Efficient Verification of Synchronous Programs

B.Tech. Project Report

Submitted in partial fulfillment of the requirements
for the degree of

Bachelor of Technology

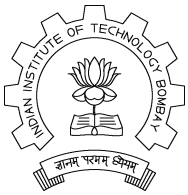
By

Vinod .G

Roll No : 97005027

Under the guidance of

Prof. S. Ramesh



Department of Computer Science and Engineering
Indian Institute of Technology
Mumbai

May 6, 2001

Acknowledgment

May 6, 2001

I would like to thank my guide, **Dr. S. Ramesh**, for giving me this interesting and absorbing topic to work on for my project. I would also like to thank him for the time that he gave me, discussing various aspects of the project. His guidance, encouragement and support were invaluable in the realization of this project.

Vinod G.

Abstract

This project considers the problem of efficient verification of synchronous programs. The underlying idea used is based on a recent proposal : To verify a property for a system, one method would be to decompose the system into smaller subsystems, and verify local properties on each of them. We use the technique of program slicing to decompose the system into smaller subsystems. In this project, we considered two synchronous languages : Communicating Reactive State Machines (CRSMs) and Esterel. CRSMs are a paradigm based on Argos, another synchronous programming language. We have proposed an algorithm that slices Argos programs and have formally proved that it produces correct slices. This algorithm was then extended to slice CRSMs. The other part of the project considers slicing Esterel programs. An attempt has been made to extend the notion of the Control Flow Graph (CFG) and the Program Dependence Graph (PDG) for Esterel.

Contents

List of Figures

Chapter 1

Introduction

In this project we discuss the problem of efficient verification of synchronous programs. Synchronous programs are used in the specification of reactive systems. A number of synchronous programming languages are used in practice, for eg. Esterel, Argos etc. [Hal93]. We restrict our attention to Communicating Reactive State Machines [Ram98], Argos [MR] and Esterel [Ber00a][BG92][Ber00b].

Communicating Reactive State Machines (CRSMs) are a graphical paradigm based on Argos that are used in the specification of distributed controllers. They consist of state machines, each of which is based on Argos, and are capable of communicating amongst themselves asynchronously. CRSMs also allow parallel composition of machines, thus allowing concurrent execution. Esterel is a popular language that is used in the specification of embedded systems.

In the verification of a reactive system, we are typically given a logical formula describing a property that the system has to satisfy. This property may be a safety or a liveness property. Typically, not all parts of the system may be required to show that it satisfies a given property. As is shown in [Ram00], we can have a compositional verification strategy. To verify a global property of a system, decompose the property into a number of local properties each of which holds in a subsystem. Verification of local properties would be simpler as subsystems would be smaller than the whole system. In this project, we try to decompose a big system into a smaller system using the technique of program slicing. Program slicing has been studied in the context of sequential programs [Wei84], concurrent programs [NR00] and programs with channels as first class objects [TM98],[TM99]. In this project we give slicing algorithms for Argos, CRSMs and Esterel.

1.1 Organization of this Report

The chapters that follow are organized as follows.

- Chapter 2 covers Communicating Reactive State Machines and Argos. We give an introduction to Argos, show its features and give an informal semantics. We also present a brief overview of CRSMs
- Chapter 3 discusses program slicing for sequential and concurrent programs. This is a prelude to Chapter 4 and Chapter 5, where we use the concepts developed in this chapter and extend the definition of a slice to CRSMs and Esterel respectively.
- In Chapter 4, we first consider a subproblem of slicing CRSMs, that of slicing Argos. We present an algorithm that has been developed to slice Argos programs. We also present a proof that the algorithm indeed produces correct slices. This algorithm is then extended to work for CRSMs.
- In Chapter 5, we present the work done in slicing of Esterel programs. We define formally the notion of a Control Flow Graph and a Program Dependence Graph for Esterel.
- Chapter 6 gives a summary of the work done and gives pointers for future work.

Chapter 2

Communicating Reactive State Machines and Argos

2.1 Introduction

In this chapter, we introduce Communicating Reactive State Machines (CRSMs) [Ram98] and Argos [MR]. CRSMs consist of asynchronously communicating state machines. Each machine can be thought of as an Argos program augmented with channels for asynchronous communication. Argos is based on the popular language Statecharts [Har87], [HN95]. Statecharts are widely used in the specification of reactive systems. We do not discuss Statecharts here, and only give a brief discussion of CRSM and Argos. We first discuss Argos, and then give a description of CSRM, because CRSMs build upon Argos.

2.2 The Program Model

CRSMs and Argos are synchronous programming languages and are assumed to follow the synchrony hypothesis, which states that the reaction time of the system to the input signals from the environment is zero. We assume that we have the system working in an environment where the environment changes at the occurrence of a clock tick and the system reaction time is very small as compared to the clock ticks as shown in the figure 2.1

2.3 The Language Argos

The language Argos [Hal93] is very much like Statecharts in that it allows parallel composition and hierarchical automata. However, Argos is simpler than

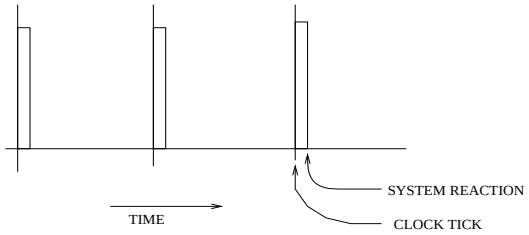


Figure 2.1: Our System Model.

Statecharts because Argos is a language whose semantics has been completely formalized, unlike Statecharts, whose semantics seems to be under discussion. Argos however differs from Statecharts, in that it *does not allow inter-level transitions.*

Argos programs consist of two main components : Automata and Operators. These Automata are used to represent processes that receive signals from the environment and emit signals to the environment. The automata can be put in parallel and each of their states can be refined into processes, which are activated whenever the parent automaton is entered, and killed whenever the parent automaton is exited. A simple process is described as an automaton, which consists of states that are named, transitions that are labeled and have signals that govern when a transition can be taken. These signals come in three flavors :

- Internal signals : those which have been declared local at either this process, or in one of it's ancestors.
- Input signals : those which cannot be emitted by the program.
- Output signals : those which cannot be used as input in any transition of the automaton.

Every transition in an automaton consists of an input part which may consist of signals or their negations (except output signals), and an output part which can only consist of signals. When a process is in a particular state, and a signal that is on the input part of a transition from that state arrives, the process takes that transition and emits the output signal associated with that transition.

The other ingredient of Argos programs is operators. The two kinds of operators allowed in Argos are the "parallel" operator and the hierarchical

decomposition operator. We shall consider both in detail and describe their semantics.

2.3.1 Parallel Operator

This is very similar to the parallel composition in Statecharts where we show two automata running parallelly by drawing them in a box and separating them by a dotted line. The sets of states of a parallel process is the Cartesian product of the sets of states of it's components. Each parallel process works in an environment that consists of the global environment and the other process. The semantics is as follows. Figure 2.2 gives two examples :

- Whenever a component can react to a signal it reacts. The communication mechanism by signal broadcasting. It is useful to think of the sequence of events happening as follows. The signals of the activated transitions are taken at once but the system enters the target states only after all the possible components that can react to the signals in the environment have reacted.
- When many components of a parallel composition react, the output is the conjunction of all the component outputs
- Components communicate with each other during a reaction. Internal signals emitted by a process can be considered as input signals for a parallelly running process.

2.3.2 Hierarchical Decomposition

The hierarchical decomposition of an automaton considers that each of the states of that automaton can themselves be considered as processes. Syntactically, it can be shown by refining the state of an automaton into a subprocess, or parallely running subprocesses. The semantics rules of this operator are described below for an automaton **A**.

- When **A** enters a state containing a subprocess, this subprocess is activated in it's initial state.(it becomes *active*)
- When **A** leaves such a state, the subprocess is killed, and all information about its current state is lost.(it becomes *inactive*)
- The signals emitted from the active subprocesses of **A**, if they are not local to these processes, are visible to **A**.

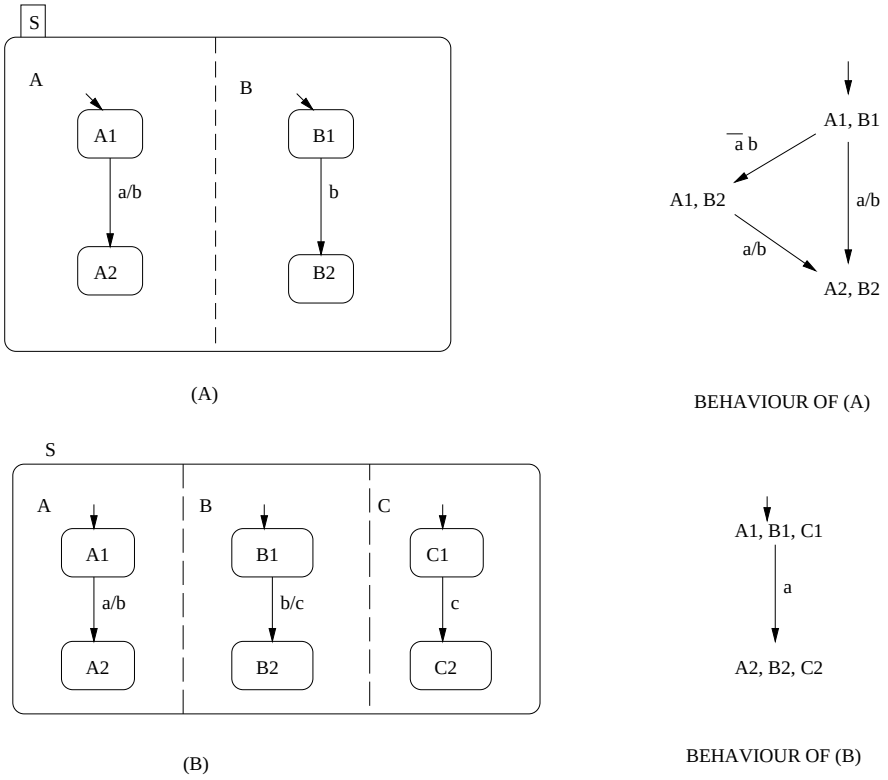


Figure 2.2: Parallel Composition and Equivalent Behavior.

- Any signal visible from **A** can be seen from an active subprocess of **A**, if the subprocess does not have a local signal with the same name.
- A subprocess cannot play any part in the reaction that activates it, however, it could participate in the transition that kills it, and the interruption will take place at the end of the reaction.

We show an example of Hierarchical decomposition and it's equivalent behavior in figure 2.3

2.4 Causality Problems

There are some paradoxes that may occur when we write Argos programs, we shall discuss them in brief over here, but for our purpose, we assume that such paradoxes can be detected by the compiler and so we do not consider these problems in the forthcoming chapters.

