Electronic Notes in Theoretical Computer Science 65 No. 5 (2002)
URL: `http://www.elsevier.nl/locate/entcs/volume65.html` LastPage pages

Slicing Synchronous Reactive Programs

Vinod Ganapathy [1] S. Ramesh [2]

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai, India

Abstract

This paper extends the well-known technique of slicing to synchronous reactive programs. Synchronous languages exemplified by Esterel, Lustre, Signal and Argos, employ a novel model of execution that is found useful for abstract and high level description of complex controllers.

Slicing is well known in the domain of sequential transformational programs and has been found to be useful in analysis, debugging and verification. The classical definition of slicing is inadequate for reactive programs. In this paper, we propose a new definition of slicing for reactive programs. An algorithm for computing slices based upon this definition is developed. We have taken the Argos language for concrete description of our ideas; they are of general applicability and can be applied easily to other synchronous languages.

1 Introduction

Synchronous languages exemplified by Esterel, Lustre, Signal, Argos and to some extent Statecharts [1,5,11,6], have been proposed for programming complex control programs found in many embedded applications. These languages employ a novel model of concurrency that enables simple and abstract high level specification of reactive systems. This model has a number of useful properties: the notion of concurrency is logical with no run-time threads and consequent overheads and uncertainty in timing computation; interaction among concurrent components and environment are through instantaneous broadcast signals which provides a very convenient abstraction; unlike other models concurrency does not give rise to nondeterminism.

An important phase in the design of reactive systems is their verification and analysis as the application domain requires very high quality systems.

[1] Email: `vg@cs.wisc.edu`
[2] Email: `ramesh@cse.iitb.ac.in`

Thorough analysis, in general, is a very costly procedure, because of the state explosion problem. One very common technique used to overcome this problem is compositional verification [10,14,4]: decompose the given large system into manageable smaller subsystems that can be verified independently. Decomposition can be achieved either by means of their structure or functionality. *Slicing* [19,17] is a functional decomposition technique that has been successfully used in verification and analysis of sequential software: to analyse a large program, slice the program into functionally independent units that can be analysed with relative ease.

This paper describes an attempt to extend the technique of slicing to reactive languages. Standard notion of slicing is defined for transformational sequential programs which is aimed to preserve the value of a chosen set of variables at a specific control point (called the slicing criterion). This is not very suitable for reactive programs as these are supposed to exhibit an infinite ongoing behavior with its environment rather than terminate producing a particular value. We have defined a new notion of slicing that is more natural for reactive programs. Based upon this notion, we have developed an algorithm for slicing.

For concreteness, our ideas have been illustrated using Argos programs, though it is of general applicability to other reactive languages. Argos [11] is based upon Finite State Machines, a well known formalism for describing the behaviour of controllers for many embedded applications. Current day embedded systems are so complex that flat and unstructured state machine descriptions of them will be very huge and difficult to debug and analyze. Argos, inspired by Statecharts [6,7], uses two main features, hierarchical states and concurrent state machines for structuring large descriptions.

There are a few attempts to extend slicing techniques to concurrent programs [9,13,2,12]. But all these approaches, like those for sequential programs, are not appropriate for reactive programs.

The organization of this paper is as follows: In Section 2 we briefly describe Argos constructs. In Section 3, we extend the notion of a slice to Argos, and give a slicing algorithm. The correctness and preciseness of the slice computed by our algorithm are also discussed. The paper concludes with Section 4.

2 Argos

Argos is a graphical language, based on statecharts [11] with a simpler syntax, fewer features and cleaner synchronous semantics. The basic components of Argos are automata. Automata are Mealy machines having transitions labelled with input and output signals. Three operators are used for enriching Mealy machines with parallel and hierarchic structures: refinement, parallel composition and encapsulation. Using refinement, a state in an automaton can be refined to contain another automaton, or in general, an arbitrary machine; two machines can be run concurrently using parallel composition; encapsulation of a machine with respect to a signal makes the signal local to the machine.

Figure 1 shows an Argos program that models a 3-bit counter [11]. The counter counts from 0 up to 7. It can be initialised to 0 and can be reset during counting.

