Refining Abstract Equivalence Analysis for Embedded System Design

Harry Hsieh  
Alberto Sangiovanni-Vincentelli  
Department of EECS  
University of California at Berkeley  
E-mail: {harry, alberto}@ic.eecs.berkeley.edu

Felice Balarin  
Luciano Lavagno  
Cadence Berkeley Laboratories  
E-mail: {felice, luciano}@cadence.com

Abstract

The synchronous assumption has made it possible to develop efficient procedures for establishing functional equivalence between different implementations in the domains of synchronous circuits and synchronous reactive systems. This notion is extended to embedded systems that do not satisfy the synchronous assumption inside their boundaries but only at the interface with the environment [5]. Efficient, but conservative, synchronous equivalence analysis algorithms have been developed. In this work, we propose extensions to these algorithms that allow trading off the complexity with the conservativeness of the results.

1 Introduction

Design space exploration is the process of analyzing several functionally equivalent alternatives to determine the most suitable one. A fundamental question is whether an implementation is consistent with the high-level specification or whether two implementations are “equivalent”. The synchronous equivalence for embedded systems is defined in [5], which strongly resembles the concept of functional equivalence for synchronous circuits.

We deal with specification in the form of asynchronous network of Finite State Machines [1]. Implementing a specification involves mapping FSM onto computational resources and synthesizing codes or design files. It is then necessary to figure out whether different mappings and different synthesis strategies result in implementations that are behaviorally equivalent. In [6], communication signature is established in the flavor of worst-case analysis in real-time scheduling [7]. If two implementations have the same communication signature, they are synchronously equivalent to each other. There is a simple intuition for the abstract analysis to focus on the communication between components. Since the corresponding components in any two implementations always have the same functionality, and the connectivity among the components is also the same, it is possible to deduce the equivalence of two implementations from the communication pattern alone. By looking at only the worst-case communication characteristics and not at the details, the analysis can be efficient. However, when two communication signatures are different, or when one or both of the communication signatures are infinite, the result of the analysis is inconclusive. In the context of design exploration, the inconclusive result in equivalence analysis may lead to final implementation that is suboptimal.

In this work, we identify and remove, bit by bit, the sources of these false negatives through the process of refinement and pruning. By successive applications of refinement and pruning, we establish a smooth path to exact simulation. If two implementations are equivalent, communication signatures, perhaps with some limited refinement and pruning, will be able to provide the answer. The analysis is formal, but is also much more efficient than what is possible with the state space search methods (e.g. [3]) or what is required to simulate the design exhaustively.

In the next section, we 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 some salient points of synchronous equivalence design methodology. Communication analysis operates at one specific level of abstraction between specification and implementation. In Section 3.2, we provide primitives to move around the abstraction/refinement continuum, trading off computational complexity of the algorithm with the conservativeness of the result. We discuss future directions in Section 4.
2 Network of CFSMs

Embedded systems can be represented as networks of interacting Co-design Finite State Machines (CFSMs) [1]. CFMS extend Finite State Machines with side-effect-free computation on the transition edges. The communication entities between CFMS are events, which may or may not carry values. A CFSM can transition only when an input event has “occurred”, and it “consumes” all input events that have occurred.

Each individual CFSM operates 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. The CFSM model of computation imposes only a minimal requirement, a one-deep buffer. Overwriting of events and values are possible.

At the “specification” level, designers specify only the structure of the design (i.e. I/O of the CFMSs) and the local (FSM) functions of the design. Implementing the specification involves allocating individual CFMSs to computational resources, assigning schedulers to shared resources, and synthesizing CFMSs into code or gate level description for the resources. At the specification level, a network of CFMSs is inherently non-deterministic. An implementation, however, is deterministic: its response is unique for any fixed input sequence. In fact, we extend the notion of implementation to include any set of rules that resolve non-deterministic choices in a CFSM network (making it deterministic). Mapping to a simulator can be considered as an implementation, since a CFSM network can be simulated only if it is deterministic. 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 a “golden” (simulation) model.

