Static Analysis of HDL Descriptions: Extracting Models for Verification

Alexander Kamkin*, Sergey Smolov*, Igor Melnichenko†

*Institute for System Programming of the Russian Academy of Sciences (ISPRAS),
†OJSC “The Institute of Electronic Control Computers named after I.S. Bruk” (INEUM)
{kamkin, ssedai}@ispras.ru, i.melnichenko@deltasolutions.ru

Abstract

The increasing complexity of hardware designs makes functional verification a challenge. The key issue of the state-of-the-art verification approaches is to obtain a “good” model for automated test generation or formal property checking. In this paper, we describe techniques for deriving EFSM-based models from HDL descriptions and briefly discuss applications of such models for verification. The distinctive feature of the suggested approach is that it automatically determines what registers of a design encode its state and use this information for model reconstruction.

1. Introduction

It is well known that functional verification is a bottleneck of the hardware design cycle. To automate verification, different kinds of models are in use. Some of them are built from requirements (properties, use cases, etc.); others are extracted from code (control flow graphs, state machines, etc.). The paper focuses on models of the second type and their application to simulation-based and formal verification. Specifically, we consider static, compile-time derivation of extended finite state machines (EFSMs) from design source code written in a hardware description language (HDL).

The EFSM formalism extends the classical FSM model by adding (1) input and output parameters, (2) registers and (3) transitions’ guards and actions defined over registers and input parameters. The main idea is to clearly separate control and datapath (exactly as it is acknowledged in hardware design and synthesis). EFSM-based models are intensively used in functional test generation (to generate conformance test suites [1] or to cover unlikely corner cases [2]) as well as in formal verification, especially in model checking (to check properties or to find errors and inconsistencies in a design model).

In this paper, we introduce a method for deriving EFSM models from HDL code and briefly discuss some EFSM-based verification techniques. The remainder is organized as follows. Section 2 reviews related work dealing with EFSM extraction and using EFSM models for verification of HDL descriptions. Section 3 introduces the basic notions used in the work. Section 4 describes the main concepts of the approach. Section 5 reports experimental results. Finally, Section 6 concludes the paper and outlines directions of future research.

2. Related work

There are not so many approaches to hardware verification using EFSM abstraction. The paper [3] presents a FAST framework that accelerates fault simulation (mutation testing) through automatic abstraction of HDL descriptions (code with injected faults) into event-equivalent transaction-level models (TLMs). Fault injection/simulation techniques have been widely investigated in the past decades. The most interesting part of the work is RTL-to-TLM abstraction, which, in turn, is based on EFSM extraction. Given an EFSM model, the tool is able to recognize computational phases (paths that must be traversed to (1) get the input data, (2) elaborate them, and (3) produce the output result). Abstraction is carried out by collapsing computational subphases according to a set of state-merging rules.

Some papers address the problem of EFSM-based test generation [1,2,4,5]. If EFSMs are derived from HDL descriptions, such tests are able to achieve high level of code coverage and, as opposed to [3], to cope with state dependencies. The dominant approach used in the area (as well as in model checking) is state graph traversal. Comparing with the FSM-targeted methods, there is an issue related to transition guards. When

---

This work was supported in part by the Russian Foundation for Basic Research under grant 12-01-31343.
guards depend on registers (not only on inputs), it is hard to determine whether a path is feasible or not. There are techniques for EFSM transformation that eliminate dependencies [1,4,5], but they may lead to state explosion. An alternative is to use backtracking (and backjumping) to search for reachable states [2].

In [2], generation of “easy-to-traverse” EFSMs from HDL code is described. The process consists of the four steps: (1) an initial EFSM (REFSM, reference EFSM) is extracted from code by using standard routines [6]; (2) new states are added to the REFSM to split transitions into “smaller” ones, with no conditional statements in actions (LEFSM, largest EFSM); (3) to be time-equivalent to the initial REFSM, the LEFSM is transformed by grouping compatible transitions (SEFSM, smallest EFSM); (4) the SEFSM is stabilized [5] on the register encoding the design state ($S^2$EFSM, semi-stabilized EFSM). The method is rather powerful, but, we believe, it could be improved if it took notice of the state register(s) from the beginning.