The Argos description is, at the top level, a two state machine (`counting` and `not_counting`) in which the signal `end` is made local. `not_counting` is the initial state, denoted by the 'half-arrow'. The transition to `counting` takes place when the signal `init` is given.

The state `counting` is refined to contain concurrent composition of three machines, each representing one of the three bits. Each of these machines has two states corresponding to the two values that a bit can take. Initially all the three bit values are zero indicated by the initial states A0, B0, C0. External signal `tick` changes the state of the right most machine (representing the least significant bit). When this machine is in state A1, the external signal `tick` results in the generation of the local signal `lt1` which simultaneously triggers the state transition of the next machine from B0 to B1; the first machine changes its state to A0. The next `tick` signal results in the transition to A1 of the first machine. In this state, the next `tick` will generate `lt1`, triggering the transition from B1 to B0. This will generate another local signal `lt2` which will trigger the transition from C0 to C1 on the left most machine.

When all the 'bits are set to 1', the `end` signal is generated which triggers the transition from `counting` to `not_counting`. At any stage of counting, the `stop` signal can effect the transition to `not_counting` state. The signals `lt1,lt2,end` are local as indicated by the decoration of states in the diagram.

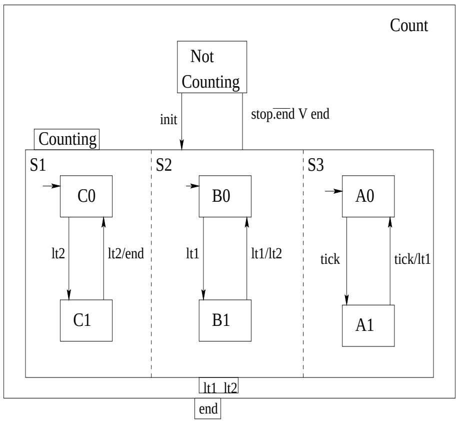

Fig. 1. Argos - an example

As this example illustrates, the operators of Argos allow hierarchical and concurrent description of complex state machines. It has been shown in [3,15] that Statecharts can be translated to efficient synchronous hardware circuits.

3 Slicing Argos programs

Program slicing, in the context of sequential programs, is defined with respect to a pair: a control point and a variable in the program; this pair is called the *slicing criterion* [19]. The slice with respect to a criterion is defined to be a program obtained from the original program by removing certain statements such that if the original program during its execution, reaches the criterion control point with a particular value of the criterion variable then the slice also reaches the same control point with the same value for the criterion variable.

This notion is not a very suitable notion for reactive programs: the behaviour of a reactive program is *not* to produce desired result at the end of execution but to maintain certain ongoing relationship with its environment. More precisely, the behavior of a reactive program is a set of I/O sequences. Also, signals and events are more predominant than variables.

We now define a new notion of slicing that, in our opinion, is more natural for reactive programs. As mentioned in the introduction, for concreteness sake, we define slice with respect to Argos programs, though it is of general applicability to any reactive programming language.

3.1 Definition of a Slice

Definition 3.1 *Slicing Criterion:* A slicing criterion for an Argos program is $< S, b >$, where S is the name of a state and b is an output signal in the program.

The choice of the slicing criterion was motivated by considering situations where taking a program slice would be useful. Program slicing is typically used for program understanding and debugging. In the case of reactive programs, the signals are the variables of interest to the end-user; this justifies the choice of an output signal in the slicing criterion. State information can be used to furthur prune out unwanted information from a slice; where the state can be any state in the Argos program. This is typically useful in situations where the end-user knows that the machine resides in a particular state, and is interested in studying the properties of the program in that state. In such a case, the information returned in the slice will include all, and only that information, that is required to study the behaviour of the program in that state.

Definition 3.2 *Slice:* Given a state S and an output signal b in an Argos program M, the slice w.r.t the slicing criterion $< S, b >$is a machine M_s such that :

(i) M_s is obtained by removing zero or more states (and transitions) from M.

(ii) The behaviour of M_s up to state S is the same as the behaviour of M up to state S as far as the signal b is concerned.

