Generating Finite State Machines from SystemC

Ali Habibi, Haja Moinudeen, and Sofiène Tahar
Department of Electrical and Computer Engineering
Concordia University
1455 de Maisonneuve West
Montréal, Québec, Canada
Email: {habibi,haja,m.tahar}@ece.concordia.ca

Abstract

SystemC is a system level language proposed to raise the abstraction level for embedded systems design and verification. In this paper, we propose to generate Finite State Machines (FSM) from SystemC designs using two algorithms originally proposed for the generation of FSM from Abstract State Machines (ASM). This proposal enables the integration of SystemC with existing tools for test case generation from FSM. Hence, enabling two important applications: (1) using the FSM graph structure to produce test suites allowing functional testing of SystemC designs; and (2) performing conformance testing, where the FSM serves as a precise model of the observable behavior of the system used to validate lower abstraction levels of the design (e.g., Register Transfer Level (RTL)).

1. Introduction

The modeling abstraction in the design flow has been raised to the system level because of the increasing complexity of today’s System-on-Chip (SoC). Until recently, modeling architectures required pin-level hardware descriptions, typically Register Transfer Level (RTL). Besides, designing and verifying the models, and simulation at RTL are tediously slow. The system level is eventually the best solution to overcome these issues [5]. In this regard, several system level languages have been proposed to raise the abstraction level of modeling and verification of today’s SoC. Among them, SystemC [11] is expected to make an adverse effect in the arenas of architecture, co-design and integration of hardware and software. The SystemC library [10] of classes and simulation kernel extend C++ to provide support for concurrent behavior, a notion of time sequential operations, data types for describing hardware, structure hierarchy and simulation.

The verification of SystemC is a bottleneck due to the object-oriented (OO) nature of the language. Two verification techniques could be used: formal verification and simulation. Model checking, as a formal technique, performs a state exploration of the system to verify the implementation of the system against the system properties. In contrast, the second technique, simulation, needs test suites to validate the design functionalities. In order to have efficient test suites for simulation, a finite state machine (FSM) model of the design could be used.

In this paper, we propose to generate FSM from SystemC designs using two algorithms originally proposed for the generation of FSM from Abstract State Machines (ASM) [4] [2]. This FSM generation for SystemC enables several applications such as conformance checking [1] between two design levels. The generated FSM could be also used to define a better coverage evaluation criteria.

We propose two algorithms (called direct and grouping algorithms). Figure 1 describes our methodology of generating FSM from SystemC and the potential applications. The original SystemC code is used to feed the direct algorithm with the entities required for the FSM generation. These entities include: the list of methods, the list of state variables and the list of trigger events. From the algorithm side, these entities correspond, respectively, to actions (A), state variables (V) and action’s pre-conditions (Apre). In addition to these entities, the grouping algorithm requires as input a set of grouping conditions (GC). Both algorithms perform a state exploration procedure in order to discover all possible system’s states starting from a set of initial states.

The rest of this paper is organized as follows: Section 2 discusses some related work to ours. Section 3 describes the algorithm to generate FSM from SystemC. Section 4
presents an application to the PCI bus standard verification to substantiate our approach. Finally, Section 5 concludes the paper.

2. Related Work

In [4], Grieskamp et al. proposed an FSM-generating algorithm from ASM. In [4], given a model in the Abstract State Machine Language (AsmL) [6], the algorithm generates an FSM. The state transitions of the ASM are used to generate a link between hyperstates which is based on single state groupings. The limitation of this algorithm is the use of single state grouping, which is not adequate to solve the state space explosion problem. Campbell et al. [2] extended the work of [4] to reduce the number of states in the generated FSM. The FSM generation algorithm from AsmL (or Spec# [9]) in [2] is based on multiple state groupings which is an extension of the concept of hyperstates. The main advantage of this algorithm is the reduction of state space, of the final FSM, using the notion of multiple state groupings.

