to be considered “functionally” equivalent.

Execution traces for the design of figure 1 are shown in table 1. Three implementations are being considered: a single processor with component A at a lower priority than component B , a concurrent hardware implementation where A and B execute in parallel with the same delay, and a single processor with A at a higher priority than B . 1/0 in the table indicates the presence/absence of an event. Execution traces can be obtained by simulation. Implementation $A < B$ with primary input $i_1, i_2 = 11$ has B executing first with input $m_1, i_2 = 01$, then A executing with input 1, then B executing again with input 10. The design is memoryless so every computation phase have the same response to the same primary input pattern.

Execution traces have the following important property:

Lemma 2 For every legal input trace, if the execution traces from

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

Table 1: Execution Traces and Maximal Execution Traces

two implementations are identical, then the two implementations are synchronously equivalent to each other.

Proof of Lemma 2 Since all local execution traces (i.e. input traces) are identical, output must be identical at all scheduling points (including stabilizing points). Two implementations are therefore synchronously equivalent to each other.

This lemma suggests a straightforward algorithm for checking synchronous equivalence: simulate all possible input traces and compare the resulting execution traces. Accordingly, implementations A<B and A=B for design in figure 1 are synchronously equivalent to each other. Exhaustive simulation is clearly not practical for all but the most trivial designs. Hence, we introduce *abstract communication analysis*. It is based on the intuition that since two implementations of any design have identical component functionalities and connectivities, the only thing that requires analysis is the communication characteristics. To this end, we look for a “signature” that summarizes the communication.

A good communication signature must be easy to compute, so it can become the inner loop of some automatic design exploration procedure. It must also have the property that if two implementations have the same communication signature, they must be synchronously equivalent. For this purpose we propose the “maximal execution trace” computed by the following algorithm.

Maximal Execution Trace Computation Procedure

1. Replace the functions of the components by “OR”s, so that any input event can cause all output events to be emitted. This is equivalent to existentially quantifying both output and transition relation of all components.
2. Simulate the transformed design with a single primary input pattern of all 1’s (presence). The execution trace from this single simulation run is the maximal execution trace. This is similar to worst case analysis in real-time scheduling [8].

Maximal execution trace is a useful communication signature for implementations whose *scheduling policies* are “well-behaved”. We say that a scheduling policy is well-behaved if an execution trace for *any* input can be obtained by only eliminating some executions from the maximal execution trace, but not reordering any remaining executions. Though this property could certainly be design dependent, tests exist to identify scheduling policies that are well-behaved regardless of the design functionality. Details of these tests are beyond the scope of this paper, but we point out that many common implementations have scheduling

policies that are well-behaved. Single processors with list scheduling or static priority scheduling, synchronous circuits, and multiple clock circuits with fixed relationship between clocks are all well-behaved. Multi-processor implementations in general are not well-behaved.

The correctness of using maximal execution trace as communication signature hinges on lemma 2 and the following theorem:

Theorem 2 If the maximal execution traces are the same for two implementations whose scheduling policies are well-behaved, then their execution traces will be identical for all possible input traces.

Outline of Proof of Theorem 2 Since the scheduling policies are well-behaved, the real execution traces can be obtained by eliminating some executions from the (same) maximal ones. It is shown by induction on the length of the maximal trace that the same executions are eliminated for both implementations. 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. The maximal execution traces are the same for implementations A<B and A=B, and the two implementations indeed do have identical execution traces. According to Lemma 2, they are synchronously equivalent to each other. If the maximal execution traces are different, as they are between A=B and A>B, we cannot conclude whether the implementations are or are not synchronously equivalent.

The complexity of this algorithm is quadratic in the number of components since 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 components. We suggest a simple extension to deal with common loop structures in section 6.

5 Case Study

We applied abstract communication analysis to a real-life industrial design: a shock absorber controller [3]. 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. Attempts to verify it automatically with the tool VIS [5] exceeded available memory and time limits.

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

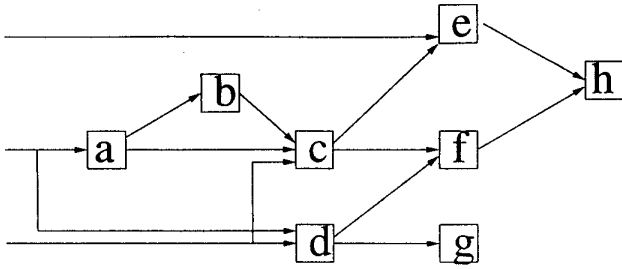


Figure 2: System Graph for Shock Absorber Controller.

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	001	1	10	11	1	10
3	1			010		01	10		10
	2			100		01	10		11
	3					01			11
4	1	1	1	111	1	11	11	1	11
	5	1	1	001	1	10	01	1	10
5	1			010		01	10		01
	2			100		01	10		01
	3					01	10		01
	4					01	10		10
	5								01
	6								10
	7								01
	8								10

Table 2: Maximal Execution Traces.

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 the table, we can conclude that implementation 1 and 3 are synchronously equivalent. Combining this result with theorem 1, we can conclude that any synchronous hardware implementation and any single processor implementation (with any delay characteristics) with the given list scheduling are synchronously equivalent. If they both satisfy timing constraints, implementation 3 may have a lower cost and implementation 1 may have better performance in terms of timing. We can also conclude similarly that implementation 2 and 4 are synchronously equivalent. The conservative analysis was performed in a very short computing time and with negligible memory occupation.

6 Summary and Future Direction

We proposed a definition of functional equivalence for embedded systems: synchronous equivalence. We also proposed simple algorithms 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 future direction is to increase the applicability of the abstract communication analysis to include architectures using more than one processor. We will probably need to make the synchronization explicit among resources. A third direction is to deal with timing issues such as tightening the bound on the 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] 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*, vol. 79, Sept. 1991.
- [3] 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.
- [4] K. L. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [5] R. Brayton, A. Sangiovanni-Vincentelli, G. Hachtel, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, S. Qadeer, R. Ranjan, T. Shiple, G. Swamy, T. Villa, A. Pardo, and S. Sarwary, "VIS: A system for verification and synthesis," in *Proceedings of Computer Aided Verification: 8th International Conference, CAV'96, Rutgers, NJ, July, 1996* (R. Alur and T. A. Henzinger, eds.), Springer-Verlag, 1996. LNCS vol. 1102.
- [6] F. Balarin and A. Sangiovanni-Vincentelli, "Schedule validation for embedded reactive real-time systems," in *Proceedings of the Design Automation Conference*, June 1997.
- [7] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *IEEE Proceedings*, Sept. 1987.
- [8] C. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the Association for Computing Machinery*, vol. 20, pp. 46 – 61, January 1973.