To define a slice more formally, let I be the set of signals that can be given as inputs to the machine M, and $\mathcal{P}(I)$, $\mathcal{P}(I)^*$ denote the power set of I, and the set of all finite sequences of subsets of I. Let $\sigma \in \mathcal{P}(I)^*$ be an arbitrary sequence. Given a machine M, we can view it as a function that takes σ and produces a sequence of output signals. Let us denote the output sequence produced by a machine M on input σ to be $M[\sigma]$. Now we can formally define a slice: M_s is slice of M w.r.t $< S, b >$ iff it is obtained by removing zero or more states from M and

$$\forall \sigma \in \mathcal{P}(I)^* : Reside(M, \sigma, S) \rightarrow (Reside(M_s, \sigma, S) \wedge (M[\sigma]/b) = (M_s[\sigma]/b))$$

where $Reside(M, \sigma, S)$ is a predicate that is true whenever M resides in S after the input sequence σ is fed to it, and $M[\sigma]/b$ means that the output sequence is restricted to only the signal b. Note that the definition does not say anything about the slice if M does not reside in S after accepting σ.

As an example, consider the program shown in figure 2. The second program, in the figure, is the slice of the first program with respect to $< S, b >$: Clearly, the second program has the same behaviour as the first one as far signal b is concerned.

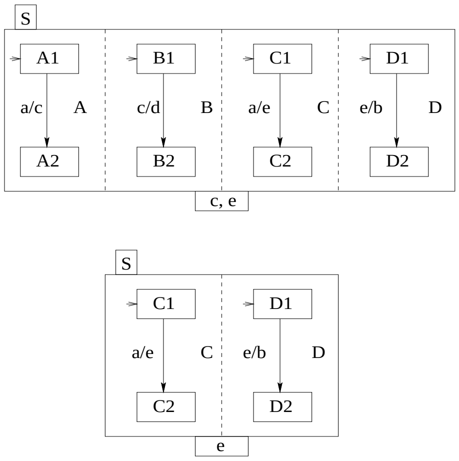

Fig. 2. Two Argos Programs

In the rest of this paper, we will give an algorithm for computing slices.

3.2 Some Definitions

The slicing algorithm is essentially a traversal algorithm that works on a graph corresponding to the machine being sliced. Given a machine M, the graph corresponding to M consists of nodes corresponding to each state of the machine. It has three kinds of edges:

- For every transition in M, there is a **transition edge** from the node corresponding to the source state of the transition to the node corresponding to the target state of the transition.

- There is a **hierarchy edge** from a node A to a node B, if the state corresponding to A contains at its next level, the state corresponding to B. In this case, A is called the parent of B and B, the child of A.
- A **Trigger Edge** from every transition to another transition that is triggered by the former. Consider the program given below. During execution, the transition from A1 to A2, when taken would trigger the transition from B1 to B2 to be taken. This fact is represented by a *trigger* edge from the triggering transition to the triggered transition, as shown by the dotted arrow in Figure 3. In the example the transition from B1 to B2 is triggered because of the local signal b generated while the other transition is taken. In general, a transition may be triggered by more than one local signal. This happens when there is a conjunction or disjunction of local signals on the input side of the triggered transition. In such cases, a separate trigger edge is drawn from each transition that generates any of the signals.

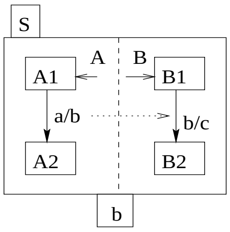

Fig. 3. Trigger Edges.

Related to trigger edges are a few more definitions :

- **source, target** of a trigger edge : The source and target of a trigger edge are respectively the source and target transitions of the edge.
- **source, target state** of a trigger edge : The parent state of the two states connected by the source transition of a trigger edge is called the source state of the trigger edge. Similarly, the target state of a trigger edge is the parent state of the two states connected by the target transition of the trigger edge.

  In the example of figure 3, state A is the source state, and B is the target state of the trigger edge.
- **highest-possible-source-state** of a trigger edge is the earliest ancestor, in the hierarchy defined by the program, of the source state of the trigger edge such that it is not the ancestor of the target state of the trigger edge; ancestor of a state is either itself or an ancestor of its parent.

  For example, in Figure 3, the highest-possible-source state of the trigger edge is A, since A is the earliest ancestor of A which is not an ancestor of B.
- **local-signal** of a trigger edge is the local signal that triggered the target transition, e.g.. in figure 3, the local signal of the trigger edge is the signal b.