Other related work concerns generating FSM from hardware description languages (HDL). For instance, in [3], a translation procedure from Verilog into timed FSM model was proposed. The resulting timed FSM was used to verify the system being modeled using model checkers. Lohse et al. presented in [7] an approach to generate BDD-based FSM from VHDL code. This is performed in two phases: (1) declarations are annotated with BDDs and processes are compiled into control graphs; and (2) the control graphs are then compiled into an FSM and optimization of the FSM was performed. The generated FSM was mainly used as an input for a model checking tool called SMV [8]. However, the approach in both [3] and [7] is restricted to synthesizable subset of Verilog and VHDL constructs respectively. The work of Vikram et al. [12] was more specific to SystemC, where the design (in SystemC) is first translated to C, then, the FSM is generated from the C code. This approach is problematic in the sense that translating SystemC to C is not always feasible. All the previously mentioned techniques could not be applied to SystemC considering the OO nature of the library and that not all SystemC is synthesizable.

3. FSM Generation Algorithms

In this section, we present two algorithms that generate FSM from SystemC code. The two algorithms have been originally proposed for ASM based languages in [2] and [4]. The FSM generation algorithms need the following entities:

- State Variables ($V$)
- State Space ($S$)
- Initial States ($S_{init}$)
- Actions ($A$)
- Transition Relation ($R$)

$V$ is a set of state variables and for each $v_i$ in $V$, there is a corresponding domain $d_{v_i}$ in $D$, where $D$ is a set of domains for every type of state variables. $S = \{s_1, s_2, s_3, \ldots s_n\}$ is a total state space of the design being modelled where each state $s_i$ in $S$ is an instantaneous description of the system that captures the value of state variables ($V$) at particular instant of time. $S_{init} \subseteq S$ is a set of initial states from where the state exploration starts. $A = \{a_1, a_2, a_3, \ldots a_n\}$ is a set of actions in the model defined as follows.

**Definition 3.1** Action ($a_i$)

$a_i$ is a four-tuple $(a_M, a_{Pre}, a_{Post}, a_{Cst})$ where,

- $a_M$ is a method
- $a_{Pre}$ is a set of pre-conditions
- $a_{Post}$ is a set of post-conditions
- $a_{Cst}$ is a set of constraints

$a_{Pre}$ is a set of Boolean propositions that have to be true in the beginning of an action $a_i \in A$ execution, in a state $s$. $a_{Post}$ is also a set of Boolean propositions that must be true at the end of an action $a_i \in A$ execution and $a_{Cst}$ is a set of Boolean propositions that must to be true at certain part of an action $a_i \in A$ execution.

Next, we define the transition, $R$, from one state to another during the action execution and it is defined as follows:

![Figure 1. Methodology and Motivation.](image-url)
Definition 3.2 Transition Relation (R)
Let S be a set of states and A be a set of actions then R is defined as
\[ R : S \times A \rightarrow S \]
\[ (s_{\text{current}}, a) \mapsto s_{\text{next}} \]

During the exploration phase of the algorithm, relevant states are stored in \( S_{\text{FSM}} \). Starting from \( S_{\text{init}} \), new discovered states are added to \( S_{\text{FSM}} \) together with the new transition \( T \). \( T = \{t_1, t_2, t_3, \ldots t_n\} \) is a set of transitions included in the FSM where \( t_i \) is a three-tuple, \( (s_{\text{current}}, a, s_{\text{next}}) \).

3.1. Helper Functions

Some helper functions are needed for the FSM generation algorithms. They include \( \text{enabled()} \) and \( \text{nonExp()} \). \( \text{enabled()} \) is used to know for a specific state \( s \) if an action \( a \in A \) is enabled and \( \text{nonExp()} \) is used to know if a state \( s \) is fully explored.

Definition 3.3 \( \text{enabled()} \)
Let \( S \) be a set of states and \( A \) be a set of actions, then \( S \times A \rightarrow \{\text{true}, \text{false}\} \)
\[ (s, a) \mapsto \text{enabled}(s, a) \]
where:
\[ \text{enabled}(s, a) = \begin{cases} \text{True}, & (a_{\text{Pre}} = \text{True}) \\ \text{False}, & (a_{\text{Pre}} = \text{False}) \end{cases} \]

Definition 3.4 \( \text{nonExp()} \)
Let \( S \) be a set of states and \( T \) be a set of transitions \( \text{nonExp}(s) : S \rightarrow T \times T \times \ldots \times T \)
s \mapsto \{ (t_1, t_2, \ldots t_n) \mid ((t_i = \langle s, a, R(s, a) \rangle) \land (\text{enabled}(s, a) = \text{false})) \}

Definition 3.5 Frontier (F)
Let \( S \) be a set of states and \( A \) be a set of actions, then the Frontier \( F = \{s \mid s \in S \mid \exists a \text{ enabled}(s, a) = \text{true}\} \) is a set of states that have not yet been fully explored during the FSM generation process. The exploration of the state space stops, if \( F \) is empty. Initially, \( F \) contains \( S_{\text{init}} \).