3. Preliminaries

Let $V$ be a set of variables. A valuation is a function that associates a variable $v \in V$ with a value $[v]$ from the corresponding domain. The set of all valuations of $V$ is denoted as $Dom_v$. A guard is a Boolean function defined on valuations ($Dom_v \rightarrow \{true, false\}$). An action is a transform of a valuation ($Dom_v \rightarrow Dom_v$). A pair $\gamma \rightarrow \delta$, where $\gamma$ is a guard and $\delta$ is an action, is called a guarded action (GA). Note that when we speak about a function, it is implied that there is a description of that function in some HDL-like language (thus, we can reason about syntax, not only semantics).

An EFSM is a tuple $(S, V, T)$, where $S$ is a set of states, $V = I \cup O \cup R$ is a set of variables, consisting of inputs (I), outputs (O) and registers (R), and $T$ is a set of transitions (all sets are supposed to be finite). Each $t \in T$ is a tuple $(s_n, \gamma, \delta, s', \delta')$, where $s_n$ and $s'$ are, respectively, the initial and final states of $t$, and $\gamma$ and $\delta$ are, respectively, the guard and the action of $t$. A valuation $v \in Dom_v$ is called a context, while a pair $(s, v) \in S \times Dom_v$ is referred to as a configuration. A transition $t$ is said to be enabled for a configuration $(s, v)$ if $s_n = s$ and $\gamma(v) = true$.

Given a clock C and an initial configuration $(s_0, v_0)$, an EFSM operates as follows. In the beginning, it resets the configuration: $(s, v) \leftarrow (s_0, v_0)$. On every tick of $C$, it computes the set of enabled transitions: $E \leftarrow \{ t \in T | s_n = s \land \gamma(v) = true \}$. A single transition $t \in E$ (chosen nondeterministically) fires. Executing this transition, the EFSM changes the configuration (updates the context and moves from the initial state of the transition to the final one): $(s, v) \leftarrow (s, v_i)$.

4. EFSM extraction

The proposed approach to EFSM extraction is as follows: (1) HDL code is parsed, and the abstract syntax tree (AST) is built; (2) the AST is traversed, and the intermediate representation (IR) is elaborated: (a) clock inputs (CIs) are identified; (b) implicit state registers (ISRs) are detected and replaced with the explicit state registers (ESRs); (3) the IR is transformed into the set of GAs; (4) data flow analysis of the GAs is performed, and the ESRs are recognized; (5) the conditions on the ESRs are analyzed, and the EFSM states are identified; (6) the EFSM transitions (with guards and actions) are generated. Section 4.1 describes the preprocessing phase (steps 1, 2 and 3). Section 4.2 considers the main phase (steps 4, 5 and 6).

4.1. HDL-to-GAs translation

The purpose of the preprocessing phase is to derive the set of clocked GAs [7] from an HDL description. The GA model is considered as being more abstract design paradigm comparing with the RTL. In contrast to the latter, GAs do not specify control logic (it is constructed by a compiler that takes into account limitations specified by a designer). We need to perform the backward transformation (with a minor reservation that we are not required to get rid of the “superfluous” control logic presented in HDL code).

Step 1 is widely examined; let us start with Step 2. One of the goals of the step is to identify CIs (input signals used to synchronize the design actions). We introduce the following heuristic. A variable $v$ is said to be a clock-like input (CLI) if the following properties are satisfied: (1) $v$ is a 1-bit input; (2) $v$ appears in the sensitivity list of some of the processes (or in a wait statement); (3) $v$ is not used in assignments (neither on the left nor on the right). Step 2 is also responsible for detecting ISRs and replacing them with ESRs. An ISR is understood here as a register that is not encoded explicitly but implied by a designer. We use a method similar to one described in [6]: (1) each process $p$ is associated with a unique ISR $r_p$; every wait statement $w$ (including $p$’s activation statement) is associated with a particular value $v_w$ of $r_p$; (2) all execution paths between all of the wait statements are evaluated; (3) the following transformations are carried out: (a) process $p$ is removed; (b) for each path (or hammock) $\pi$ (between $w_i$ and $w_j$), a new process $p_\pi$ is added: (i) the activation
condition of $p_z$ coincides with $w_i$’s; (ii) $p_z$ has the body if $r_p = v_{wi}$ then $\gamma$; $r_p \leq v_{wi}$ end if.