We will use the following notations in the algorithm:

- For a transition T, tr-parent(T) denotes the parent state of the two states connected by T, output(T) denotes the set of local signals emitted when T is taken, input(T) denotes the set of local signals needed to trigger T, and states(T) returns the set of two states connected by T.
- For machine M, $root(M)$ denotes its root or the topmost state.
- Given a state S in the machine, S.childlist is the set of all its children, S.parent is the parent state of S, and S.sibling is a set of states sharing the same parent as S.

3.3 Drawing Trigger Edges

In Algorithm 1, we describe our method of drawing trigger edges. Before we run the algorithm, we must ensure that all local signals have a unique name. This could easily be done by renaming the local signals.

Algorithm 1 Algorithm to draw Trigger Edges

for all (transitions M, N | (tr-parent(M) $\neq$ tr-parent(N)) $\wedge$ ($\exists$b | b$\in$output(M) $\wedge$ b$\in$input(N)) **do**
  Draw a Trigger Edge from M to N
end for

This algorithm draws a number of redundant edges. It can be improved by constructing a global state machine to prune out unreachable states. But this would, in general involve an exponential blow-up in the number of states.

3.4 The Slicing Algorithm for Argos

We now present the proposed algorithm that takes a machine M and a criterion $< S, b >$ and produces the slice w.r.t. $< S, b >$. M is given as a graph with all three types of edges being drawn.

The algorithm is essentially a traversal algorithm that traverses along various edges in M, including into the slice all the states and transitions encountered during the traversal. The algorithm starts from the state S, traverses down the hierarchy edges to include all and only those states inside S that might preserve the behaviour of M w.r.t. the signal b. Then it traverses up the transitions edges, in the reverse direction, including all the states encountered. When all the required states at a level are traversed, the traversal proceeds along the hierarchy edges in the reverse direction to include states at higher and higher hierarchy levels. For every state reached, traversal down the hierarchy edges are also carried out to include all the required states inside the state. The traversal eventually reaches the topmost level, when the algorithm proceeds traversing along the trigger edges in the reverse direction to include all states that are concurrent to the states already in the slice.

The main algorithm computing the slice is Algorithm 2. Traversal down the hierarchy edges inside a state is performed by the routine *Top-to-Bottom* slice, while traversal up the transition edges is implemented by the routine *Slice-this-Level*; *Trigger-Edge-Slice* traverses up the trigger edges.

For the sake of simplicity, the algorithm computes the slice as a set of states. The machine corresponding to this set can be constructed easily using the transition and hierarchy edges and the details of hidden signals given in M.

Algorithm 2 The Main Slicing Procedure : Slice (M, $< S, b >$)

```
slice := φ
Call Top-to-Bottom (< S, b >)
while (S ≠ root(M)) do
  Call Slice-This-Level (< S, b >)
  S := S.parent
end while
slice := slice ∪ { root(M) }
Call Trigger-Edge-Slice(M)
Return slice
```

3.4.1 Auxiliary Procedure Top-To-Bottom

This procedure takes a machine whose root state is the state mentioned in the slicing criterion. It checks if the machine has the signal b inside it. If it does not have the signal, then it returns the machine without including the inner states. If however, the machine has the signal b, then all the child states are included in the slice and the same procedure is recursively called on each of the children.

Algorithm 3 Top-To-Bottom ($< S, b >$)

```
slice := slice ∪ { S }
if has(< S, b >) then
  for all (child ∈ S.childlist) do
    Call Top-To-Bottom (< child, b >)
  end for
end if
```

3.4.2 Auxiliary Procedure Slice-This-Level

In this procedure, a reachability analysis is performed to include all those states from which the state in the criterion is reachable. Since the machine may enter these states before it enters the state mentioned in the slicing criterion, Top-To-Bottom slicing is done on all the states included in all these states.