3 Synchronous Assumption and Synchronous Equivalence

**Definition 3.1 (Synchronous Assumption)** The operation of the design is split into two alternating non-overlapping phases, constituting a cycle. An interaction phase where the environment interacts with the design and a computation phase where the components in the design perform executions and communicate among themselves.

This notion of synchronicity is a system-level extension to the fundamental mode operation of asynchronous circuits [10]. We will only consider specifications that satisfy this synchronous assumption.

**Definition 3.2 (Synchronous Equivalence)**

*Under the synchronous assumption, two 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 (more precisely, of the computation phase).*

As long as the results (outputs of the network of CFMSs) are the same at the end of every cycle, the delay of the executions of CFMSs, the order of the executions, or even the parallel/serial nature of the executions do not matter to the functional correctness of the design. These may lead to freedom in, respectively, synthesis and optimization, scheduler selection, and assignment of components to computational resources.

3.1 Design Exploration Methodology

Figure 1 illustrates a possible design methodology using the synchronous assumption and synchronous equivalence. The designer specifies the functionalities of the components and the structure of the design as a network of CFMSs. One or more behaviors among those allowed by the non-determinism in the network are chosen as golden model(s). They are represented by reference implementations that produce those functional behaviors and only those functional behaviors. The functional behaviors under synchronous assumption of these implementations are considered “correct” and different implementations with the same functional
behavior are all functionally equivalent. Separate analysis, in the flavor of [2], is performed to make sure that non-functional constraints are satisfied and to decide whether one implementation is superior to others given some design metrics. The best one is chosen to be synthesized.

The result of a conservative equivalence analysis algorithm can be true positive, or inconclusive due to true negative, false negative, or space/time out. The analysis techniques span a continuum on computation time and conservativeness of the analysis, as shown in Figure 2. Conservativeness is measured by the number of false negative results produced by the analysis algorithm, given a set of implementations of a CFSM specification. Exact analysis of any form produces no false negative results, but it often has very long computation time. Conservative analysis methods require less computation time, at a cost of false negative results. Delay insensitivity and communication analysis were introduced in [5] and [6]. Next, we review communication analysis, and then propose its refinement in section 3.2.

3.2 Communication Analysis

Communication analysis [6] can be used to conservatively check the equivalence between two implementations with different scheduling policies. Communication signature is established such that if two implementations have the same communication signature, they are synchronously equivalent to each other.

One such proposed signature is the execution covers (ECs) of an implementation. An execution cover of a given CFSM network is a directed acyclic graph. EC nodes can be thought of as "containers" representing possible events, in the flavor of event graph nodes representing partially ordered histories [8], but using the additional notion of possibility to further abstract them. Thus, a container can contain either a "0" (event absence) or a "1" (event presence). However, if the container is absent, it indicates a 0, i.e. the corresponding event is absent. Roughly speaking, edges in the ECs represent dependencies between input and output events, as well as the ordering among events belonging to the same signal. To obtain its EC, the implementation is abstractly executed with the CFSMs replaced by their least non-decreasing cover. The least non-decreasing cover of some CFSM C has the same inputs and outputs as C, and it produces a container on its output if and only if there exists a state of C and an assignment of 0's and 1's to containers on its inputs for which C would produce an event. 

Figure 2. Trade-Off in Analysis Methods.

3.3 Container Refinement

The abstract nature of EC gives it efficiency, but also yields many inconclusive results. Consider, for example, the system in Figure 3, and two of its implementations, one using static priority scheduling (SPS) with priority of A less than priority of B, and the other with cyclic executive scheduling (CES) where in each cycle A executes before B. It can be established through exhaustive simulation that not only these two implementations produce finite output traces for any finite input trace, but also that they are synchronously equivalent. However, both implementations have infinite ECs and communication analysis returns inconclusive result.