3.2. Direct FSM Generation Algorithm

In the algorithm presented in Figure 2, the states that are discovered and are going to be part of the generated FSM are stored in \( S_{\text{FSM}} \). The transition from a current state to a new state is stored in \( T \). The algorithm starts exploring the states if the Frontier \( (F) \) is not empty (line 4). In the beginning, \( F \) contains \( S_{\text{init}} \). For each action \( a_i \in A \), the algorithm checks if an \( a_i \) is enabled in the current state \( (\text{current}) \) using a helper function \( \text{enabled()} \) (line 7). Thereafter, the new state \( (\text{next}) \) is discovered using the transition relation \( R \) (line 8). If the new discovered state \( (\text{next}) \) is not in \( S_{\text{FSM}} \), it is added to \( S_{\text{FSM}} \) and the transition \( t \) (current,a,next) is added to \( T \) (lines 10 & 11). The new state next is also added to \( F \) if there exists an action enabled in this new state. If \( next \) is already in \( S_{\text{FSM}} \), \( t \) is still added to \( T \) if it does not exist in it. The algorithm terminates once \( F \) becomes empty. The output of the algorithm is an FSM which is a tuple \( ( S_{\text{FSM}}, T ) \).

```
1: \( S_{\text{FSM}} = \{ S_{\text{init}} \} \)
2: \( F = \{ S_{\text{init}} \} \)
3: \( T = \{ \} \)
4: while \( (F \neq \{ \}) \) {
5:     \( current := F.\text{Head} \)
6:     foreach \( a \in A \{
7:         if (\text{exists } (\text{enabled}(current,a)) \}
8:             \text{next} := R(current,a)
9:         if (\text{next} \notin S_{\text{FSM}}) \{
10:             \( S_{\text{FSM}} := S_{\text{FSM}} \cup \{ \text{next} \} \)
11:             \( T := T \cup \{ (current,a,\text{next}) \} \)
12:         \}
13:         \text{if (exists } a \text{ in } A \text{ where enabled(next,a)} \}
14:             \text{true} \} \( F := F \cup \{ \text{next} \} \)
15:     \}
16:     \text{elseif } (\text{current,a,next} \notin T) \{
17:         \( T := T \cup \{ (current,a,next) \} \)
18:     \}
19: 
20: \( F := F.\text{Tail} \)
21: }
```

Figure 2. Direct FSM Generation Algorithm.

3.3. Grouping FSM Generation Algorithm

State groupings [2] is a technique for controlling scenarios by selecting representative states with respect to an equivalent classes. Generally, the state space is very large and it is indispensable to prune it as much as possible. In Figure 3, we present the algorithm that we adapted for SystemC from [2] to generate a grouped FSM. Figure 3 shares some parts of the direct FSM algorithm but the vital difference is the incorporation of multiple state groupings. It uses a state-based grouping condition to group the states. The grouping condition \( G_C \) is a Boolean proposition that uses state variables \( (V) \) and functions defined in the design. It is defined as follows.

Definition 3.6 Grouping Condition \( G_C \)
\[ G_C = f(v_1, v_2, v_3, \ldots v_n) \mid v_i \in V \]
Having defined $G_C$, we need to define the grouped state $s_g$, which is a set of states that are equivalent under one grouping condition $G_C$, formally defined as follows.

**Definition 3.7** Grouped State ($s_g$)

$$s_g = \{ s \mid s \in S \mid G_C = true \}$$

Let $S_G$ be a grouped states, i.e., a set of grouped states $s_g$. In order to group the states, we need to have a way to map the states $s$ to grouped state $s_g$. This is achieved by a grouping function $g$. $g$ maps states of the model to concrete values defined by a state-dependent grouping condition ($G_C$). The value $g(s)$ is called the $g$-label of $s$. Two states are $g$-equivalent if they have the same $g$-label. $g$ is defined as follows.