There are two kinds of paradoxes, as shown in figure 2.4. Figure 2.4(a) shows absence of behavior, If a does not occur then the transition is activated,

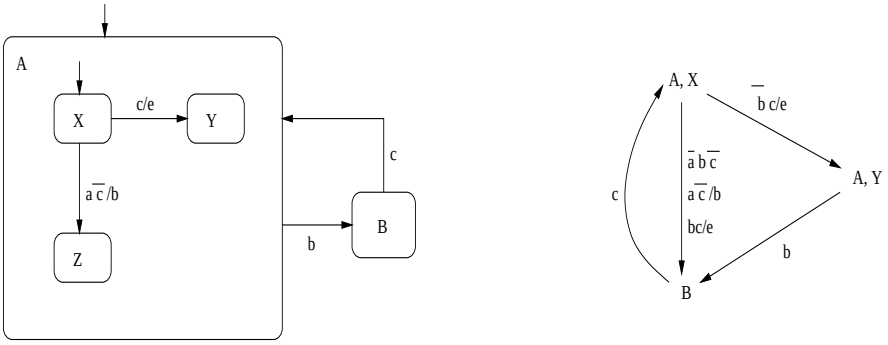


Figure 2.3: Hierarchical Decomposition and Equivalent Behavior.

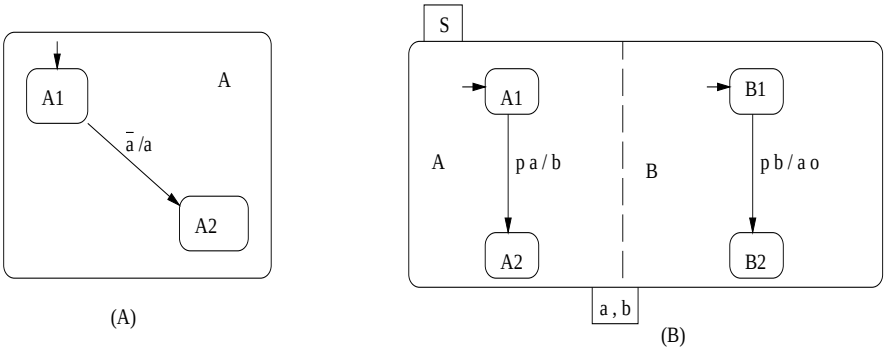


Figure 2.4: Causality Problems in Argos.

and a is emitted, which in turn means that, in the first place the transition should not have taken place at all. The other situation is described in figure 2.4(b), where either both the parallel components make a transition, or none makes a transition. This is called implicit non-determinism.

2.5 Semantic Functions

We discuss two possible semantic functions for Argos :

- **Input Output Semantics :** Here the machine is assumed to be associated with a function F : $P(\iota)^* \rightarrow P(o)^*$, where ι and o are the inputs and the outputs of the machine respectively. In case of the example in figure 2.3, the function F is such that F({c}, {b}, {c}, {c, a}) = ({e}, ϕ, ϕ, {e}).

- **Operational Semantics :** We first define the *configuration* of a machine to be the maximal set of states that it can reside in at any point of time. When an input is fed to the machine, it may lead to a change in configuration. For eg., suppose in figure 2.3 the root state was S and it

was fed with the input sequence ({c}, {b}, {c}, {c, a}) the following is the configuration at the end of each input fed to the machine :

Initially : {S, A, X}.

After {c} : {S, A, Y}.

After {b} : {S, B}.

After {c} : {S, A, X}.

After {c, a} : {S, A, Y}.

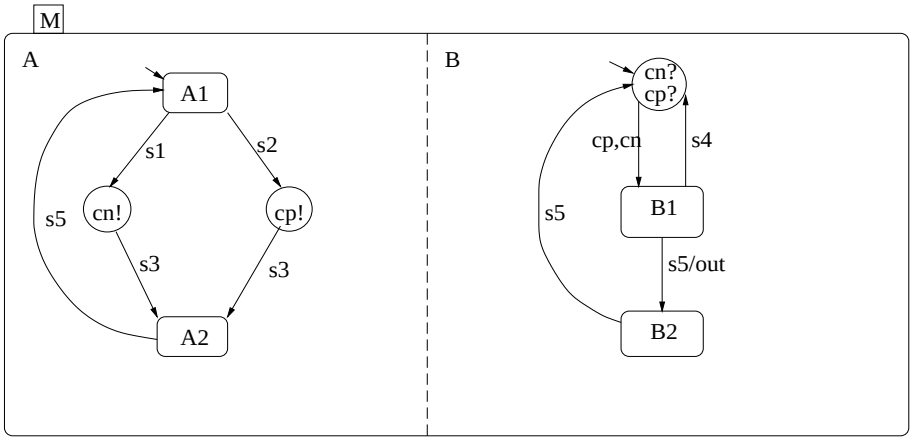
2.6 Communicating Reactive State Machines

As previously mentioned, this is a language that builds upon Argos. The difference here is that we have a number of machines, each of which can be thought of as an Argos program. They are allowed to communicate with each other asynchronously by means of channels. When Argos is augmented with channels, we get CRSMs.

CRSMs are used in the description of distributed control systems consisting of a number of autonomous subsystems controlling disjoint physical systems, but at the same time having system-wide control laws. They are becoming popular because of their properties of concurrency and modularity. Here a brief overview of CRSMs is given.

A CRSM is a network $N_1||N_2||...||N_m$ of independent reactive programs or nodes. Each node has its own reactive interface, separate input output signals and its own notion of instants. Each node is described by an Argos program. Thus each of the sub-components has all the features that we described for Argos, viz. hierarchical decomposition, parallel composition, and signal hiding. These features are sufficient for describing centralized controllers controlling local environments. The additional feature that CRSMs add is that of *Asynchronous Composition*. This feature is essential when we consider controllers for distributed systems. We introduce special kind of states called *rendezvous states*. A rendezvous state indicated a communication with another node via a channel. The channel name is specified as a label in the state. When we enter such a state it means that we are initiating an attempt to communicate via that channel. The node will remain in that channel till the communication has taken place. Once the communication is completed, exit from the state

takes place through one of the exit transitions from the state. A node of a simple CRSM is shown in the figure 2.5. This example is drawn from the train controller discussed in [Ram98].



Note : 1. Rendezvous states are shown as circles.
2. cp! means that the signal cp is going to be sent across the channel at that state
3. cp? means that the signal cp is going to be revceived from the channel at that state

Figure 2.5: Node of a CRSM.

2.7 Motivation for the discussion

The question arises, why we discussed all this. We will be discussing, in the forthcoming chapters, slicing of CRSMs, for which we will consider slicing of Argos first. To be able to slice the program given to produce a smaller but equivalent machine, we need to understand the semantics of the language clearly. This chapter provided an overview of the features provided by Argos and CRSMs, and also briefly discussed the semantics, albeit informally. We shall be using these concepts in the later chapters.

Chapter 3

Program Slicing

3.1 Introduction

In this chapter, we describe program slicing for sequential and concurrent programs. Program slicing will later be described for Argos programs, which will essentially draw from the ideas presented in this chapter, particularly the one on slicing of concurrent programs.

3.2 Program Slicing in Sequential Programs

In sequential programs, we are often interested acquiring information as to which lines of the program affect the value that a variable takes at a particular line in the program. This is solved by slicing the sequential program with respect to certain criteria which we discuss here. In this section, we are interested only in programs that terminate.

3.2.1 Slicing Criterion and Definition of a Slice

We define a *state trajectory* of length k as a finite list of ordered pairs (n_1,s_1), ... , (n_k,s_k), where each n_i is a statement of the program P, and s is a state, the mapping of variables to their values. Each (n,s) gives the value of the variables before the execution of n. A *slicing criterion* of a program P is a tuple $< i, V >$, where i is a statement in P and V is a subset of the variables in P.

A slice S of a program P on a slicing criterion C $= < i, V >$ is any executable program with the following 2 properties :

1. S can be obtained from P by deleting zero or more statements from P.

2. Whenever P halts on an input I with state trajectory T, then S also halts on input I with input state trajectory T', and $\text{Proj}_C(T) = \text{Proj}_{CP}(T')$, where CP = <succ(i), V> and succ(i) is the nearest successor to i in the original program which is also in the slice, or i itself if i is in the slice.

$\text{Proj}_C(T)$ basically considers only those variables from the state trajectory that are of interest to us. What the definition of the slice claims is that, if we consider the state trajectory of the sliced program and the state trajectory of the original program restricted to only those statements that consist of variables that we are interested in, then these state trajectories will be the same. For a state trajectory T = t_1 t_2... t_n, we define $\text{Proj}_C(T) = \text{Proj}'_C(t_1)...\text{Proj}'_C(t_n)$ where, $\text{Proj}'_C((n,s)) = (n, s|V)$ if n = i, and the empty string otherwise.

3.2.2 The Slicing Algorithm

The original slicing algorithm, proposed by Weiser in [Wei84], works on the basis of finding program dependencies. We refer to the set of variables referred to in a statement ie, occurring on the RHS by REF(n), and the set of variables whose value is defined by the program statement as DEF(n). On the basis of this, we define $R_C{}^0(n)$, which is the set of relevant variables at line n, as all variables v such that,

1. n = i and v is in V.
2. n is an immediate predecessor of a node m such that either,
   (a) v is in REF(n) and there is a w in both DEF(n) and $R_C{}^0(m)$ or,
   (b) v is not in DEF(n) and is in $R_C{}^0(m)$.

The statements included in the slide by $R_C{}^0(n)$ are denoted by $S_C{}^0$ and is defined as all those program statements such that $R_C{}^0(n+1) \cap \text{DEF}(n) \neq \phi$. $R_C{}^0(n)$ maps statements to sets of variables while $S_C{}^0$ is a set of statements, which is called the slice of the program. However, this is not all that there is to it. We have so far defined an algorithm only if the control flow in the program strictly starts at the first line, goes through all the lines of the program and ends at the last line of the program. It is not sufficient to consider control structures like branching, looping etc. For eg., consider the program

```
1. READ (X)
2. IF X < 1
3.      THEN Z := 1
```

4. ELSE Z := 2
5. PRINT (Z)

In the above program, if we were to slice with respect to the criterion ¡3, Z¿ then for the slice to correct, we must include statement 2 in the slice as well. The current model does not consider this. To solve this problem, Weiser considers what is called the inverse dominator of a particular statement, which is a statement that occurs in every path from the statement of interest to the exit point of the program. All the lines in the program from the line under consideration to the nearest inverse dominator are included in the slice. Now, all the statements influencing the values of the variables in the statements just included are included in the slice. This procedure is carried on till a fixed point is reached. We do not go into the details of this method here.

The later versions of slicing algorithms for sequential programs do not perform all the calculations that we have done here. They assume the existence of data dependency and control dependency edges, construct the *Program Dependency Graph*. The problem of slicing is reduced to a reachability problem here. We just trace back the path from the statement of interest to the starting node of the problem, in the process including all the nodes of the graph that we encounter. This is considered in detail in the chapter on Esterel slicing, so we defer a detailed discussion to that chapter.

3.3 Slicing Concurrent Programs

In this section, we introduce the slicing of concurrent programs [NR00]. This algorithm uses the concept of the program dependency graph that we introduced in the previous section. This algorithm is the one from which a number of ideas have been drawn for the slicing of Argos programs.

3.3.1 Slicing Criterion and Definition of a Slice

The difference here as compared to the sequential program case is that there are multiple threads of execution, and the statements of the parallely executing threads can be combined in any arbitrary order to give a valid execution sequence.