The infinite ECs are due to the abstraction of event containers. To make the abstract simulation finite, containers need to be "refined" to take on more precise information. To do so, we extend the definitions from [6] to ternary space. 1, or 1-container, denotes the presence of the event and the container. 0, as before, denotes the absence of the container and event. x, or x-container denotes the presence of the container only, which may correspond to either the presence (1) or the absence (0) of the event.

Ternary execution covers for an implementation with a given scheduling policy can be obtained by the Ternary Execution Cover Generation Procedure:

**Step 1**: Create containers for primary inputs and la-
bel them with level 0. Set the current level to 0. The input container may be filled, left with content unknown, or not be created at all, depending on what is known about the inputs. For example, if an input is known to be present for all cycles, then it can be set to be a 1-container.

**Step 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 the previous execution level of that CFSM (i.e. there exists a container that has not been “consumed” by the previous execution).

**Step 3**: Let all CFSMs with at least one active input be enabled. If no CFSM is enabled, then STOP.

**Step 4**: Apply the original scheduling policy to choose the CFSMs to be “executed”.

**Step 5**: Abstractly execute the selected CFSMs by increasing the current level by 1 and evaluating a ternary non-decreasing cover of each CFSM output function, with all active inputs with a 1-container set to 1, all active inputs with an x-container set to x, and all other inputs set to 0. If the ternary non-decreasing cover evaluates to 1 or x then:

- create a new 1-container or x-container, respectively, 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.

**Step 6** Go to Step 2.

The ternary non-decreasing covers in Step 5 are computed by converting the output functions to a ternary non-decreasing cover. Any non-decreasing cover will do. If we cannot take away any minterm without making the function non-covering or decreasing, the function is then the least ternary non-decreasing cover. Unfortunately, the iterative technique for the binary container space described in [6] can not be applied here. We only note that similar ternary functions are computed also for symbolic simulation in the logic domain [9].

![Figure 4. Refined ECs for Example in Figure 3.](image)

The ternary least non-decreasing covers for the example in Figure 3 are:

\[
\begin{align*}
o_1^{(1)} &= i_1^{(1)} * i_2^{(0)} + o_2^{(1)} * i_2^{(0)} \\
o_1^{(0)} &= i_2^{(1)} + i_1^{(0)} * o_2^{(0)} \\
o_1^{(x)} &= o_1^{(1)} + o_1^{(0)} \\
o_2^{(1)} &= i_2^{(1)} * i_1^{(0)} + o_1^{(1)} * i_1^{(0)} \\
o_2^{(0)} &= i_1^{(1)} + i_2^{(0)} * o_1^{(0)} \\
o_2^{(x)} &= o_2^{(1)} + o_2^{(0)}
\end{align*}
\]

Abstract execution of the example in Figure 3, with \(i_1\) refined at the primary input, produces the execution covers shown in Figure 4. We now have a pair of “split” execution covers. One, on the left side of the figure, with \(i_1=1\), denoted by a solid box. The other, on the right side of the figure, with \(i_1=0\), denoted by no box at \(i_1\). If the corresponding portions of the execution covers are identical, we can say that the two implementations are synchronously equivalent to each other, since \(i_1\) can only be 0 or 1 in the original event space.

The refined ECs are still infinite. Further refinement is needed. The new execution covers are shown in Figure 5. After both \(i_1\) and \(i_2\) input containers are refined, the corresponding ECs are indeed identical. The implementations are therefore synchronously equivalent to each other.

We are able to (conservatively) check two implementations for synchronous equivalence by comparing their ternary ECs, as stated by the following result.

**Theorem 3.1** If two implementations of a given CFSM network with well-behaved scheduling policies have identical ternary ECs, then they are synchronously equivalent.