**Definition 3.8** Grouping Function ($g$)

$$g : S \rightarrow S_G$$

$$s \mapsto s_g$$

Now, we define a new transition relation $R_g$ that provides the transition of one grouped state to another during the action execution. Unlike $R$ where it deals with individual state $s$, $R_g$ deals with grouped state $s_g$ and is defined as follows.

**Definition 3.9** Transition Relation in Grouped FSM ($R_g$)

$$R_g : S_G \times A \rightarrow S_G$$

$$(s_g^{current}, a) \mapsto s_g^{next}$$

We have defined all the needed components for the grouping FSM generation algorithm. The output of the grouping algorithm will be $FSM_G$ that has two entities viz., $S^G_{FSM}$ and $T_G$. $S^G_{FSM}$ is a set of grouped states that have been discovered and $T_G$ is a set of transitions among the grouped states in the generated FSM. They are formally defined below.

**Definition 3.10** Grouped FSM States ($S^G_{FSM}$)

$$S^G_{FSM} = \{ s \mid s \in S_{FSM} \land \exists g_i \in g \mid g_i(s) = true \}$$

**Definition 3.11** Grouped FSM Transitions ($T_G$)

$$T = \{ (s_g^{current}, a, s_g^{next}) \} = \{ (s_g^{current}, a, R_g(s_g, a)) \}$$

The grouping algorithm in Figure 3 starts with $F$ containing $S_{init}$ and then explores all the possible actions in $A$ that can be enabled in the current state ($current$) (line 8). Then, the new state ($next$) is discovered using the transition relation $R$ (line 9). The new discovered state is mapped to a grouped state based on the grouping function $g$. If the new grouped state ($g(next)$) is not in $S^G_{FSM}$, it is added to it and the transition ($current, a, g(next)$) is added to $T_G$. The new state next is also added to $F$ for further exploration. Finally, the algorithm terminates once $F$ becomes empty.

1: \( S^G_{FSM} = \{ \emptyset \} \)
2: \( F = \{ S_{init} \} \)
3: \( T_G = \{ \emptyset \} \)
4: \( g = \{ g_1, g_2 \ldots g_k \} \)
5: \( while(F \neq \emptyset) \{ \)
6: \( current := F_{Head} \)
7: \( foreach a \in A \{ \)
8: \( if (enabled(a, current)) \{ \)
9: \( next := R(current, a) \)
10: \( if (g(next) \notin S^G_{FSM}) \{ \)
11: \( S^G_{FSM} := S^G_{FSM} \cup \{ g(next) \} \)
12: \( T_G := T_G \cup \{ (current, a, g(next)) \} \)
13: \( if (exists a in A where enabled(next, a) \)
14: \( = true) \{ F := F \cup \{ next \} \)
15: \( \} \)
16: \( } \)
17: \( elseif ((current, a, next) \notin T) \{ \)
18: \( T_G := T_G \cup \{ (current, a, g(next)) \} \)
19: \( \} \)
20: \( } \}
21: \( F := F_{Tail} \)
22: \)

**Figure 3.** Grouping FSM Generation Algorithm.

### 4. Application: PCI Bus Standard Verification

The PCI [13] bus standard boasts a 32-bit data path, 33MHz clock speed and a maximum data transfer rate of 132MB/sec. A 64-bit specification exists for future PCI designs, which will double data transfer performance to 264MB/sec. In Figure 4, we show a generic structure of the PCI bus with a single master and a slave. We added also an external monitor module that will be used to track the signals at the input and output ports of the bus in order to validate the good functioning of the bus. Each PCI master has a pair of arbitration lines that connect it directly to the PCI bus arbiter. When a master requires the use of the PCI bus, it asserts its device specific REQ# line to the arbiter. When the arbiter has determined that the requesting master should be granted control of the PCI bus, it asserts the GNTH# (grant) line specific to the requesting master. In the PCI environment, bus arbitration can take place while another master is still in control of the bus.

In the PCI terminology, data is transferred between an initiator, which is the bus master, and a target, which is the bus slave. The initiator, drives the C/BE[3:0]# signals (Figure 4) during the address phase to signal the type of trans-
Figure 4. PCI Bus Structure.