The slicing criterion for a concurrent program is defined by a triple <t, p, V>, where t is a thread in the program p and V is the set of variables that we

are interested in. Thus given the threaded program dependency graph (TPDG), we define the slicing criterion as just a node in the TPDG. The various kinds of edges in the TPDG are the control edges, data dependence edges, loop carried data dependence edges, and interference edges.

Before we define what we mean by a slice of a TPDG, we give some definitions.

1. A trace in the TCFG (threaded control flow graph) G, of a concurrent program is ordered set of nodes <n_1, n_2, ..., n_k> that forms a valid execution path in G.

2. The trace set of G, denoted by Tr(G) is the set of all traces in G.

3. A trace witness in G is an ordered sequence of nodes <n_1, n_2, ..., n_k> in G, such that it forms a subsequence of some trace in the set Tr(G).

A slice of the TPDG w.r.t a slicing criterion p consists of all nodes q on which p transitively depends and there exists a trace witness in G for each such q. Formally,

$$S(p) = \{q \mid P = <n_1, n_2, ..., n_k>, q = n_1 \rightarrow^{d_1} ... \rightarrow^{d_{k-1}} n_k = p,$$
$$\text{for } 1 \leq i < k,\ d_i \in \{cd, dd, id, dd_l, \text{for some loop l in G}\}\},$$
$$\text{where P is a trace witness in G.}$$

In other words, it means that the slice consists of all those statements that affect the value of p, and hence occur in a trace witness ending with p.

3.3.2 The Slicing Algorithm

There is an important point to be considered here before we discuss the outline of the algorithm. Interference dependence edges are not transitive. In figure 3.1, we see that the statement 3.1 is dependent on 2.2 and 2.1 is dependent on 3.1. However, 2.1 can never be dependent on 2.2. So during the slicing process, we must keep track of this fact.

The slicing algorithm is a work-list based algorithm. It maintains a tuple containing one entry per thread of the program to keep track of the last node visited in each thread. This tuple changes in each iteration of the program. This is required because of the problem we just mentioned. Interference dependence is not transitive, and so we must keep track of the last node visited in each thread. We must add a node to a thread in the slice only if we find that the node being added (which affects the value of the node mentioned in the slicing

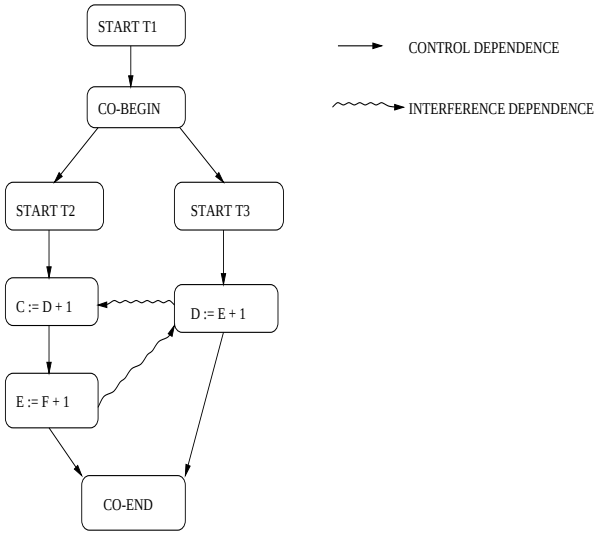


Figure 3.1: Interference Dependence in a TCFG.

criterion) is such that there is a path from that node to the node corresponding to the entry for that thread in the tuple we are maintaining.

The slice is calculated incrementally. It starts with the node mentioned in the slicing criterion and moves backwards along the data dependence, interference dependence, control dependence and loop carried data dependence edges, and in the process adds the nodes obtained to the partial slice obtained All nodes added using data and control dependence are just added to the slice. For those node which are obtained by considering interference edges, we look at the tuple and add the node only if the criterion previously mentioned is satisfied. This is done till we cannot add any more nodes to the slice at which point the algorithm terminates. This algorithm has been proved correct. It also produces minimal slices in that it adds no more nodes to the slice than those which affect the statement mentioned in the slicing criterion.

3.4 Summary

The question arises now, how are we going to apply these slicing techniques to slicing Argos programs? Argos is different in the sense that it is a graphical paradigm. The algorithms presented in this chapter must somehow be extended to accommodate state machines. A number of questions are to be answered : What could be the slicing criterion? And what do we mean by a slice of an Argos program? These questions are addressed in the next chapter.

Chapter 4

Slicing CRSMs

4.1 Introduction

The main aim of this project is to extend the notion of a slice to CRSMs, and present an algorithm that produces a slice conforming to this definition. Since CRSMs are based on Argos, we first define the slice of an Argos program, give an algorithm to slice Argos programs, and prove that it produces correct slices. The algorithm is then extended to CRSMs.

4.2 The Slicing Criterion for Argos Programs

Before we describe the slicing criterion, we must decide for what reasons we would slice a program written in Argos. Since Argos programs consist of automata with output, a debugger or a verifier will be interested in the behavior of the system. The behavior is described by the signals that are output by the machine, when it is fed with an input sequence. Thus a signal is surely part of the slicing criterion. Now, if we have additional information about the state of the system after it's response to an input sequence, it will certainly help us in debugging. Thus we must also include the state that the system resides in, in our slicing criterion.

Advantages of having an *output signal* only as slicing criterion :

1. More realistic from the viewpoint of an user of the system, who most probably will have no information about the state of the program.

2. We may not always have precise information about which state the system is in, for eg. we may just know that the system is in a particular state, but may not know in which of it's sub-states it resides in.

Advantages of an *output signal + state* as a slicing criterion :

1. Very much like the slicing criterion in sequential and concurrent programs. We mentioned not only the variables that we are interested in but also the line number that we are interested in. Here the parallel of the variable is the output signal, and the analogue of the line number is the state of the program.

2. This approach is more useful from a debugger's point of view.

3. If the state mentioned need not be basic, then this method can be thought of as a more general version of using just the signal as the slicing criterion.

The slicing criterion was hence chosen as $< S, b >$, where S is any arbitrary state of the Argos program, and b is an output signal, whose behavior we want to study.

4.3 Slice of an Argos Program

Consider an arbitrary input sequence I* (I* is a finite input sequence). Note that the input sequence is composed of the inputs given to the machine from the environment at each clock tick. Each member of the sequence can thus in general be a set of signals. If a signal b is not in the input, then we can assume the existence of the event $\bar{b}$ in the environment at that point of time. Consider a machine M. M is assumed to have a state S and b is assumed to be an output signal. A slice of the machine M with respect to a slicing criterion $< S, b >$is defined as any machine M_s such that :

1. M_s is obtained by removing zero or more states from M.

2. Let ι be the set of signals that can be given as inputs to the machine M. Let $P(\iota)$ denote the power set of ι. Then $(P(\iota))^*$ is an input sequence to M. We denote $(P(\iota))^*$ by I* . Let M reside in a state S when I* is fed to it. M_s should be such that when I* is run on it, it resides in S, and the output sequence M_s[I*] should be such that M[I*]/b = M_s[I*]/b, ie., restricted to the signal b, the output sequences should match. (here M[] and M_s[] are the functions for the machines M and M_s respectively, as defined in the Input-Output semantics)

More formally : M_s is any machine such that,
($\forall$I* $\in$ Reside (M, I* , S)) : ((Reside (M_s , I* , S) $\wedge$ (M[I*]/b) = (M_s[I*]/b))).

where M_s is obtained by removing zero or more states from M, and Reside(M, I^* , S) is a predicate that is TRUE whenever M resides in S after the input sequence I^* is run on it.

Note that we do not say anything about the slice if the machine does not reside in the state S in the slicing criterion for an input sequence I^* . What we only claim is that if the machine M resides in the state S after I^* is executed on it, *then* M_s also will reside in S and the outputs sequences will match with respect to the signal b.

4.4 An Example

We present an example of what we mean by a slice in figure 4.1. The slice that we have given is what we would expect as a slice of the program with respect to $< S, b >$. That is because we are interested in the behavior of the system with respect to the signal b. We have in the slice shown all possible ways in which we can produce b, and reside in S. The slice shown in the figure is precisely what is generated by our algorithm.

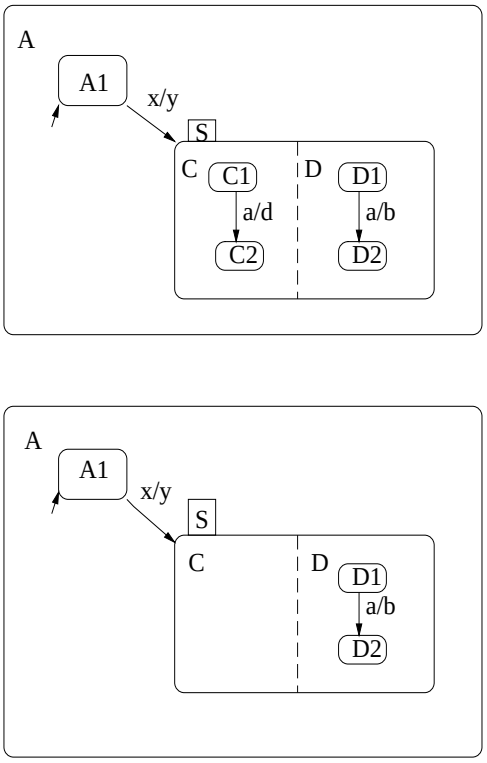


Figure 4.1: A CRSM and its slice.

4.5 The Slicing Algorithm

We now present an algorithm that given a machine and a slicing criterion produces a machine that conforms to the definition of the slice that we have just given. Before that a few terms have to be defined.

- **Trigger Edges :** These edges are drawn by our algorithm. They are drawn between two edges such that the output of one affects the input of another. In other words, if a transition 2 is such that whether it can be taken or not is dependent on whether another transition 1 is taken then we draw a trigger edge from edge 1 to edge 2. An example is shown in figure 4.2

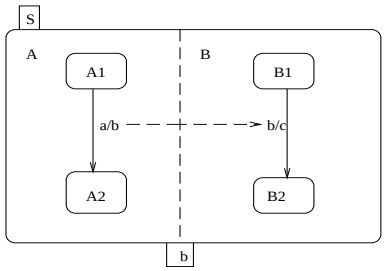


Figure 4.2: Trigger Edges.

- **Source, Target** of a trigger edge : The transition *from* which the trigger edge is drawn is the Source, while the transition *to* which the trigger edge is drawn is the Target of the trigger edge.

- **Source, Target State** of a trigger edge : The parent state of the two states which lie on either side of the source of a trigger edge is called the Source State of the trigger edge. The Target State is defined similarly. In the example of figure 4.2, 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 defined by construction as follows :

  - For the trigger edge in consideration, write down it's source and target states, let us call them Src and Tgt.
  - Write down the hierarchy of Src in reverse order till the root state of the machine.
  - Write down the hierarchy of Tgt in reverse order till the root state of the machine.

- In the two lists, note the first state (ie. farthest from root state) that is common to both lists.
- The child of this state in the hierarchy of the Src is the Highest-Possible-Source-State of the trigger edge.

Let us consider 4.2 as an example. The source state is A and the target state is B. the hierarchies of these states are {A, S} and {B, S}. The state that is common to both lists and is farthest away from the root state is S. the child of S in the hierarchy of the source state is A. Hence A is the highest possible source state.