3.4.3 Auxiliary Procedure Trigger-Edge-Slice

This procedure adds those transitions which may affect the emission of the signal b. In algorithm 5, all those transitions which directly affect the emission

Algorithm 4 Slice-This-Level ($< S, b >$)

S.wl := ϕ
if (S.parent == "AND-state") **then**
  S.wl = S.sibling
else
  S.wl := S.wl ∪ {ch ∈ (S.sibling ∪ S) |(ch → S) ∨ (ch → K), for K∈S.wl)}
end if
for all (ch ∈ S.wl) **do**
  Call Top-To-Bottom (< ch, b >)
end for
slice := slice ∪ { S }

of the signal b (mentioned in the slicing criterion) are included in the slice. Then all those transitions which could affect the emission of the signals added in the previous step are considered. The procedure terminates when no more new transitions are added.

Algorithm 5 Trigger-Edge-Slice(M)

list-trigger := {t | t is a Trigger Edge}
while (∃(t ∈list-trigger) | (states(target(t)) ⊆ slice) ∧ ¬(states(source(t)) ⊆ slice)) **do**
  Call Top-To-Bottom(<highest-possible-source-state(t),local-signal(t)>)
end while

3.4.4 An Example

Figure 4 shows an example of an Argos program and its slice as produced by our algorithm. The slicing criterion is $< D, b >$. The algorithm would first include all the children of D in the slice since b occurs in D. It then includes S since D is reachable from S. The transition from S to D is also included in the slice. But this transition is triggered by m that is generated when the transition from $B1$ to $B2$ is taken. There is a trigger edge from this to the other transition and hence the *Trigger-Edge-Slice* procedure would include both $B1$ to $B2$ and the transition between them. The state C is not included in the slice because the D is not reachable from the C.

3.5 Correctness and Precision

Suppose M_s is the output produced by the algorithm, given M and $< S, b >$. Then the following theorem states the correctness of the algorithm:

Theorem 3.3 *For any σ such that $Reside(M, \sigma, S)$, $M[\sigma]$ has b at any step if and only if $M_s[\sigma]$ also has b at the same step.*

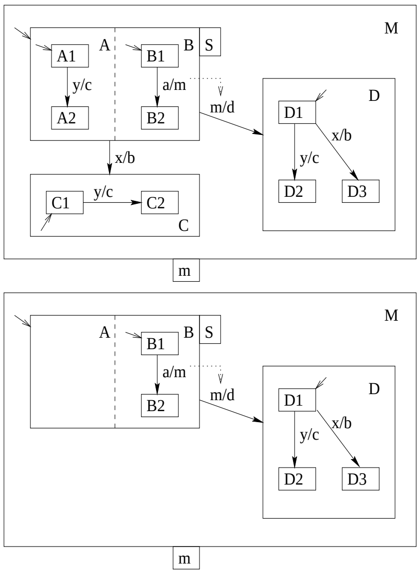

Fig. 4. An Example

For want of space, we give the proof of this theorem for the case that S is the root state; the proof is in Appendix.

The above theorem states that the algorithm produces correct slices. But it does not produce precise slices. Consider the example shown in Figure 2. Our algorithm produces as slice not the machine given in the bottom of the figure but the one that has both the state A and B (the inner states of A, B are, of course, not included). However, using some simple post-processing these extra states can be gotten rid of.

More complex post-processing may have to be done to eliminate more redundant states. For example, in figure 5, in the slice of X with respect to $< X, b >$ it would suffice if we just included the states X, A and B in the slice, whereas our algorithm includes all the states. However if X were part of a bigger machine where it was possible to visit X more than once, then it becomes necessary for us to include in the slice all the children of X.

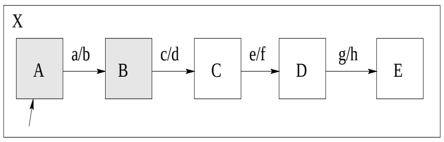

Fig. 5. The slice is not precise

3.6 Complexity

The complexity is expressed in terms of the parameters of the graph G corresponding to the machine M. Let G have n nodes, and t trigger edges.

The recursive procedure *Top-to-Bottom* takes $O(n)$ time. Trigger-Edge-Slice in the worst case could call *Top-to-Bottom* for every trigger edge in G.

Thus the complexity of the algorithm is O(tn). This however, is the worst case running time, and running times are much lower in practice.

