Formal Verification of Embedded Systems based on CFSM Networks *

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

Abstract

Both timing and functional properties are essential to characterize the correct behavior of an embedded system. Verification is in general performed either by simulation, or by bread-boarding. Given the safety requirements of such systems, a formal proof that the properties are indeed satisfied is highly desirable. In this paper, we present a formal verification methodology for embedded systems. The formal model for the behavior of the system used in POLIS is a network of *Codesign Finite State Machines*. This model is translated into automata, and verified using automata-theoretic techniques. An industrial embedded system is verified using the methodology. We demonstrate that abstractions and separation of timing and functionality is crucial for the successful use of formal verification for this example. We also show that in POLIS abstractions and separation of timing and functionality can be done by simple syntactic modification of the representation of the system.

1 Introduction

Design verification of embedded systems is typically performed by prototyping and simulation. Prototyping is clearly expensive in terms of turn-around time, and, in addition, cannot be performed until most of the detailed design is completed. Simulation is valuable, but for complex systems, only relatively few input patterns can be tried.

Formal verification is a set of techniques that allow for proving mathematically that some formally specified properties are true for a design. Formal verification requires a formal model of the behavior of a system, as well as the properties we wish to verify. In our approach to HW/SW codesign, we describe systems with Codesign Finite State Machines (CFSM's) [1], a model that is abstract enough to include behaviors of all possible implementations. *POLIS* [1, 2] is a codesign environment for embedded systems based on this model.

*This work was partially supported by SRC Contract DC-324-028, and by MURST under project "VLSI Architectures".

In this paper, we show how an automata-theoretic approach to formal verification can be applied to CFSM's. In the automata theoretic approach [3], systems are modeled by *finite-state automata*, and the language of the automaton (i.e. a set of sequences of inputs and outputs observable at the ports of the system) is taken to be the behavior of the system. The task of formal verification is to show that all these sequences are "acceptable". Acceptable sequences are specified as a language of another automaton, so the verification problem reduces to checking language containment between two automata.

The main advantages of this approach are that it can be completely automated, and that it allows conservative abstractions to reduce the complexity of the computation. Let A be some automaton. If A is modified (say to A') in a manner that can only add new sequences, but never eliminate a sequence from the language, then A' is a conservative abstraction of A in sense that A satisfies all the properties that A' does (i.e. the language of A is contained in all the languages that the language of A' is).

Neither CFSM nor automata deal with quantitative timing issues. In fact, CFSM's are build on the assumption that a reaction to an event can take an unbounded amount of time. This assumption provides an implementation-unbiased starting point for a verification-driven design methodology [2]. The first step in that methodology is to try to verify the system with unbounded delays. If the verification fails (as it most often will), the error trace is analyzed in order to suggest timing constraints necessary to satisfy the property. Then, the verification is attempted under the assumption that timing constraints are met. If successful, the used assumptions are recorded as constraints to be met by the implementation. In this way, a verification tool is used as design aid to refine the specification of the system.

The rest of this paper is organized as follows. CFSM's are reviewed in section 2. The original work on CFSM's [1] included automata-based semantics of CFSM's, but the description was still abstract, leaving some details unspecified. Therefore, in section 3, we present a detailed construction of an equivalent automaton for a CFSM, which also includes some minor semantic changes in the model that occurred since the original work. In section 4 we present a case study: the verification of properties of a real-life design.

2 Codesign Finite State Machines

A Codesign Finite State Machine (CFSM) is basically constituted by a set of input events (each with its associated

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA 0-89791-833-9/96/0006..$3.50

set of values), a set of output events (each with its associated set of values and possibly with an initial value), and a transition relation.

The transition relation describes how input events can cause output events. It is a *set* of *pairs of sets*. The first member of each pair is a set of *valuations of input events*. The second one is a set of *valuations of output events*. Each transition is *triggered* by the input events with the appropriate values and *emits* the output events with the appropriate values. The *reaction time* (i.e. the time between each input event and each output event) is *unbounded* and *non-zero*.

Formally, a CFSM is given by:

- a finite set I of input events,
- a finite set O of output events,
- a finite set $V_I \subseteq \{(i,v) | i \in I\}$ of possible valuations of input events, and a finite set $V_O \subseteq \{(o,v) | o \in O\}$ of possible valuations of output events,
- a set $R \subseteq V_O$ of possible initial values of (some) output events,
- a *transition relation* $F \subseteq \{(f^I, f^O) \,|\, f^I \subseteq V_I, f^O \subseteq V_O\}$.