Why these definitions were required will become clear when the algorithm is described, however we give a small motivation as to why Highest-Possible-Source-state was required. Trigger edges can be drawn from one transition to another, and each of these transitions could be as low down in the hierarchy as possible. The highest possible source state can be thought of as the highest level in the hierarchy up to which the trigger edge can "rise" as the trigger of another transition, before it starts "falling" into the hierarchy of states in which it triggers a transition.

4.5.1 The Main Slicing Procedure

The main idea used in the slicing algorithm is to retain all the transitions that output the signal b mentioned in the slicing criterion. We will also retain those transitions that somehow "lead" to these transitions that output b. We examine each state to see if it "has" inside it a transition that gives and output b. If so, we refine it straightaway. Otherwise, it is refined only if it contains a transition that can trigger a local signal - which in turn may trigger the emission of b.

The main procedure that slices the Argos program is the *Slice* procedure described in Algorithm 1. This procedure takes as input an Argos program M, and a slicing criterion of the form $< S, b >$. It produces as output sufficient information to construct another program - M_s , that conforms to the definition of slice that we have given. It calls 4 procedures namely *Draw-Trigger-Edge*, *Top-to-Bottom*, *Slice-This-Level* and *Trigger-Edge-Slice* which are described in the following sections.

It must be noted that the machine M given as input can be thought of as a set of states. To completely specify the machine M, in addition to the set of states, we also need to mention the transitions, the hierarchical order between the states, which signals are hidden and at what level of the hierarchy they

are hidden. The output of the algorithm : *slice* is given as a set of states. It is a variable that is declared global, and hence can be accessed by all the auxiliary procedures. Since the sliced machine is defined to be a subset of M, the set of states *slice* must be interpreted as those set of states from M which are included in the sliced machine M_s . All transitions between the states mentioned in *slice* that were present in M, are present in M_s as well. The hierarchical order between the states must also be maintained. All signals that were hidden in M remain hidden in M_s also. Therefore, the set of states *slice* is sufficient to completely specify the machine M_s .

Algorithm 1 The Main Slicing Procedure : Slice (M, $< S, b >$)

slice := ϕ
Call Draw-Trigger-Edge(M)
Call Top-to-Bottom ($< S, b >$)
while ($S \neq root$(M)) **do**
  S = **Call** Slice-This-Level ($< S, b >$)
end while
if ($root$(M) $\notin$ slice) **then**
  slice := slice $\cup$ { $root$(M) }
end if
Call Trigger-Edge-Slice(M)
Return slice

4.5.2 Auxiliary procedure Draw-Trigger-Edge

This procedure draws the trigger edges. In the following discussion, we describe a method that computes the trigger dependencies in a precise manner. The method draws upon the formal definition of encapsulation as suggested in [MR]. Intuitively, a transition is present in the automaton after encapsulation if it is such that a local signal that is supposed to be present has to be emitted in the same reaction, and a local signal that is supposed to be absent should not be emitted in the same reaction.

The algorithm that draws trigger edges must capture the fact that a transition is triggered by another transition if and only if it is such that the second transition is triggered by a local signal, which is emitted as a result of the first transition. Clearly, if there is a state S such that exit from S can be triggered by a local signal, and the local signal is generated from within S as a result of some transition between two descendants of S, say A and B, then there has

to be a trigger edge from the transition between A and B and the transition triggered by the local signal, leading out of S.

In case we have two automaton working in parallel, drawing of trigger edges is more complicated. We shall explain our method by means of the example shown in figure 4.3.

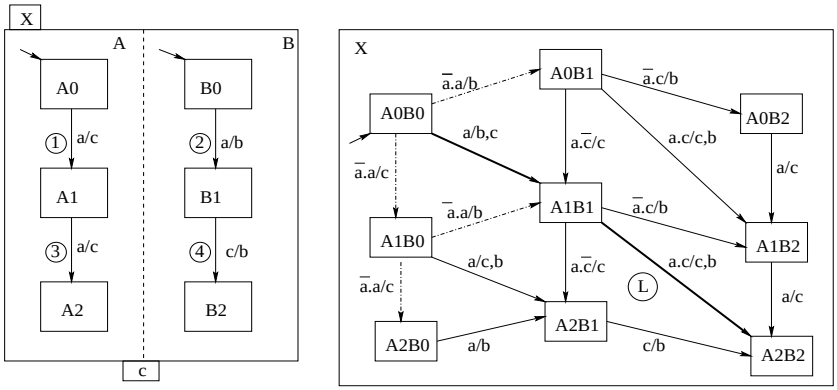


Figure 4.3: Drawing Trigger Edges.

- Our algorithm starts off by considering the product automaton. Transitions which have inputs like a.$\bar{a}$ are not considered legal and are dropped from the product automaton.

- In the resulting product automaton, we consider the states reachable from the starting state and only consider the transitions amongst these. In the example shown in figure 4.3, we consider only those transitions shown in thick lines.

- Amongst these transitions, look at those transitions which have a local signal, in this case "c", in both the input and output part of the transition. Only the transitions corresponding to this transition in the original machine will have trigger edges between them. For eg, in figure 4.3, transition L in the product automaton corresponds to transition 3 and transition 4 in the original automaton. Only those have a trigger edge between them.

It can be seen that this procedure blows up the number of states in the product automaton. Thus, we do not use this method in our algorithm. We instead use a less precise method where we draw trigger edges between every pair of

edges such that the output of the one contains a local signal, say "c" and the input of the second contains the same local signal. Our method of drawing trigger edges will draw more edges than are required. For eg. in 4.3, we will draw a trigger edge from transition 1 to transition 4, and from transition 3 to transition 4. The former is not not a trigger dependency as is clear from the product automaton. It must be emphasized that the method we adopt to draw trigger edges will not result in wrong slices - the slice will only include more transitions than the minimum number required.

4.5.3 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 recursively has the signal b (also mentioned in the criterion) inside it. If it does not have the signal, then we can return the machine without refining it, because, then, behavior w.r.t b is maintained. If however, the machine had the signal b, then we include all the child states in the slice and call the same procedure recursively on each of the children. The algorithm is described in Algorithm 2. It must however be noted that Algorithm 2 refers to a computation of $has(< S, b >)$, which is a very costly procedure, since it recurses down the structure of the machine to search whether a transition that outputs the signal b is present anywhere. This is repeated for each recursive call of the procedure, thus making the call to Top-To-Bottom a very costly one. Hence, we also give in Algorithm 3 and Algorithm 4 an efficient implementation of the same algorithm which does not compute $has(< S, b >)$ at each step. In this implementation, we check for the presence of b not recursively, but by only checking if the transitions between the children of the state under consideration output b or not. However this algorithm is more complicated to implement, so there is a tradeoff between simplicity of the procedure and the difficulty of implementation. The algorithm takes as input an Argos program M whose root state is S.

Algorithm 2 Top-To-Bottom ($< S, b >$)

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

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

Call Top-To-Bottom-1 ($< S, b >$, S)

Algorithm 4 Top-To-Bottom1 ($< S, b >$, caller-state)

if (S == caller-state) **then**
  slice := slice $\cup$ { S }
end if
if (S.child-list == ϕ) **then**
  Return FALSE {basic state}
end if
S.transitions := {t | $S_1 \rightarrow^t S_2$, for some S_1, $S_2 \in$ S.child-list}
Temp-slice := ϕ
for all (child $\in$ S.child-list) **do**
  if (*Top-To-Bottom-1*($<$ child, $b >$, caller-state) == TRUE)) **then**
    Temp-slice := Temp-slice $\cup$ { child }
  end if
end for
for all ((child $\in$ S.child-list) | child $\notin$ Temp-slice) **do**
  if ($\exists$t | (t $\in$ S.transitions) $\wedge$ (Out(t) == b) $\wedge$ (To(t) == child)) **then**
    Temp-slice := Temp-slice $\cup$ {child}
  end if
end for
if (Temp-slice == ϕ) **then**
  Return FALSE
else
  slice := slice $\cup$ S.child-list
  Return TRUE
end if

4.5.4 Auxiliary Procedure Slice-This-Level

This is the procedure by means of which we move up the hierarchy from the state mentioned in the slicing criterion. We perform a reachability and 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, we refine these states in such a way that the behavior with respect to the signal "b", mentioned in $< S, b >$, is retained. This procedure will not be executed if the state mentioned in the slicing criterion is the root state of the machine. Its return value is the parent state of the state mentioned in the slicing criterion.

Algorithm 5 Slice-This-Level ($< S, b >$)

S.worklist := ϕ
if (S.parent == "AND-state") **then**
  S.worklist = {st | st = sibling (S)}
else
  while ($\exists$(ch $\in$ sibling(S)) | (ch $\rightarrow$ S) *or* (ch $\rightarrow$ K), for K$\in$S.work-list) **do**
    S.work-list := S.work-list $\cup$ { ch }
  end while
  if (S is involved in a cycle in S.work-list) **then**
    S.work-list := S.work-list $\cup$ { S }
  end if
end if
for all (ch $\in$ S.worklist) **do**
  Call Top-To-Bottom ($<$ ch, b $>$)
end for
slice := slice $\cup$ { S }
Return *Parent* (S)

4.5.5 Auxiliary Procedure Trigger-Edge-Slice

This procedure adds those transitions which may affect the emission of the signal b mentioned in the slicing criterion. We look at those transitions which directly affect the emission of b and include these in the slice. Then we look at those transitions which could affect the emission of the signals added in the previous step. We continue till we are unable to add any more transitions.

Algorithm 6 Trigger-Edge-Slice(M)

List-Trigger := {t | t is a Trigger Edge}
while (∃(t ∈ List-Trigger) | Target-Transition(t) ∈ slice ∧ (Source-state(t) ∉ slice)) **do**
  Call Top-To-Bottom (<Highest-Possible-Source-State(t), b>)
end while

4.6 Another Example

Consider figure 4.4. Clearly even if B and C are not refined, the behavior of the machine with respect to the signal b remains the same. Thus when we slice the machine A with respect to the criterion $< A, b >$, we expect the second machine shown in 4.4. That is precisely the output of our algorithm.

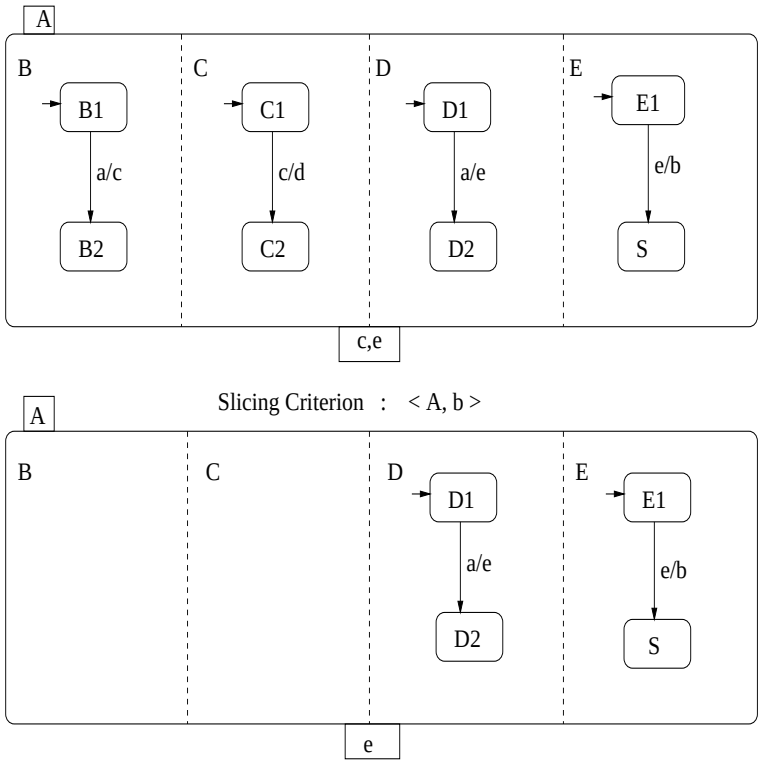


Figure 4.4: A CRSM and its slice (with Trigger dependencies).