At Step 3, clocked GAs are extracted. The procedure is straightforward. For each process $p$ generated at Step 2, a set of clocked GAs $\{(C_p^{(i)}, \gamma_p^{(i)} \rightarrow \delta_p^{(i)})\}_{i=1,n}$ is created: (1) $C_p^{(i)}$ contains all CLIs used in $p$; (2) $\gamma_p^{(i)}$ specifies the condition of the $i$th branch of the top-level conditional (or case) statement (if there is no such a statement, then $n = 1$ and $\gamma_p^{(i)} = \text{true}$); (3) $\delta_p^{(i)}$ consists of the statements of the $i$th branch (or of the whole $p$’s body if $n = 1$). Note that after applying the procedure there might be situations when some of the actions $(\delta_p^{(i)})$ contain embedded conditional statements. To simplify further analysis, they are lifted to the guards (taking into account dependencies from the preceding assignments); this leads to extra splitting of the GAs.

Let us illustrate GAs extraction on an example of a counter. The device has three inputs (clock, reset and enable), one output (count) and one register (state). The VHDL code is presented in Table 1 (on the left). Extraction is done as follows: (1) the input clock is identified as a CLI; (2) two processes are recognized: (a) the first one is activated on clock’event; (b) the second one (assignment of state to count) has the trivial activating condition; (3) according to the number of branches in processes, two GAs ($x$ and $y$) are generated for the first process; one GA ($z$) is constructed for the second one (see the right side of Table 1).

<table>
<thead>
<tr>
<th>process (...) begin</th>
<th>if</th>
<th>if</th>
<th>elseif</th>
<th>elseif</th>
<th>end if</th>
<th>count &lt;= state;</th>
<th>end process:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>clock’event</td>
<td>reset = '1' then</td>
<td>state &lt;= '0';</td>
<td>enable = '1' then</td>
<td>state &lt;= state + '1';</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$C_v$</td>
<td>$\gamma_v$</td>
<td>$\delta_v$</td>
<td>$C_v$</td>
<td>$\gamma_v$</td>
<td>$\delta_v$</td>
<td>count = state</td>
</tr>
<tr>
<td></td>
<td>${\text{clock}}$</td>
<td>(reset = 1)</td>
<td>state = 0</td>
<td>${\text{clock}}$</td>
<td>(reset = 1) &amp; (enable = 1)</td>
<td>state = state + 1</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. An HDL-to-GAs translation example

4.2. GAs-to-EFSM translation

A data flow graph (DFG) is an essential way to organize a set of GAs. Let $x$ and $y$ be GAs and $v$ be a variable. They say that $v$ is defined in $x$ (and write $v \in \text{Def}_x$) if $x$’s action contains an assignment to $v$; $v$ is said to be used in $y$ ($v \in \text{Use}_y$) if $v$ appears in $y$’s guard or action but is not modified. A GA $y$ is said to be dependent on $x$ if $\text{Def}_y \cap \text{Use}_y \neq \emptyset$. A DFG is a directed graph that represents dependencies between GAs. In Figure 1, the DFG for the above example is depicted (labels are the defined-used variables).

![Figure 1](image_url)

Figure 1. The DFG for the above example

Step 4 performs data flow analysis of the GAs and identifies the ESRs (the registers that encode the control state). The heuristic is as follows. A variable $v$ is said to be a state-like register (SLR) if the following requirements are met: (1) $v$ is not an input; (2) there is a GA $x$, such that $v \in \text{Use}_x$; (3) there is a GA $y$, such that $v \in \text{Def}_y$; (4) there exists a path in the DFG from $x$ to $y$. Additional constraints include: (5) all GAs that use $v$ have the same non-trivial clocks ($C \neq \emptyset$); (6) for each GA that uses $v$, there should be at least one clock-free execution path to a GA that defines $v$ (usually, $v$ is defined in the same action). Note that synthesis tools stipulate stricter requirements to ESRs. For example, the register count in the above code is an SLR, but it is not synthesized as an ESR (ESRs should be assigned with expressions statically evaluating to constants). For complex designs, we use one more requirement: (7) guards on $v$ should be equivalent to $[v] \in X$, where $X$ is a bounded set of constants (the maximum size of the set may depend on the design complexity).