The operational cycle of a CFSM goes through the following four phases:

1. idle,
2. detect input events,
3. transition, according to which events are present and match a transition relation element,
4. emit output events.

3 Modeling CFSM's with Automata

We interpret CFSM's as automata with one-place input buffers to store incoming events, and one-place output buffers to store events to be emitted. More precisely, for any CFSM $\mathcal{C}$ we construct an automaton C whose alphabet consists of all possible assignments to the set of variables containing:

- for every input $i \in I$: a binary variable $*i$ (indicating an occurrence of the corresponding event), a multi-valued variable i (indicating a value of the event), and an auxiliary binary variable aux_*i,
- for every output $o \in O$: a binary variable $*o$ (indicating an occurrence of the corresponding event), a multi-valued variable o (indicating a value of the event), an auxiliary binary variable aux_*o, and an auxiliary multi-valued variable aux_o,
- a binary variable go indicating an occurrence of a transition.

The automaton C is built as the composition of the following automata:

- one "input-buffer" automaton IB_i for every input $i \in I$,
- one "output-buffer" automaton OB_o, and one "output-value-buffer" OV_o, for every output $o \in O$,
- the main automaton M.

The main automaton M is a single-state automaton that accepts only those signals which are consistent with the transition relation of the CFSM $\mathcal{C}$. It communicates with other automata through auxiliary variables, which are then appropriately buffered by other automata. It also sets variable go to indicate when transitions are taken. More precisely, the transition relation of M contains the following assignments to alphabet variables:

- $go = 0$, $aux\_*o = 0$ for all outputs o, and arbitrary values of all other variables (these transitions model idling),
- for every (f^I, f^O) in the transition relation of the CFSM:
  - $go = 1$,
  - $aux\_*i = 1$, $i = v$ if (i,v) in f^I, and $aux\_*i = 0$, i arbitrary if $\forall v : (i,v) \notin f^I$,
  - $aux\_*o = 1$, $aux\_o = v$ if (o,v) in f^O, and $aux\_*o = 0$, aux_o arbitrary if $\forall v : (o,v) \notin f^O$.
  - $*i$ for every input $i \in I$, and $*o$ and o for every output $o \in O$ all arbitrary.

The input-buffer automaton IB_i has a single binary state variable indicating whether an event i has occurred since the last transition of an CFSM. When the CFSM transitions (i.e. when $go = 1$), it will pass that information to the main automaton, by assigning the appropriate value to variable aux_*i. More precisely, the initial value of the state variable is 0, and the transition relation is given by:

if $go = 1$ then $\{aux\_*i := state \vee *i;\ state := 0;\}$
else $\{aux\_*i := 0;\ state := state \vee *i;\}$

Note that if $*i$ occurs at the same time as go, then it is not latched. Instead, it is passed to aux_*i.

The output-buffer automaton OB_o has a single binary state variable indicating whether there is an event which has to be emitted as a part of the previous transitions. The event can be emitted at any time *after* the transition, but no later than at the time of the next transition. Whether the event is to be emitted is signaled at the time of transition by the main CFSM through the variable aux_*o. The initial value of the state variable is 1 if some initial value is specified for that event, and 0 otherwise. Formally, the initial value is 1 if $(i,v) \in R$ for some v. The transition relation of OB_o is given by:

if $go = 1$ then $\{*o := state;\ state := aux\_*o;\}$
else choose $*o := 0;$
or $\{*o := state;\ state := 0;\}$

where the construct "choose A or B" indicates that either A or B are randomly chosen to be executed.

The output value buffer automaton OV_o maintains the value of the event. When a CFSM makes a transition that emits o, OV_o stores its value in variable int (for internal), and when the event is actually emitted, it makes that value visible to the rest of the world by moving it to the state variable ext (for external). More precisely, the transition relation is:

if $*o = 1$ then $\{o := int;\ ext := int;\}$
else $o := ext;$
if $aux\_*o = 1$ then $int := aux\_o$

The initial value of int is arbitrary. If some initial values for o are specified in R, then ext can initially take any of these values. Otherwise, the initial value of ext is arbitrary.

4 A Case Study

In this section we present the verification of a shock absorber controller [4]. The controller sets shock absorbers' motors to HARD, MEDIUM, or SOFT level, according to values from a set of sensors: steering wheel, vertical acceleration, speed, and battery voltage. The system consists of modules that compute the horizontal speed, horizontal acceleration, vertical acceleration, steering angle and steering speed from the input sensors, and suggest the shock absorber level based on the appropriate look-up table, a module that signals if the battery voltage is out of the given range, and a module that records current suggestions of all other modules and sets the motors to the hardest of suggested values.