4.7 Proof of Correctness

In this section, we will prove the correctness of a restricted version of the slicing algorithm that has been proposed. Recall that we defined the slice of a machine

M with respect to $< S, b >$to be another machine M_s such that :
($\forall I^* \in$Reside(M, I^* , S)) : (Reside(M_s , I^* , S) $\wedge$ (M[I^*]/b) = (M_s[I^*]/b)).

While proving the algorithm, we impose the restriction that the state information in the slicing criterion $< S, b >$be the root state of the machine M. In other words, we are forcing the Reside predicate to be TRUE always. Thus, if in $< S, b >$, S = $root$(M), then the slice M_s of a machine M is any machine that satisfies : $\forall I^*$: (M[I^*]/b = M_s[I^*]/b). In the proof that follows, we show that the output of our algorithm satisfies this condition.

Let the machine M_s be the output when the machine M is given as input to our slicing algorithm. Let I^* be the input sequence fed to the machine M and the machine M_s . 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^f respectively.

Lemma 1 : $\forall I^* : M_s^f \subseteq M^f$.

Proof : The proof proceeds by induction on the length of the input I^* .

Base Case : I^* is of length 0, ie., $M^f = M^i$ and $M_s^f = M_s^i$. Hence we are required to prove $M_s^i \subseteq M^i$. We define the *height* of a machine as `max`(height of its children) + 1, and set the value of height of the NIL machine = 0. Let h denote the height of a machine. The base case is proved by induction on h :

Base Case : $h = 0$. This means that the machine is NIL. For the NIL machine $M_s^i = M^i$. Hence $M_s^i \subseteq M^i$.

Inductive Case : Assume that $M_s^i \subseteq M^i$ for all machines with height $<$ j. Consider a machine of h = j. The root state of the machine can either be an AND-state or an OR-state (as shown in the figure 4.5. Consider both cases separately :

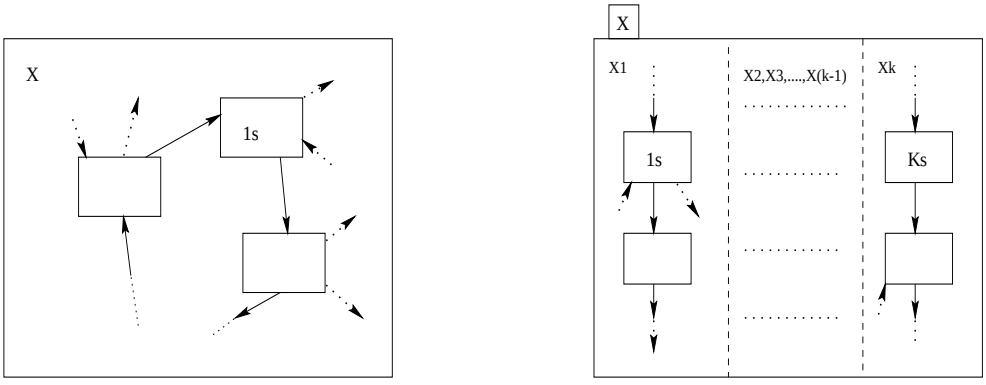


Figure 4.5: Two ways of constructing a machine of height = j

- *OR :* In this case, X is the root state of M. It is refined into many states as shown in figure 4.5. At any point of time, control will reside in exactly one of the children of X. Say the control resides in 1S. For the machine K, with root state as 1S, we know that $K_s^i \subseteq K^i$ by the induction hypothesis. Our slicing algorithm refines the state X of the machine M if and only if it "has" the signal "b" inside it (ie. there is a transition between two descendents of X that emits b). If "b" is present somewhere inside X :

  $M_s^i = K_s^i \cup \{ X \}$ and $M^i = K^i \cup \{ X \}$.

  If "b" is not present anywhere inside X :

  $M_s^i = X \subseteq K^i \cup X = M^i$.

  In any case, $M_s^i \subseteq M^i$.

- *AND :* As shown in the figure 4.5, let X be the root state of M. Let it be refined into a number of states X1, X2,..., Xk. For each Xi, let Pi be the machine with root state Xi. We know that for each Pi, $Pi_s^i \subseteq Pi^i$. As in the previous case, if X "has" the signal "b" somewhere inside it, our slicing algorithm will refine X. In that case :

  $M_s^i = \bigcup_{j=1}^{j=k} Pj_s^i \cup \{ X \}$ and $M^i = \bigcup_{j=1}^{j=k} Pj^i \cup \{ X \}$.

  In the case when X did not have "b" inside it:

  $M_s^i = \{ X \}$, and $M^i = \bigcup_{j=1}^{j=k} Pj^i \cup \{ X \}$.

  Hence, $M_s^i \subseteq M^i$.

Inductive Case : Here we show that $M_s^f \subseteq M^f$ for inputs of non-zero length. Assume that for inputs of length (l-1) the relation is satisfied. We must show that even after the i^{th} input has arrived, the same relation is satisfied. Assume that after an input of length (l-1) is consumed, the configuration of the machine M changes from M^i to N, and the configuration of the machine M_s changes from M_s^i to N_s. Once the i^{th} input has been absorbed, the configurations of M and M_s change to M^f and M_s^f respectively. We use the notation $\Rightarrow$ to denote a macro-step transition, and $\rightarrow$ to denote a micro-step transition. Hence :

$$M^i \Rightarrow^* N \Rightarrow M^f \text{ and } M_s^i \Rightarrow^* N_s \Rightarrow M_s^f.$$

We know that $M_s^i \subseteq M^i$ and $N_s \subseteq N$. The i^{th} input causes the transition $N^s \Rightarrow M_s^f$ and $N \Rightarrow M^f$. We look at the micro-steps that form part of the transition on the i^{th} input. Let after the i^{th} micro-step, the configuration be denoted by μ^i. Hence the sequence of micro-steps in the machine M looks like :

$$\xi = \mu^0 \rightarrow \mu^1 \rightarrow \mu^2 \rightarrow \ldots \rightarrow \mu^p \rightarrow \mu^{p+1},$$

where μ^0 stands for N, and μ^{p+1} stands for M^f. From ξ, we can construct a sequence ξ_s for M_s as follows :

- Let the first configuration in the sequence be N_s.
- For each μ^i, i$\in${1, 2, ..., (p+1)} define $\mu_s^i = \{ x \mid x \in M_s \wedge x \in \mu^i \}$
- Let $M_s^f = \mu_s^{p+1}$

Define $\xi' = N_s \rightarrow \mu_s^1 \rightarrow \mu_s^2 \rightarrow \ldots \mu_s^p \rightarrow \mu_s^{p+1}$.
Compress the sequence ξ' as follows :

- If the sequence contains the subsequence $N_s \rightarrow \mu_s^i$ and $N_s = \mu_s^i$ for some i, then replace $N_s \rightarrow \mu_s^i$ by N_s.
- If the sequence contains the subsequence $\mu_s^i \rightarrow \mu_s^j$ then replace the subsequence by μ_s^i.

The final sequence, after transformation is ξ_s. Let :
$\xi_s = N_s \rightarrow \nu^1 \rightarrow \nu^2 \rightarrow \ldots \rightarrow \nu^k \rightarrow \nu^{k+1}$. ($\nu_s^i = \mu_s^j$ for some j, and $\nu^{k+1} = M_s^f$).

Clearly, the constraint $M_s^f \subseteq M^f$ has been met by the sequence 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 a part of the sequence ξ' : $\mu_s^i \rightarrow \mu_s^{i+1}$ ($\mu_s^i \neq \mu_s^{i+1}$) that is triggered by a local signal "c". Clearly the transition $\mu^i \rightarrow \mu^{i+1}$ in ξ is also triggered by "c". Hence, since ξ is a legal sequence of micro-steps for M, there exists j < i such that $\mu^j \rightarrow \mu^{j+1}$ is a change in configuration caused by a transition that outputs "c". Clearly then, there is a trigger edge from the transition corresponding to the configuration change $\mu^j \rightarrow \mu^{j+1}$ to the transition corresponding to the configuration change $\mu^i \rightarrow \mu^{i+1}$ in both the machine M and the machine M_s . Therefore, the slicing algorithm would include the transition corresponding to the configuration change $\mu^j \rightarrow \mu^{j+1}$ in the machine M_s . Hence the configurations μ_s^j and μ_s^{j+1} in ξ' will be distinct. Hence, even in the sequence ξ', if a transition is triggered by "c", there will be a transition that occurs earlier in the sequence and outputs "c". The same property is followed in the sequence ξ_s as well. Hence ξ_s is also a legal sequence. Thus, lemma 1 is proved.

Lemma 2 : The machine M generates the output "b" on an input I^* if and only if the machine M_s generates the output "b" on the same input I^* .

Proof : The proof proceeds by induction on the length of the input sequence I^* . The base case is when the input sequence is of length 0. In this case, both Machine M and machine M_s do not generate the output signal "b". Hence the base case is proved. Consider that the lemma is true for all input sequences of length (l-1). We will show that on the i^{th} input the machine M will generate "b" if and only if the machine M_s generates "b" on the i^{th} input. For this purpose, assume that the configurations of the machines M and M_s after absorbing the inputs of length (l-1) are N and N_s respectively. Let the final configurations after absorption of I^* be M^f and M_s^f respectively.

($\Leftarrow$) : Suppose the machine M_s generates the output "b" on the i^{th} input. Let this output "b" be generated as a result of a transition between two states A and B. Clearly A$\in N_s$ and B$\in M_s^f$. Since $N_s \subseteq$ N and $M_s^f \subseteq M^f$ (from Lemma 1), A$\in$N and B$\in M^f$. Hence, the same transition will be taken on the machine M, and so the output "b" is generated on the i^{th} input.

($\Rightarrow$) : Suppose the machine M generates the output "b" on the i^{th} input. Let this output "b" be generated as a result of a transition between two states A and B. Clearly A$\in$N and B$\in M^f$. If A$\in N_s$, then the same transition will be enabled in M_s as well. Hence the output "b" is generated in M_s as well. It is not possible that A$\notin N_s$ since $N_s \subseteq$ N, and all transitions which generate "b" as output are included in M_s by the algorithm.
Hence the lemma 2 is proved. Lemma 2 is precisely the statement of correctness that proves the correctness of M_s as a slice of M.