3.7 Experimental Results

The slicing algorithm has been implemented, and tested on a number of examples, including the ones given in the paper and the digital watch example given in [5]. Since there are no benchmark Argos programs, the performance of the algorithm was studied on a number of randomly generated programs, with large number of states and trigger edges. In each case the slicing criterion also was chosen randomly. The running time of the algorithm was found to be very insignificant for the chosen samples. Also, the size of the slices produced were all small compared to the original program

The following table describes the performance of an implementation on a Pentium III machine with 64MB RAM.

Sl. no.	States	Trigger Edges	N_s/N	Avg. CPU Util.
1	100	1	0.56	69%
2	500	13	0.15	25%
3	1000	39	0.15	34%
4	2000	79	0.14	15%

where N, N_s are the number of states of the original programs and the corresponding slices repectively.

4 Conclusions

In this paper, we have introduced a new definition of slicing that is more natural for reactive programs. Based upon the definition, an algorithm for computing slices are given. The correctness of the algorithm has also been proved. The proposed algorithm is shown to be efficient in practice. The proposed idea is illustrated here on Argos programs though it is of general applicability. This has been demonstrated in [18], where slicing algorithms for Esterel and Communicating Reactive State Machines (CRSMs)[16] have been developed; CRSMs extend Argos by allowing asynchronous CSP-like communication between machines.

The proposed approach can also be used for slicing VHDL and Verilog programs by extracting state machine descriptions from these programs.

References

[1] G. Berry and G. Gonthier, The Esterel Synchronous Programming Language: Design, Semantics and Applications, Science of Computer Programming, 19(2),

1992.

[2] E. M. Clarke, et al., Program Slicing of Hardware Description Languages, Proc. of Charme'99, 1999.

[3] D. Drusinsky and D. Harel, Using Statecharts for Hardware Description and Synthesis, IEEE TCAD, Vol. 8 (7), 1989.

[4] O. Grumberg and D. E. Long, Model Checking and Modular Verification. In *Proc. of Concur '91*, Vol 527, LNCS, Springer 1991.

[5] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, 1993

[6] D. Harel, Statecharts : A Visual Approach to Complex Systems. In *Science of Computer Programming*, pp 231-274, 8(3), 1987.

[7] D. Harel and A. Naamad, The `STATEMATE` Semantics of Statecharts. In *ACM Transactions on Software Engineering and Methodology*, 5(4), pp 293-333, October 1996.

[8] G. J. Holzmann, The model checker `SPIN`. In *IEEE Transactions on Software Engineering*, May 1997

[9] J. Krinke, Static Slicing of Threaded Programs, In Program Analysis for Software Tools and Engineering (PASTE 98), ACMSOFT, 1998.

[10] R. P. Kurshan, Computer-aided Verification of Coordinating Processes, Princeton University Press, 1995.

[11] F. Maraninchi, The Argos Language : Graphical Representation of Automata and Description of Reactive Systems. In *Proc. of IEEE International Conference on Visual Languages,* 1991.

[12] L. I. Millet and T. Teitelbaum, Slicing Promela and its applications to Model Checking, Simulation and Protocol understanding, In proc. of 4th workshop on Automata Theoretic Verification with the SPIN model checker, Nov. 1998.

[13] M. G. Nanda and S. Ramesh, Slicing Concurrent Programs. In *ACM SIGSOFT International Conference on Software Testing and Analysis (ISSTA 2000)*, August 2000.

[14] S. Ramesh, Refinement and Efficient Verification of Synchronous Programs. In *IFAC Workshop on Distributed Control Systems, Pergamon Press,* December 2000.

[15] S. Ramesh, Efficient Translation of Statecharts into Hardware Circuits, Proc. of 12th Int. Conf. on VLSI Design, IEEE Press, Jan. 1999.

[16] S. Ramesh, Communicating Reactive State Machines : Design, Model and Implementation. In *IFAC Workshop on Distributed Computer Control Systems, Pergamon Press*, September 1998

[17] F. Tip, Survey of Program Slicing Techniques, J. of Program. Lang., Vol. 3, 1995.

[18] G. Vinod, Efficient Verification of Synchronous Programs. *B.Tech Project Report, Computer Science and Engineering Department, IIT Bombay*, April 2001

[19] M. Weiser, Program Slicing. In *IEEE Transactions on Software Engineering,* pp 352-357, July 1984.