The proof of the theorem can be found in [4].
3.4 State Refinement

For some designs, the false negatives from communication analysis cannot be removed without explicitly representing states. Consider the example in Figure 6. The ternary least non-decreasing covers for the functions are:

\[
\begin{align*}
m_1^{(1)} &= \text{false} \\
 m_1^{(0)} &= o_1^{(0)} \\
o_1^{(x)} &= o_1^{(x)} + o_1^{(1)} \\
o_1^{(1)} &= m_1^{(1)} + i_1^{(1)} \\
o_1^{(0)} &= m_1^{(0)} + i_1^{(0)} \\
o_1^{(x)} &= m_1^{(x)} + i_1^{(0)} + m_1^{(0)} + s_1^{(x)} + m_1^{(x)} + i_1^{(x)}
\end{align*}
\]

The ternary ECs for two selected scheduling policies are shown in Figure 7. UDP denotes a Unit Delay Parallel scheduling policy which corresponds to a synchronous circuit implementation. One cannot conclude whether or not the two implementations are synchronously equivalent because the ECs are infinite. It can be shown that no amount of event container refinement can make the execution covers finite.

Without loss of generality, we assume that there is only one state variable per CFSM. The single state variable is one-hot encoded into a set of state value signals. Each state value signal is represented by separate events/containers. A 1-container at a state value signal means that the CFSM is at that state. An x-container at more than one state value signals means that the CFSM may be in any one of those states. State values have the following property.

**Lemma 3.1** A set of one-hot encoded state value signals for a CFSM has a 1-container if and only if it is the only container present for that set of state values.

**Proof:** By definition, any CFSM must be in some state at any given time. Only one container present means that there is only one state that it can "possibly" be in. The "event" is therefore definitely present. Conversely, if there is a 1-container at some state value, then the CFSM cannot possibly be in any other state, so other state values must not have any container present.

Due to the lemma, we will convert state value signals with a single x-container immediately into a 1-container. Ternary execution covers for an implementation with a given scheduling policy and some state refinement can be obtained by the Ternary Execution Cover Generation Procedure with the following modifications:

**Step 1 modified:** Create containers for primary inputs and state values of the CFSMs and label them with level 0. Set the current level to 0. The input containers may be filled, left with content unknown, or not be created at all, depending on what is known about the inputs and reachable states.

**Step 3 modified:** Let all CFSMs with at least one non-state active input be enabled. If no CFSM is enabled, then STOP.

**Step 5 modified:** Abstractly execute the selected CFMSs by increasing the current level by 1 and
evaluating a ternary non-decreasing cover of each CFSM output function, with all active inputs with a 1-container set to 1, all active inputs with an x-container set to x, and all other inputs set to 0. If the ternary non-decreasing cover evaluates to 1 or x then:

- create a new 1-container or x-container, respectively, 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,
- for state value containers, additionally create an edge from all previous state value containers to all new state value containers.

The implementations for Figure 6 can be shown to have identical ECs after the states are refined.

3.5 Pruning Execution Covers

Synchronous equivalence, as defined in Section 3, relates to the primary inputs and outputs of a design. Communication analysis infers this global property from the communication between components, which is not a strictly global characteristic. Inferring a global property from characteristics of local elements can be conservative. Consider the example in Figure 8. ECs for three different scheduling policies are shown in Figure 9. It can be established through exhaustive analysis, or simple observation, that the three implementations are actually synchronously equivalent to each other, though there are two different sets of ECs. It is equally apparent that many containers in the ECs have nothing to do with the primary output o1 and are indeed unobservable from the primary outputs.

Definition 3.3 (Observable Container) A container c in an execution cover is observable if there is an assignment of x-containers, consistent with transition and output relations, such that assigning 0 or 1 to c produces different primary outputs for the actual implementation. Otherwise, c is unobservable.

The next lemma clarifies the relationship between an implementation producing a primary output, and the primary output containers in an execution cover.

Lemma 3.2 If there are one or more 1-containers at some primary output of an execution cover, the output is emitted by the corresponding implementation.

Proof: By the definition of synchronous equivalence (Definition 3.2), primary outputs matter only at the end of a cycle. By the definition of CFSM, an emitted output cannot be "cancelled".

A container that is not observable can be removed from the execution cover.