4.8 Is the slice minimal ?

There are two aspects desired of any slicing algorithm : that of correctness, and that of producing minimal slices. We have already shown the correctness of the algorithm for a special case, we now come to the notion of minimal slice. A minimal slice M_s of a machine M would be a machine such that removal of any state from the M_s will result in a machine that does not conform to the definition of the slice. It can be seen from the example given in figure 4.4 that the algorithm given does not produce minimal slices, as in the example, we could remove the states B and C from the final machine - and yet be left with a machine that obeys the definition of a slice. Consider also figure 4.6. Here we could have included in the slice (with respect to $< X, b >$) just the states A and B. However, our slicing algorithm includes all the siblings of A and B.

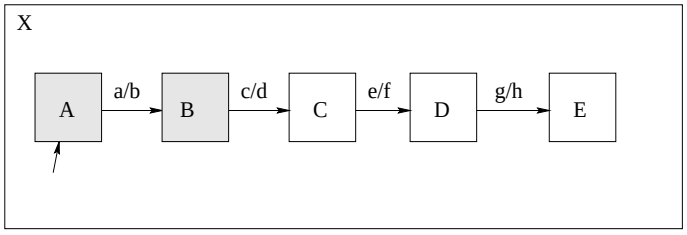


Figure 4.6: Non-minimal slices produced by the slicing algorithm.

4.9 Extending the algorithm - Slicing CRSMs

Now that we have an algorithm that slices Argos programs, we can attempt to slice CRSMs using ideas similar to the ones employed in the slicing of Argos. We need to arrive at a slicing criterion and define what we mean by the slice of a CRSM.

4.9.1 The slicing criterion

CRSMs are usually used in the specification of distributed controllers. Each of the distributed components communicates with the others by means of channels. The input and outputs of each component are not visible to the other components.

The slicing criterion that we chose was very similar to the one chosen for Argos. A CRSM is typically a set of machines $< M_1, M_2, M_3, \ldots, M_n >$. The slicing criterion is $< M_i, S, b >$, where i$\in\{1, 2,\ldots, n\}$, and S is a state (not a rendezvous state) in M_i. b is the signal whose behavior we want to retain in the slice of the CRSM. It must however be noted that b is a signal that is local to only the machine M_i. So when we slice the CRSM with respect to this slicing criterion, we will be studying only the local properties of the machine M_i.

4.9.2 Definition of a slice of a CRSM

The slice of a CRSM $N_1||N_2||\ldots||N_m$ with respect to a slicing criterion $< N_i, S, b >$, i$\in\{1, 2,\ldots, m\}$ is defined to be a CRSM $M_1||M_2||\ldots||M_q$, where each M_i is associated with a unique N_j by a function f :$\{1, 2,\ldots, q\} \rightarrow \{1, 2,\ldots, m\}$ such that j = f(i), and
$(\forall I_1^* I_2^* \ldots I_m^* \in$ Reside (N_k, S, I_k^*)):(Reside (M_p, S, I_k^*) $\wedge$ ($N_k[I_k^*]$/b = $M_p[I_k^*]$/b))
where f(p) = k. The Reside predicate has the same semantics as it had in the case of Argos - however there is a difference. In this case, it is not just the machine N_k running in isolation. It may have works with other machines that

are independent and there may be points where two machines may synchronize by channel communication. That is the reason why we did not quantify over just the input sequence I_k^*, but quantified over the set of input sequences that are run over each distributed component. In the slice, a machine M_a will be fed with the input sequence I_b^* where f(a) = b.

4.9.3 The Slicing Algorithm for CRSMs

The idea used in slicing a CRSM with respect to a criterion $< N_k, S, b >$ is very similar to the idea used in Argos. We first consider the machine N_k. We then convert N_k to an Argos program P by considering the rendezvous states as "normal" states and slice P with respect to the criterion $< S, b >$ by using the Argos slicing algorithm. This operation will include all transitions in N_k which generate b, and also all transitions from which these can be reached. However, since channel communication is present in CRSMs, we must take care of channel dependencies. Exit from a rendezvous state can occur only when the channel communication (which involves handshaking) is complete (barring the case where we have pre-emption). Thus, if the Argos slicing algorithm includes in the slice a rendezvous state, say A! from the machine N_k, then we must look for the rendezvous state A?, from some other machine N_l and include it in the slice of the CRSM. In addition to this, we must also include in the slice all those transitions in N_l from which we can reach the state A?. This procedure continues till no more rendezvous states can be added to the slice.

Algorithm 7 CRSM-slice ($N_1||N_2||\ldots||N_m, < N_k, S, b >$)

CRSM-slice := ϕ
for all (N_j such that 1≤j≤k) **do**
  Draw-Trigger-Edge(N_j)
end for
Call Draw-Rendezvous-Dependency-Edges ($N_1||N_2||\ldots||N_m$)
CRSM-slice := CRSM-slice ∪ **Call** Argos-Slice(N_k, $< S, b >$)
while (∃r | r is a rendezvous dependency edge between A and B ∧ A ∈ CRSM-slice ∈ B ∉ CRSM-slice) **do**
  Call Rendezvous-Slice (N_j, B) {Here B is a state in the machine N_j}
  Call CRSM-Trigger-Edge-Slice(N_j)
end while
Return CRSM-slice

4.9.4 Explanation of the procedures used

The procedure *Draw-Rendezvous-Dependency-Edges* draws trigger edge like dependencies between a pair of channels. The difference here is that the dependency is undirected, because if a channel is included, it means that its counterpart also has to be included to complete the handshake to complete communication. The procedure *Argos-Slice*(N_k, $< S, b >$) is a procedure that works exactly like the slicing algorithm for Argos programs (Algorithm 1). The difference here is : in CRSMs, channels can also appear as states (rendezvous states). Algorithm 1 does not consider such a case. Hence we define procedure Argos-Slice(N_k, $< S, b >$) which does not differentiate between rendezvous states and ordinary states, and includes both in the set that is returned as the output. *Rendezvous-Slice* (N_j, B) works as shown in Algorithm 8. The procedure *CRSM-Trigger-Edge-Slice(N_j)* is very similar to the Algorithm 6, but differs in that it does not differentiate between normal states and rendezvous states. The procedures *Rendezvous-Slice* and *CRSM-Trigger-Edge-Slice*, together algorithm 7, include in the slice all those transitions from N_j from which the rendezvous state B is reachable.

Algorithm 8 Rendezvous-Slice (N_j, B)

```
i := B
while (i != root(N_j)) do
    CRSM-slice = CRSM-slice ∪ {all siblings of i (rendezvous and normal)}
    i := parent(i)
end while
```

Algorithm 9 CRSM-Trigger-Edge-Slice (N_j)

```
List-Trigger := {t | t is a Trigger Edge in machine N_j}
while (∃(t ∈ List-Trigger) | Target-Transition(t) ∈ CRSM-slice ∧ (Source-
state(t) ∉ CRSM-slice)) do
    Call CRSM-Top-To-Bottom (<Highest-Possible-Source-State(t), b>)
end while
```

4.10 Summary

This completes the discussion on the slicing of CRSMs and Argos. The algorithm that slices CRSMs is not a minimal algorithm. It can however be improved by using the idea suggested in [NR00], that interference dependence

Algorithm 10 CRSM-Top-To-Bottom($< S, b >$)

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

is not transitive. We can draw a parallel between concurrent programs and CRSMs by arguing that each thread of a concurrent program is like a node in the CRSM, and that rendezvous dependency edges are like the interference dependence edges. Then a slicing algorithm for CRSMs on the lines given in [NR00] will give smaller slices.

Chapter 5

Slicing Esterel

In this chapter we consider the problem of slicing Esterel [Ber00a], [BG92], [Ber00b]. We extend the traditional tools used in program slicing, such as *Control Flow Graphs (CFGs)* and *Program Dependence Graphs (PDGs)*, to slice esterel programs.

5.1 The Esterel Syntax and its intuitive semantics

In the discussion that follows we will limit ourselves to Pure Esterel, where the information carried by a signal is limited to its presence/absence status. In full Esterel, signals can carry values of arbitrary types as well. A Pure Esterel program (module) has an input-output interface and an executable body as shown :

`module M;`
`input` *names;*
`output` *names;*
statement
`end module`

An input event specifies the presence/absence of a input signal. The program reacts by computing and output event : where it assigns a status to each output signal. The reaction - as in other synchronous languages - is instantaneous and a reaction is called an instant.

5.1.1 The Language

The kernel language that we will be using is :

```
S ::= nothing
```

```
emit S
pause
present S then p else q end
suspend p when S
p; q
loop p end
p || q
trap T in p end
exit T
signal S in p end
```

';' binds tighter than '||'. '[' and ']' can be used to group statements in arbitrary ways. Both the `then` and the `else` parts in the `present` clause are optional. All other constructs in Esterel are derived from these constructs.

5.1.2 Intuitive Semantics

The status of an input signal is determined only by input events and explicit emission of an input signal using `emit` is disallowed.

- `nothing` does nothing and terminates immediately.
- `pause` pauses for one reaction and terminates at the next.
- `emit S` instantaneously broadcasts the signal S, and terminates immediately. The emission is valid for that instant only.
- `present S then p else q end` starts p immediately if S is present in that instant, otherwise it starts q.
- `suspend p when S` : Here "S" is a *guard* statement. The guard controls the execution of the statement p in every instant except the first. If p does terminates or exits, then so does the entire suspend statement. If p has paused at the first instant, then as long as p remains active the guard signal is tested for its presence. If S is present, the p is not executed and is kept unchanged till the next instant. If S is not present, then p receives the control for that instant.
- `p; q` : Here the control flows to p and the statement behaves like p as long as p is active. After that the statement behaves like q, except in the case when p exits a trap, when the statement q is discarded.

- `loop p end` immediately starts the body p. When p terminates (if it does), it is restarted again. The body of a loop statement is not allowed to terminate instantaneously when it is started.

- `p||q` immediately starts p and q in parallel. The statement is active as long as one of its branches is active. When any branch exits a trap, then entire statement exits the trap. If two statements exit distinct traps at the same instant, then the parallel statement exits the outermost of the traps.

- `trap T in p end` defines a lexically scoped exit point for p. The trap statement starts the body p and behaves as p till it terminates or it executes an exit statement.

- `exit T` statement instantaneously exits the trap T. The corresponding trap statement is terminated unless an outer trap statement is also concurrently exited.

- `signal S in p end` immediately starts the body p with a fresh signal S overriding one that may already exist.

5.2 The Inadequacy of the Traditional Approach

In traditional program slicing, we first construct the CFG of a given program, and from there, we construct the PDG by observing various kinds of dependencies such as control dependency, data dependency etc. The PDG is very well defined for traditional programming language constructs such as `if` statements, `while` statements etc. The problem in slicing Esterel using the traditional method is that these concepts such as the CFG, PDG etc. are not defined for the constructs in Esterel. In fact, the definition of control dependency that we have proposed for Esterel is different from the traditional definition given in the literature. We will also introduce new kinds of edges called time dependency edges while drawing the PDG. We will however stick to the traditional back propagation algorithm for slicing.

5.3 Slicing Criterion and Definition of a Slice

The slicing criterion we chose for an esterel program was <**line number, signal**>. The line number is an optional information and can be avoided as

well, in which case, we use the same criterion with the line number getting the value ϕ.