During normal operation, the output is continuously updated according to input sensors. At the same time, input modules continuously check input sensors for erroneous conditions, and signal if such a condition is detected. The specification requires seven different conditions to be checked, and appropriate actions to be taken if these are detected. In this paper we present in detail verification of the following condition and action:

P1: Speed parasitic(glitch): If the speed sensor indicates impossibly high speed on more than three occasions, the shock absorbers are to be set to HARD until the RESET event occurs.

Since all other properties have a similar form, they are verified in a similar way. Verifying one property at a time enables us to abstract events and modules not related to the property at hand.

We now describe in detail intended behavior of the system relevant to property at hand. The error is first detected by the module called SPEED_DIAG_PAR. To understand the behavior of that module more precisely, we first need to explain how speed is calculated. The speed sensor generates a sequence of events (called SPEED_SENS), the frequency of which is proportional to the speed of the vehicle. Speed is calculated by counting clock events (called CLOCK_500) between any two occurrences of SPEED_SENS events. The count is stored in variable D_TIME. Even at the highest possible speed of the vehicle, there can be no less than 6000 CLOCK_500 events between two SPEED_SENS events. Thus, if D_TIME is less than 6000 when the new SPEED_SENS event occurs, that must be a parasitic signal (glitch). Upon detection of a glitch no immediate action is taken. However, the number of detected glitches is recorded (in variable MIN_TPAR_NUM), and if that number is larger or equal than 3, then an error event (called ERR_PAR) is emitted.

The ERR_PAR event is received by a module called MOT_CTRL_DAMAGE. It records error events from all the input modules (including SPEED_DIAG_PAR), and generates the DAMAGE event with a value HARD, MEDIUM, or SOFT, as required by the specification. This error status is stored and can only be cleared upon RESET.

The DAMAGE event is received by a module called DRIVER. It also receives another event called COMMAND_IN (the suggested setting of motors computed from look-up tables used in normal operation), stores them both, and generates the COMMAND_OUT event with the harder of two values.

The verification can be further simplified by decomposing a property into "local" sub-properties that each module must satisfy. In particular, we prove property **P1** by proving the following:

P1.1: If the speed sensors indicate impossibly high speed at least three times and no RESET events occur, then ERR_PAR event will be generated.

P1.2: If the MOT_CTRL_DAMAGE module receives ERR_PAR event, and no RESET events occur, then a DAMAGE event with a value HARD will be generated, and no DAMAGE event with some other value will be generated before the RESET event occurs.

P1.3: If the DRIVER module receives DAMAGE event with value HARD, then the setting of the shock absorber will be set to HARD, and the setting will not change until either a RESET event, or a DAMAGE event with a value other than HARD occur.

Often, properties decompose naturally into local sub-properties, and a simple check is needed to verify that sub-properties indeed imply the desired property, but this is not always true. Finding a good decomposition can be a hard task, and it cannot be completely and efficiently automated. Also, in non-trivial cases one needs to use another formal technique (e.g. automated theorem proving), to show that sub-properties imply the property.

4.1 Verifying property P1.1

The precise formulation of the error condition is as complex (and thus as error-prone) as the description of the module. To avoid the problem of false property specification, we verify the following simple property instead:

P'1.1: If four SPEED_SENS events occur between two CLOCK_500 events, then ERR_PAR event is generated.

This obviously covers only a small portion of the intended behavior, but it is often the case that such simple "sanity checks" reveal interesting bugs in the system.

The automaton obtained from the CFSM description of the original module by the procedure described in section 3 has 141 binary latches. The formal verification tool HSIS [5] runs out of the 480Mb of main memory before even constructing the internal representation.

Verification with abstracted integers

The CFSM, on closer inspection, consists mainly of two 16-bit integers used as counters, and two comparators comparing these integers to constants. However, for the property we are interested in only one value of D_TIME is distinguished: 0, indicating that no CLOCK_500 events have occurred between two SPEED_SENS events. We can thus abstract this counter to only two states: "0" and "> 0", and modify the comparator and incrementer accordingly. If the input of the incrementer is 0, the output must be > 0, and if the input is > 0 the output is chosen non-deterministically to be > 0 or 0 (due to possible overflow). Similarly, the comparator checking whether D_TIME is less then 6000 is modified so that if its input is 0 the output is 1, and if its input is > 0, the output is either 0 or 1. In a similar way, we abstract MIN_TPAR_NUM to four distinguished values: 0, 1, 2, and ≥ 3. These abstraction introduces some new behaviors (e.g. when the actual value of D_TIME is larger than 6000 but the comparator still outputs 1), but these additions do not affect the outcome of verification. It is important to notice

that this abstraction can be done automatically by syntactic modification of the intermediate description of the system, so that the size of the description is guaranteed not to increase. However, deciding which abstractions are consistent with the property is as hard as the original verification problem, so the user guidance is crucial in this step.