To evaluate the performances of both algorithms we consider as criteria: the CPU time needed for generating the FSM and the number of its states and transitions. The grouping condition (see Figure 5) is a conjunction of the destination slave and the final status of the transmission (completed or stopped). The procedure GroupingCondition() returns an integer value in {0, 1, 2, 3} or -1 when an error happens. Each of these values identifies a grouped state. Therefore, considering the definition of GroupingCondition(), the maximum number of grouped states is four. The variable MASTERS in Figure 5 refers to the set of the masters connected to the PCI bus. The integer data member m_dest identifies for the owner’s object (master) the target slave for the current transaction. Finally, the Boolean data member m_stop specifies if the transaction can be stopped by the slave before it is fully completed.

<table>
<thead>
<tr>
<th>Number of Masters</th>
<th>Number of Slaves</th>
<th>CPU Time (s)</th>
<th>Number of FSM Nodes</th>
<th>Number of FSM Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.34</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0.80</td>
<td>58</td>
<td>85</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4.44</td>
<td>236</td>
<td>341</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4.31</td>
<td>293</td>
<td>449</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>108.03</td>
<td>1881</td>
<td>3153</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>727.01</td>
<td>3880</td>
<td>7542</td>
</tr>
</tbody>
</table>

Table 1. FSM Generation: Direct Algorithm.

Tables 1 and 2 show the experimental results for multiple numbers of masters and slaves in the cases of the direct and grouping algorithms, respectively. For both, the numbers of states and transitions increase exponentially as a function of the number of masters and slaves connected to the bus. However, we can note that the CPU time required for the FSM generation using the direct algorithm is shorter than the one required for the grouping algorithm. Finally, as expected the grouped FSM is smaller than the general FSM in terms of number of states and transitions.

<table>
<thead>
<tr>
<th>Number of Masters</th>
<th>Number of Slaves</th>
<th>CPU Time (s)</th>
<th>Number of FSM Nodes</th>
<th>Number of FSM Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.36</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0.86</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4.56</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4.61</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>148.47</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>772.60</td>
<td>6</td>
<td>23</td>
</tr>
</tbody>
</table>

Table 2. FSM Generation: Grouping Algorithm.

Figure 6 gives a snapshot of the grouped FSM for the case of two masters and two slaves for the grouped condition is the one described in Figure 5. For example, the group 0 (the initial state G0 in Figure 6) corresponds to the case when the destination slave is Slave1.

---

1 Simulation platform: Pentium IV/1GB memory/WinXP-SP2.
\( m_{dest} = 1 \) and the transaction succeeds \( m_{stop} = 1 \).

\[
\text{GroupingCondition()} = \\
\text{if (} \exists x \text{ in } MASTERS \\
\text{ where } x.m_{dest} = 1 \text{ and } x.m_{stop} = true) = true) \\
\text{then Return 0} \\
\text{else} \\
\text{if (} \exists x \text{ in } MASTERS \\
\text{ where } x.m_{dest} = 1 \text{ and } x.m_{stop} = false) = true) \\
\text{then Return 1} \\
\text{else} \\
\text{if (} \exists x \text{ in } MASTERS \\
\text{ where } x.m_{dest} = 2 \text{ and } x.m_{stop} = true) = true) \\
\text{then Return 2} \\
\text{else} \\
\text{if (} \exists x \text{ in } MASTERS \\
\text{ where } x.m_{dest} = 2 \text{ and } x.m_{stop} = false) = true) \\
\text{then Return 3} \\
\text{else Return -1}
\]

Figure 5. Grouping Condition.

Figure 6. Generated FSM using the Grouping Algorithm: 2 Masters / 2 Slaves.

5. Conclusion

We presented, in this paper, two algorithms for the generation of FSM from SystemC. In the first algorithm (called direct algorithm) the state exploration procedure searches for all relevant states. In the second algorithm (called grouping algorithm), in addition to the

default state exploration performed by the direct algorithm, we add a set of grouping conditions to help reducing the final FSM size. Grouping conditions act like conditions to classify states into groups. The generated FSM can be of great importance for several applications such as model checking, guided test generation, coverage evaluation and conformance testing.

We illustrated the feasibility of our approach on a PCI bus standard of the SystemC model. We showed that the FSM generation time for both algorithms (direct and grouping) is relatively short. The grouping algorithm requires more CPU time but outputs smaller FSM than the direct algorithm. Our future work includes using the generated FSM in generating guided test sequences and performing conformance checking of low abstraction level SystemC designs.

References