Suppose S is an esterel program, and I_i is the input vector at the i^{th} instant, ie., it tells the absence/presence status of the inputs at the i^{th} clock tick. Suppose $I^* = \cup_{i \geq 0} I_0 I_1 \ldots I_i$. We give a different definition of a slice for each of the slicing criteria, ie. where we have a line number in the slicing criterion, and where we do not. An esterel program S', that is a subset of S is considered to be a slice of S if it satisfies the following definitions :

For slicing criteria of the form $< \phi$, b$>$:

$\forall I^*$: (O(S, I^*)/b = O(S', I^*)/b)

For slicing criteria of the form <line-number, b> :

($\forall I^* \in$Possible-to-Reach(l, S, I^*)) : (O(S, I^*)/b = O(S', I^*)/b)

The predicate Possible-to-Reach(l, S, I^*) is TRUE for all input sequences I^* which cause the control to reside at a point m in the program, and there is a valid execution path of the program such that it is possible to reach l from mi.

5.4 Outline of the Slicing Algorithm

- Make the CFG of the given program.
- Draw the dependency edges - these are of various kinds : control dependence, time dependence, loop carried time dependence, trap-exit edges and signal dependence.
- Draw the PDG
- Now given the slicing criterion, just trace back the path using the dependency edges

In this project although we have not formally proved that this approach works, we have tried it out for a number of examples and it seems to work. We have however given an exact definition of CFGs and the dependency edges, and shown how to draw the PDG.

5.5 Control Flow Graph (CFG)

A control flow graph of an esterel program shows how control flows from one construct to another. The flow of control through each esterel construct is shown in the figure 5.1 Note that each of these structures is a black box with a

single entry edge and zero or more exit edges. In the figure, **En** edges are the entry edges into the structure, **Ex** edges are the normal exits from the structure and **TrEx** are the trap exits. The Trap-Exit structure however requires some explanation. **Ex1** edges correspond to the normal exits from the program body *p*. **TrEx2** corresponds to exit point from *p* corresponding to the *exit T* statement, **Ex1** corresponds to the normal exits and **TrEx1** corresponds to trap exits from *p* other than those corresponding to *exit T*. For the suspend clause however, a different procedure is followed. Suppose the statement is `suspend p when S`, we break up *p* into a number of pause free blocks B_1, B_2, B_3, ... as shown in the figure 5.2. These blocks may be arbitrarily interconnected with each other. The `suspend p when S` structure is then replaced by the structure shown in figure 5.2. The interconnections between the pause free blocks are retained in the new structure as well.

We draw a dummy node called `start`. An edge leading out of this node is connected to the entry point of the first node of the program. We also have a dummy node called `finish`. The exit point from the last node in the program is connected to this node. This completes the specification of the CFG.

5.6 Dependency Edges and the PDG

Now that we have described the CFG construction, we describe the construction of the PDG. The PDG is constructed by adding various kinds of dependency edges to the CFG. All the analysis to obtain the dependency edges will now be based on the properties of the CFG. We describe the various kinds of dependency edges and also describe how to go about drawing them.

5.6.1 Control Dependency Edges

In Esterel, we define a control structure to be one of the following nodes from the CFG : `start`, `present X` (for some signal X), `trap T`, `loop`, `co-begin`, $\texttt{Theta}_i$. The end of these control structures in the CFG is denoted by the nodes `finish`, `end`, `end`, `end`, `co-end`. We do not have any explicit structure to denote the end of the $\texttt{Theta}_i$ nodes since when the threads end, the do so directly into the `co-end` node, and hence that serves as the control structure to denote the end of a thread. A node j is control dependent on node i if :

1. There is a path $i \rightarrow^* j$ in the CFG.

2. i is a control structure.

3. There are no more control structures or ending of control structures on the path from i to j.

If a node j is control dependent on a node i, in the CFG, we draw a control dependency edge from node i to the node j.

5.6.2 Loop-Exit edges

These edges are drawn from `exit` nodes to `loop` nodes. We have identified the following condition for which these edges must be drawn. Consider the structure :

```
trap T in
    <stmt>
end
```

A loop-exit edge is drawn from *every* `exit T` node in `<stmt>` to *every* `loop` node in `<stmt>`.

5.6.3 Time Dependency edges

These edges are drawn from a `pause` node to another node. A node j is time dependent on a node i (which must be a `pause` node) if :

1. There is a path $i \rightarrow^* j$ in the CFG.
2. Node j is the post-dominator of node i.
3. There are no more `pause` statements on the path from i to j.

5.6.4 Loop Carried Time Dependency Edges

These occur only if there is a loop in the program. In this case, we have a path from a node i to a node j, where i and j are nodes within a loop, then we have a path from the node j to the node i as well. Hence we expand out the loop once, and examine the resulting structure for time dependence. The new edges that are added are called loop carried time dependence edges.

5.6.5 Signal Dependency Edges

These edges are drawn from an `emit M` statement to a `present M` statement. The edge on a `present M` tells us which `emit` nodes can potentially affect the presence of M when the control of the program is at that `present` statement.

Thus they are drawn only when the `present` and `emit` nodes are on different threads of a parallel construct. A naive way to draw these edges would be to draw an edge from an `emit B` statement on one thread to a `present B` statement in every parallely working thread.

We can however do better with a little more analysis. Break up each thread into a number of pause free blocks, and number them as shown in the figure 5.3. We can construct a graph with A_i's and B_i's as nodes. An edge is drawn between two nodes A_m and B_p in this graph if and only if an `emit` statement in block A_m can affect the execution of a `present` statement in block B_p, or an `emit` statement in block B_p can affect the execution of a `present` statement in block A_m. Now suppose we have an `emit M` statement in block A_i. We draw signal dependency edges from this `emit M` node to all those `present M` statements in blocks B_j such that B_j is adjacent to block A_i in the graph we constructed. As an example, we consider the program given below. In figure 5.3 we have given a portion of the CFG and have shown the pause free blocks and the signal dependency edges. If we had used the ad-hoc method, we would have also drawn the signal dependency edge between the `emit N` in B_2 and the `present N` in A_1.

```
module example:
inputoutput N, I;
output M, J, A, B;
[
loop
    emit M;
    present N then emit I else emit J end;
    pause;
    emit I;
    pause;
end
||
loop
    present I then emit A else emit B end;
    pause;
    emit N;
    pause;
    emit N;
    pause;
```

```
    present I then emit B else emit A end;
    pause;
end
]
end module
```

5.6.6 The slicing algorithm

Now to slice the program, we just trace back the path from the node of interest to the `start` node along the dependency edges we just drew, and include all the nodes we encounter into the slice. If there was a dependency edge from node i to node j, and node j was included in the slice, node i also gets included in the slice. This procedure continues till no more nodes can be included in the slice. If we used a slicing criterion of the form $< line - number, signal >$, then we would start of with just one node in the slice and trace back the path to the `start` node. If however, we used a slicing criterion of the form $< \phi, signal >$, then we would have to include all the `emit signal` nodes in the slice, and repeat the procedure for each of these nodes.

5.7 An Example

Consider the car controller program shown below. We have not shown the CFG of this program, however the PDG is shown in figure 5.4. We slice the program with respect to the signal *control_throttle*. The statements marked <—— are not included in the slice. These correspond to the shaded statements in the PDG. Note that if we just include the statements from the PDG we do not get a syntactically correct program. Some amount of post-processing is required to get correct programs.

```
module car_controller :
input ignition_on,ignition_off,accel;
input door_opened, door_locked;
output alarm,control_throttle,door_lock;

loop
    trap ABORT1 in
        [
            trap AWAIT1 in
                loop
```

```
                pause;
                present ignition_on then exit AWAIT1 end;
            end
        end;
        emit door_lock;                    %   <-----
        loop
            trap AWAIT2 in
                loop
                    pause;
                    present door_locked then exit AWAIT2 end;
                end
            end;
            trap ABORT2 in
                [
                    loop
                        pause;
                        present accel then emit control_throttle2 end;
                    end
                ||
                    loop
                        pause;
                        present door_opened then exit ABORT2 end;
                    end;
                ]
            end;
        end;
        emit alarm;                        %   <-----
        emit door_lock;                    %   <-----
    ||
        loop
            pause;
            present ignition_off then exit ABORT1 end;
        end;
    ]
  end;
end;
end module
```