Appendix

Proof of Correctness

Here we will prove the correctness of the algorithm for the restricted case of criterion $< S, b >$, where S is the root state. This means that the predicate 'Reside' used in the definition is true. Hence what we need to prove is that the slice M_s given by the algorithm is such that for any input sequence σ, we have that $(M[\sigma]/b = M_s[\sigma]/b)$

For the purpose of proof, we model the execution of a machine as a sequence of *machine configurations*. A machine configuration represents the control state of the machine at a particular point of execution. It is essentially a set of machine states at which the control resides at the given point of execution. Due to the presence of hierarchy and concurrency, a configuration can contain more than one concurrent state. For example, when control resides at a child node, the control resides at its parent also. Similarly, control resides simultaneously at a set of states that are concurrent. The initial configuration consists of the root state and all the other appropriate states inside the root state, defined by the default arrows.

An execution step of a machine, in a given configuration of states results in a new configuration. A single execution step, due to the presence of concurrency, involves a number of multiple state transitions.

For example, in Figure 1 one of the configurations is the set of states { `count,counting,S1,S2,S3,A1,B1,C1` }. One step of execution in this configuration results in the new configuration { `count,not_counting` }. In this step the following series of state transitions takes place: `A1` to `A0`, `B1` to `B0`, `C1` to `C0` and `Counting` to `not_counting`. These transitions are triggered respectively by the signals `tick,lt1,lt2,end`. Except for the first one all the other signals are generated by the machine itself.

We will refer to the individual state transitions involved in a step as a *micro step* and whole the step itself as *macro step*; we use the notation $\rightarrow$ and $\Rightarrow$ to denote the micro and macro steps respectively. Thus using the notation, we have

{`count,counting,S1,S2,S3,A1,B1,C1`} $\Rightarrow$ { `count,not_counting`}

which represent a macro step. This macro step consists of the following micro steps:

$\{$`count,counting,S1,S2,S3,A1,B1,C1`$\} \rightarrow \{$`count,counting,S1,S2,S3,A0,B1,C1`$\}$
$\rightarrow \{$`count,counting,S1,S2,S3,A0,B0,C1`$\} \rightarrow \{$`count,counting,S1,S2,S3,A0,B0,C0`$\}$
$\rightarrow \{$ `count,not_counting` $\}$
Let M^i and M_s^i denote the initial configurations of the machines M and M_s respectively, and M^f and M_s^f denote the final configurations of M and M_s respectively after consuming the input sequence σ.
First we prove the following lemma :

Lemma 4.1 *$\forall\sigma$: $M_s^f \subseteq M^f$.*

Proof. The proof proceeds by induction on the length of the input σ. Let k be the length.
Base Case : $k = 0$
In this case, we have to prove that $\mathrm{M}_s^i \subseteq \mathrm{M}^i$. This is obvious since M_s has the same or less number of states than that of M. Further no new initial states are introduced.
Inductive Case : We assume that for all σ of length less than k, the result holds and prove that the result holds for σ of length k.

Let N, N_s be the configurations of machines M and M_s respectively after an input of length (k-1) is consumed. Further let M^f and M_s^f be the configurations of the respective machines after the kth input is absorbed. Then we have

$$\mathrm{M}^i \Rightarrow^* \mathrm{N} \Rightarrow \mathrm{M}^f \text{ and } \mathrm{M}_s^i \Rightarrow^* \mathrm{N}_s \Rightarrow \mathrm{M}_s^f.$$

where M^i and M_s^i are initial configurations of M and M_s respectively.

We know that $\mathrm{M}_s^i \subseteq \mathrm{M}^i$ and $\mathrm{N}_s \subseteq \mathrm{N}$. The kth input causes the macro step transition $\mathrm{N} \Rightarrow \mathrm{M}^f$. There exists a sequence of micro steps ξ that corresponds to this macro step. Let

$$\xi = N \rightarrow \mu^1 \rightarrow \mu^2 \rightarrow \ldots \rightarrow \mu^p \rightarrow \mathrm{M}^f,$$

where μ^is are intermediate configurations reached during the execution of the macro step.