With these abstractions, building and verifying the model takes less than 10 seconds. Unfortunately, the property fails. The error trace indicates that the CFSM can react too slowly, so SPEED_SENS events can be over-written without being sensed by the CFSM (e.g. if it is not assigned a high enough priority by the scheduler).

Verification under timing assumptions

Indeed, the property can be satisfied only if some timing assumptions are made about the CFSM. We will follow the usual methodology of separating timing and functional properties as much as possible. More precisely, we will verify the conditional property: "if the CFSM reacts to every SPEED_SENS event, then the property holds". Such a property is local, i.e. it depends only on the behavior of the CFSM. On the other hand, verifying that indeed the CFSM reacts to every SPEED_SENS event is a separate problem with which we will not deal here. We will just note that it is a hard problem, because it involves the characteristics of the hardware, of all the other tasks in the system, as well as those of the scheduling algorithm.

Fortunately, there is a very simple way to restrict the behavior of the system to the case where no input events are ever over-written. It suffices to remove from the transition relation of the input buffers IB_i all elements where *state* and $*i$ are both 1. Again, this modification requires a simple syntactic change on the intermediate description, which does not affect its size.

The conditional property showed an error in the design. Under the original specification, the property is not satisfied if the error condition occurs immediately after initialization. After correcting this error, the conditional property is verified in less than 10 seconds of CPU time.

4.2 Verifying property P1.2

The automaton modeling MOT_CTRL_DAMAGE has 81 binary latches. The tool HSIS [5] could not construct the internal representation of the module even after several hours of computation time.

Timing assumptions and freeing variables

For the property at hand, values of all internal variables except the one holding the value of ERR_PAR, are irrelevant. We can thus "free" them, i.e. allow them in every execution step to take any value from their respective domains, and remove from the model all computation of these variables. Again such an abstraction can be done automatically with a single pass through the intermediate description of the system, but deciding which variables can be freed is generally a hard problem.

With this abstraction the verification time is reduced to few seconds. The property is verified under the assumption that the system cannot ignore forever an input event. Note that this is a weaker assumption than the one requiring that every input event is reacted to, because it still allows "overwriting" of input events. The verification of the conditional property takes less than a minute of CPU time.

4.3 Verifying property P1.3

The module DRIVER is simple enough that it can be verified without any abstractions. During the verification, an error was found: (DAMAGE event was ignored if COMMAND_IN event was occurring simultaneously). After correcting the description of the DRIVER module, the property is verified.

5 Conclusions

We showed how the behavior of Codesign Finite State Machines can be represented by automata with a reasonably small increase in the size of the representation. We have also showed on a real example how existing formal verification tools can be used to verify properties of CFSM's.

In course of this exercise we have observed that reactive systems of interest often have state spaces which are too large to handle for existing formal verification tools, thus the use of abstraction is crucial. Typical abstractions can often be performed automatically, by syntactic modification of the intermediate representation of automata. However, choosing the abstraction that is appropriate for the property to be verified can only be done by a designer with knowledge of the intended behavior of the system.

We have also observed that even though CFSM's are based on the unbounded delay assumption, most of the properties can be verified only if some timing constraints are imposed on the behavior of CFSM. The assumption of the CFSM reacting to *every* input events provides a convenient way to separate timing and functional considerations. Restricting the behavior of the system such that the assumption is satisfied can also be performed by simple syntactic modification of the intermediate representation.

References

[1] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. A formal specification model for hardware/software codesign. Technical Report UCB/ERL M93/48, U.C. Berkeley, June 1993.

[2] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. A formal methodology for hardware/software codesign of embedded systems. *IEEE Micro*, August 1994.

[3] R. P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, 1994.

[4] L. Lavagno, M. Chiodo, P. Giusto, H. Hsieh, S. Yee, K. Suzuki, A. Jurecska, and A. Sangiovanni-Vincentelli. A case study in computer-aided codesign of embedded controllers. In *Proceedings of the International Workshop on Hardware-Software Codesign*, 1994.

[5] T. Shiple, A. Aziz, F. Balarin, S. Cheng, R. Hojati, T. Kam, S. Krishnan, V. Singhal, H. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Formal design verification of digital systems. In *Proceedings of TECHCON*, 1993.