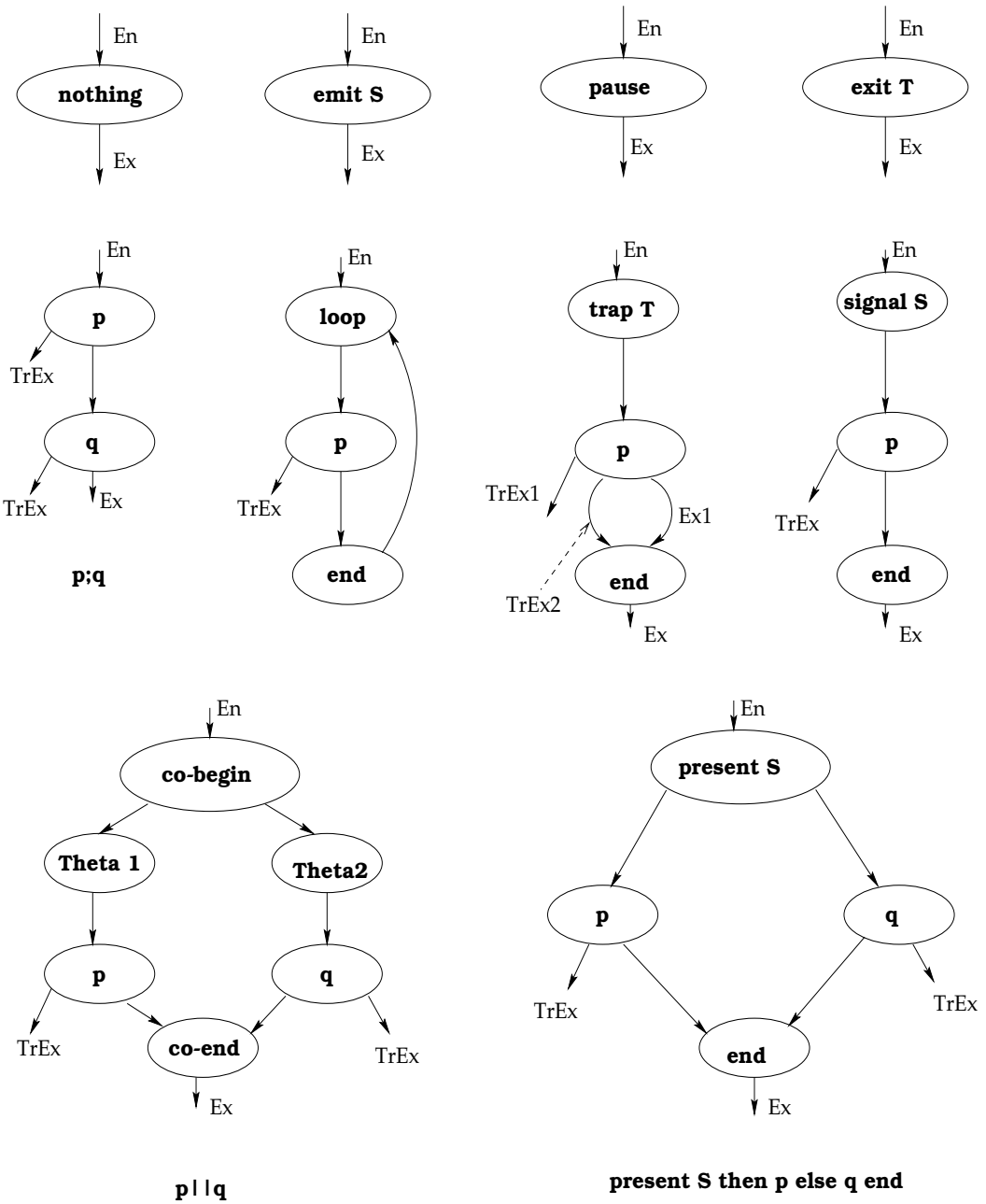


Figure 5.1: Control Flow in Esterel Constructs

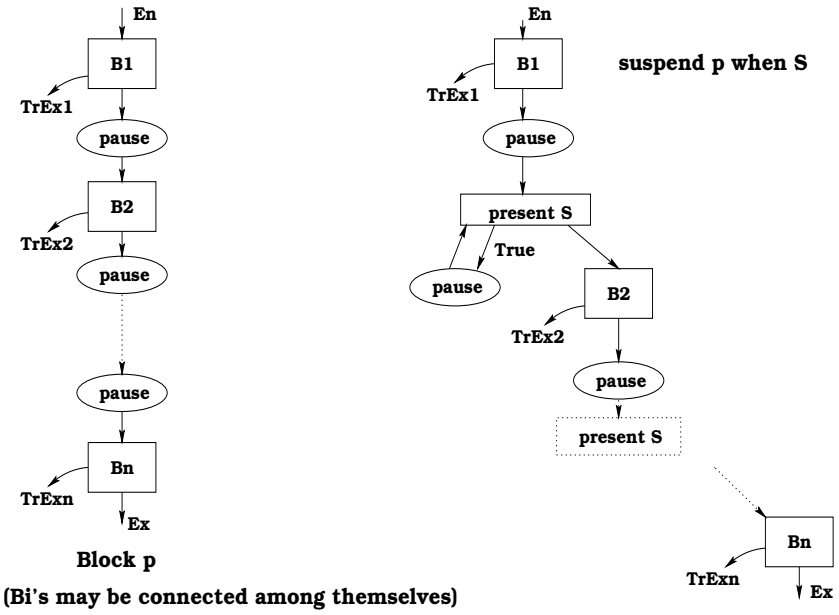


Figure 5.2: The Suspend Statement

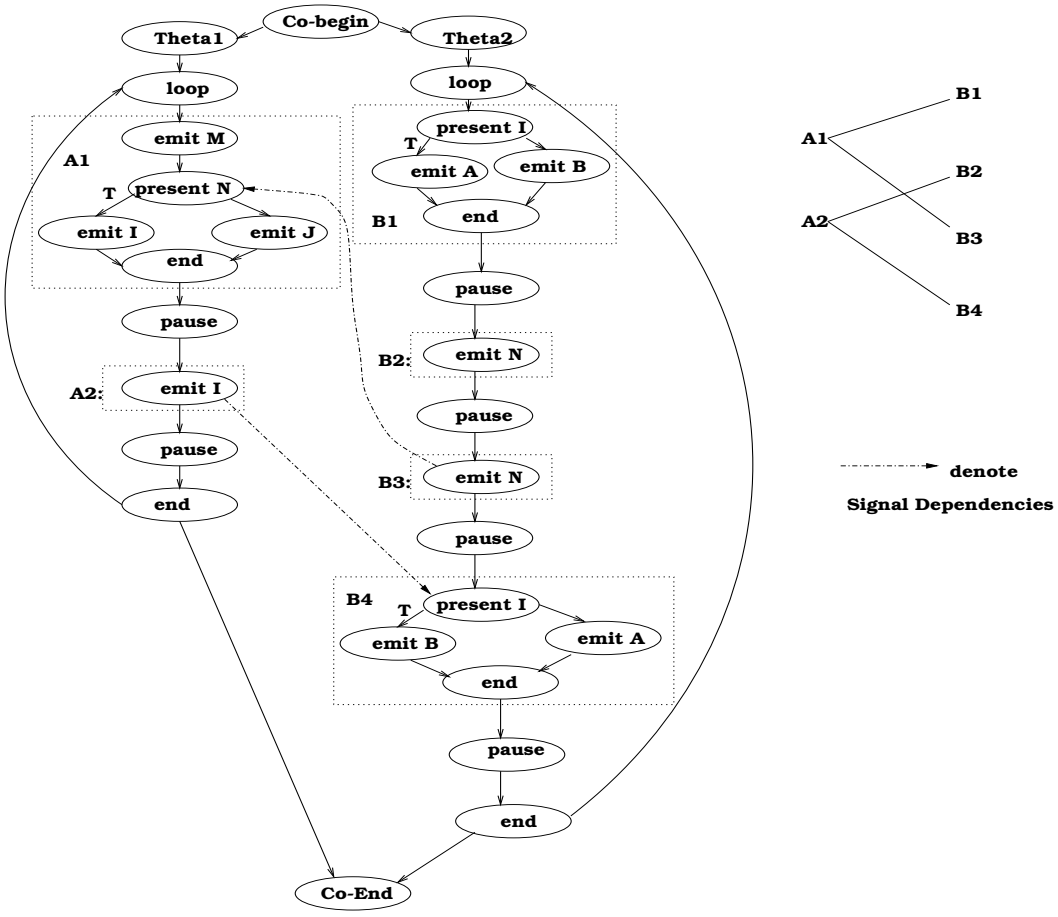


Figure 5.3: Signal Dependency Edges

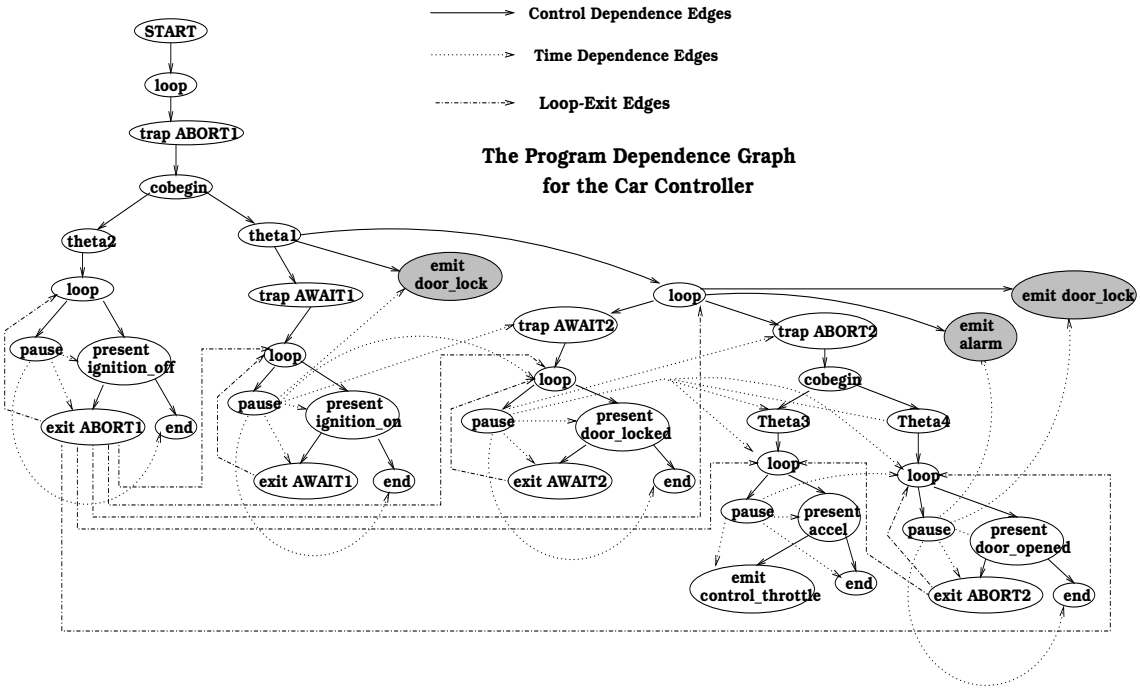


Figure 5.4: Slicing the Car Controller

Chapter 6

Summary

6.1 Synopsis

This project has covered in detail the problem of slicing CRSMs, Argos and Esterel. A slicing algorithm was given for Argos and was proved to produce correct slices. This algorithm was extended to give a slicing algorithm for CRSMs. For Esterel, the definitions of the CFG and the PDG, have been formalized.

6.2 Future Work

The following work remains to be done :

- The slicing algorithm that has been proposed for Argos is not minimal. It can be extended/modified so that minimal slices are produced.
- A formal proof of correctness of the slicing algorithm for CRSMs has to be given.
- We have not precisely characterized what we mean by pause free blocks in the definition of the PDG for esterel programs. A formal definition of these pause free blocks on the lines of *basic blocks* is awaited.
- A proof of correctness of the slicing algorithm for Esterel programs also remains to be given.

Bibliography

[Ber00a] G. Berry. The constructive semantics of Esterel. In *Draft Book, current version 3.0*, 2000.

[Ber00b] G. Berry. The Esterel v5 Language Primer Version v5-91. Sohpia-Antipolis, June 2000.

[BG92] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language : Design, Semantics and Implementation. In *Science of Computer Programming 19(2)*, pages 87–152, 1992.

[Hal93] Nicholas Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, IMAG Institute, Grenoble, France., 1993.

[Har87] David Harel. Statecharts : A Visual Approach to Complex Systems. In *Science of Computer Programming.*, pages 231–274, 8(3) 1987.

[HN95] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. 1995.

[Hol97] Gerard. J. Holzmann. The model checker SPIN. In *IEEE Transactions on Software Engineering*, May 1997.

[MR] Florence Maraninchi and Yann Rémond. Argos : An Automaton Based Synchronous Language. Paper obtained on communication with the author.

[NR00] M. Gowri Nanda and S. Ramesh. Slicing Concurrent Programs. In *ACM SIGSOFT International Conference on Software Testing and Analysis (ISSTA 2000)*, August 2000.

[Ram98] S. Ramesh. Communicating Reactive State Machines : Design, Model and Implementation. In *IFAC Workshop on Distributed Computer Control Systems, Pergamon Press*, September 1998.

[Ram00] S. Ramesh. Refinement and Efficient Verification of Synchronous Programs. In *IFAC Workshop on Distributed Control Systems, Pergamon Press*, December 2000.

[TM98] Teitelbaum Tim and Linda I. Millett. Slicing Promela and it's applications to Model Checking, Simulation and Protocol Understanding. In *Proceedings of the SPIN 98 workshop*, pages 75–83, Paris, France, November 1998.

[TM99] Teitelbaum Tim and Linda I. Millett. Channel Dependence Analysis for Slicing Promela. In *Proceedings of the International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 52–61, Los Angeles, California, May 1999.

[Wei84] Mark Weiser. Program Slicing. In *IEEE Transactions on Software Engineering*, pages 352–357, July 1984.