From ξ, we construct a sequence ξ_s' for M_s that causes the macro step transition $N_s \Rightarrow \mathrm{M}_s^f$:

$$\xi_s' = N_s \rightarrow \mu_s^1 \rightarrow \mu_s^2 \rightarrow \ldots \rightarrow \mu_s^p \rightarrow \mathrm{M}_s^f,$$

where $\mu_s^i = \{\mathrm{x} \mid \mathrm{x} \in \mathrm{M}_s \wedge \mathrm{x} \in \mu^i\}$, $i \in \{1, 2, \cdots, p\}$.

ξ_s' may have duplicate occurrences of successive configurations. Let ξ_s be the sequence after removing the duplicate occurrences as given by:

$$\xi_s = N_s \rightarrow \nu^1 \rightarrow \nu^2 \rightarrow \ldots \rightarrow \nu^k \rightarrow \mathrm{M}_s^f (\nu^i = \mu_s^j, \text{ for some } j).$$

Clearly, the constraint $\mathrm{M}_s^f \subseteq \mathrm{M}^f$ is satisfied by ξ_s. We only have to show that the sequence ξ_s is a legal sequence for M_s . A sequence of micro-steps, such as ξ, is legal if and only if the following condition is satisfied: if there is a transition $\mu^i \rightarrow \mu^{i+1}$ that is triggered by a local signal b, then there must be a j $<$ i such that the transition $\mu^j \rightarrow \mu^{j+1}$ outputs the local signal b. To show that ξ_s is a legal sequence for M_s , we proceed as follows :

Consider the micro step $\mu_s^i \rightarrow \mu_s^{i+1}$ (with $\mu_s^i \neq \mu_s^{i+1}$). Suppose that this micro step is as a result of taking a transition T_i, triggered by a local signal `c`.

Clearly the same transition T_i causes the micro step $\mu^i \to \mu^{i+1}$ in ξ and this is also triggered by `c`. Hence, since ξ is a legal sequence of micro-steps for M, there exists j < i such that the micro step $\mu^j \to \mu^{j+1}$ is caused by a transition T_j that outputs `c`. This implies that there is a trigger edge from T_j to T_i. Since T_i is included in M_s , T_j is also included in M_s . Hence μ_s^j and μ_s^{j+1} will be distinct. Therefore, in the sequence ξ, if a transition is triggered by a signal, there will be a transition that occurs earlier in the sequence and outputs the same signal. Thus ξ_s is a legal sequence for M_s . Hence the lemma. □

Theorem 4.2 *The machine M generates* `b` *on an input sequence σ if and only if the machine M_s generates* `b` *for the same input sequence.*

Proof. The proof proceeds by induction on the length k of σ.

The proof for base case (when $k = 0$) is obvious since both M and M_s do not generate `b`.

Induction: Assume that the lemma is true for all σ of length $(k-1)$ and show that the result holds for σ of length k.

Let σ be of length k. Let $M \Rightarrow N \Rightarrow M^f$, $M_s \Rightarrow N_s \Rightarrow M_s^f$,
where $N(N_s)$ and M^f (M_s^f) be the configurations that result after $(k-1)$ and k macro steps of execution of M (M_s) with the input σ. We prove that b is generated at the kth macro step of the above execution of M iff b is generated at the k macro step of the above execution of M_s.

($\Leftarrow$) : Assume that M_s generates the output `b` at the k^{th} macro step. Suppose that this is generated as a result of a transition from state A in N_s to a state B in M_s^f. From the previous lemma, it follows that $\mathrm{N}_s \subseteq \mathrm{N}$ and $\mathrm{M}_s^f \subseteq \mathrm{M}^f$. Hence $\mathrm{A} \in \mathrm{N}$ and $\mathrm{B} \in \mathrm{M}^f$and the same transition will be taken on the machine M, and so the output `b` is generated on the kth macro step.

($\Rightarrow$) : Assume that M generates `b` at the kth macro step. Suppose that `b` is generated as a result of a transition from state $A \in N$ to $B \in M^f$. If $\mathrm{A} \in \mathrm{N}_s$, then the same transition will be enabled in M_s as well and hence `b` is generated in the kth macro step of M_s as well. It is not possible that $\mathrm{A} \notin \mathrm{N}_s$ since M_s includes only and all transitions which generate `b`.

Hence the theorem. □