At Step 5, specific states are identified. The preparatory step is to factorize the SLRs by the dependence relation. Two variables $u$ and $v$ are called dependent if there are two GAs, $x$ and $y$, such that $u \in \text{Use}_x \cup \text{Def}_y$ and $v \in \text{Use}_y \cup \text{Def}_x$, and there exists a clock-free execution path between $u$ and $v$ (possibly of length 0); otherwise, $u$ and $v$ are called independent. Each equivalence class $R$ of the SLRs is processed separately and corresponds to one EFSM. State extraction is done as follows: (1) all constraints (conditions) on $R$’s registers are collected $(\Phi(R))$; (2) the constraints of $\Phi(R)$ are decomposed so as to form the smallest set of disjoint constraints $(\Phi'(R))$; (3) each constraint $\varphi \in \Phi'(R)$ is associated with a symbolic state $s$. Note that the routine is not hard if the constraints have the form $[v] \in X$.

Finally, Step 6 generates transitions and produces the EFSM. Providing that all conditional statements in the GAs’ actions are lifted to the guards (see the note to Step 3), the process is as follows: (1) the GAs are split in such a way that a single guard is consistent with the only state constraint, and guards associated with the same state are either equivalent or incompatible; (2) for each GA $x$, if $\gamma_x$ is consistent with $\varphi_x$, then the proto-transition $t = (s, \gamma_x \rightarrow \delta_x, -)$, whose final state needs to be determined, is saved (proto-transitions having the same states and guards are merged); (2) for each proto-transition $t$, the action $\delta_i$ is symbolically executed,
mapping \( \phi_s \land \gamma_l \) to a new constraint \( \gamma'_l \); (a) if \( \gamma'_l \) coincides with some \( \phi_s \), then \( t \) with \( s' \) as the final state is added to the set of transitions; (b) otherwise, for all \( s' \), such that \( \phi_s \land \gamma'_l \) are consistent: (i) \( \phi_s \land \gamma'_l \) is backward executed, producing the precondition \( \gamma'_l^* \); (ii) the transition \( t = (s, (\gamma_l \land \gamma'^*_l) \rightarrow \delta, s') \) is added to the set of transitions. Figure 2 shows the EFSM for the above example. It is trivial, because there are no conditions on the register state (identified as a SLR).

5. Experiments

We have implemented a tool for EFSM extraction and made some experiments. The tool is written in Java with the help of JUNG [8] and zamiaCAD [9]. It handles VHDL designs described in the synthesizable subset of the language (currently, a hardware design is processed as a single whole not taking into account its hierarchical structure). We have analyzed four open-source VHDL projects: PLASMA, DLX, HC11 and UART (42 modules consisting of 14 KLOC). For 70% of the modules, CLIs have been extracted (the rest modules describe combinational logic). The CLIs include all clock and reset signals and also some signals related to interruptions. SLRs have been detected in 25% of the modules. In about 30% of the cases, a SLR’s name has the “state” substring (moreover, all of the “state” registers have been recognized as SLRs). In other cases, the results are adequate. For example, the tool extracts registers that keep queues’ fullness (when Requirement 7 from the definition of a SLR is not used).

6. Remarks and conclusion

EFSMs are widely used in hardware verification, including simulation-based and formal techniques. A number of EFSM-based methods have been proposed for generating functional tests and checking formally specified properties [1,2,4,5]. We have suggested the method for extracting EFSMs from HDL code. The approach has many similarities with one described in [2], but it also has some distinctions. The main of them is that it automatically determines what registers of a target design represent the EFSM state and extracts states and transitions by analyzing the conditions on the identified registers (in [2], information on state registers is used at the last step of the process). We are planning to compare our approach with the above-mentioned one in the nearest future.

Another thing we are going to do is to study deadlock and conflict detection in networks of EFSMs with shared variables. For example, if an EFSM defines a variable in one transition and uses it in another transition, there should not be other EFSMs that could modify the variable in between the define-use chain of the first EFSM. Improving test generation methods is another interesting option. We think, it is a promising idea to use concolic testing techniques (which combine concrete and symbolic execution) [10] for constructing test sequences for HDL descriptions.

7. References