Theorem 3.2 Removing unobservable containers, along with all their edges, results in an execution cover that remains a communication signature of the implementation.

Proof: Consider an implementation, A, its execution cover, C, and an execution cover, C^Pruned, where some unobservable containers have been removed. Let A^Pruned be an implementation that will produce the execution cover C^Pruned. By the definition of unobservable containers and synchronous equivalence, A^Pruned is synchronously equivalent to A. By transitivity of equivalence relations, any implementation that is synchronously equivalent to A^Pruned is also synchronously equivalent to implementation A.
Figuring out whether a container is observable is analogous to the classical observability problem in sequential circuits. Finding an exact solution could require an exponential computation, though simple pruning can go a long way in removing a large number of these unobservable containers.

**Corollary 3.1** If there are one or more 1-containers at some primary output of an execution cover, all but one of the 1-containers of that primary output can be removed and the execution cover remains a communication signature of the implementation.

*Proof:* By Lemma 3.2, the output will be emitted if one or more 1-containers exist at some primary output of the execution cover. This will remain true if all but one of these 1-containers are removed. By Theorem 3.2, the pruned execution cover is a communication signature of the implementation.

**Corollary 3.2** Any non-primary-output container that has no directed edge to an x-container can be removed and the execution cover remains a communication signature of the implementation.

*Proof:* Assigning 1 or 0 to a non-primary-output container with no directed edge to an x-container clearly cannot affect the primary output, so the container is unobservable and can be removed. Removing non-primary-output 1-container that has no directed edge to an x-container also cannot affect the primary output. By Theorem 3.2, the pruned execution cover is a communication signature of the implementation.

A large number of unobservable containers can be removed by the following procedure:

1. For all primary outputs that have two or more containers and at least one 1-container, remove all containers of that primary output except a single 1-container.

2. Iteratively remove all non-primary-output x-containers and 1-containers that do not have an edge to an x-container.

The pruned execution covers for the example in Figure 8 (derived by pruning the original execution cover in Figure 9) are shown in Figure 10. The original primary input containers are retained for clarity. The implementations are indeed synchronously equivalent to each other.

### 3.6 Relationship with Exact Simulation

With container refinement and pruning, execution covers can be related to exact simulation. Exact simulation requires deterministic transition and output relations, as well as a deterministic input trace. A deterministic input trace for an actual implementation corresponds to an "abstract" execution with an input trace consisting of only 1-containers.

**Theorem 3.3** Given deterministic transition relations and output relations, a well-behaved scheduling policy, and a deterministic input trace, the output trace from simulation is identical to a trace of the pruned execution cover from the corresponding abstract simulation of the cover functions.

*Proof:* The transition and output relations are deterministic, and the primary input of the abstract simulation consists of only 1-containers. By Step 5 modified of the ternary EC generation procedure, only 1-containers will exist in the execution cover. By Corollary 3.1 and Corollary 3.2, only a single 1-container at each primary output will remain. The trace of the execution cover is therefore a trace of 1-containers at the primary output. In addition, a container will be present for every event present in the output trace of the corresponding simulation run.

The theorem relates a simulation run of the implementation to an abstract execution of its non-decreasing covers. However, not all refined pruned execution covers have corresponding simulation runs, since not all combinations of the states of the components are reachable, and the given pruning algorithm does not identify all unobservable containers. The former
can be solved by existing state reachability technique on the global state, and the later can be solved by further refining remaining x-containers, both at a cost of higher computational complexity.

4 Conclusions and Future Directions

In this paper, we related communication analysis to exact simulation through a series of refinement and pruning operations on the communication signatures. An algorithm can choose to work at any abstraction level trading off computational efficiency with the possibility of inconclusive result due to false negatives. We provided primitives to move amongst different abstraction levels that exist between communication analysis and exact simulation. Some of the future directions may include applying iterative refinement techniques to choosing which container to refine. It may also be interesting to see what effect repartitioning the FSM network will have on the problem of equivalence analysis.

References


