Generating Configuration Confirming Sequence with ObjectGeode Simulator

Zi Jun HU

School of Computer Science
McGill University, Montreal

Directed by

Prof. Alexandre Petrenko
Prof. Monroe Newborn

July, 1999

A thesis submitted to the Faulty of Graduate Studies and Research,
in partial fulfillment of requirements for the degree of
Master of Science

0-612-64373-5

Abstract

Test cases are very useful in industry. They can be generated for various purposes. In this thesis the test case is generated for configuration conformance in the Extended Finite State Machine (EFSM) model. This kind of test case is called Configuration Confirming Sequence (CCS).

Test case generation has been studied for years and fruitful results have been produced [1]. A number of approaches for automatic test case generation have been studied, one of them is based on the reachability analysis in the form of formal specification of the Extended Finite State Machine of the system under test. This thesis adopts such an approach for CCS generation, but with some extensions — after producing all separating sequences for the configuration by reachability analysis, a selecting program is executed to choose out proper separating sequences as CCSs.

Today formal description techniques (FDT), including Specification and Description Language (SDL) are widely used in specification and analysis of the EFSM model. A FDT system is a formal, unambiguous, hierarchical description of a set of EFSM models. Various FDT toolsets are now commercially available. Proper use of them can make test case generation simpler. In this thesis, ObjectGeode SDL is selected as a tool for system design and reachability analysis.

In this study, the method of CCS generation from an SDL specification is presented first. Then, to demonstrate the correctness of this method, a CCS generator for the Responder process in INRES protocol is designed and implemented, and results of the experiments are presented. A theoretical proof of correctness is also presented. We conclude by pointing out possible future work.

Résumé

La génération de cas d`épreuve (testcases) a été étudiée depuis plusieursannées et des résultats fructueux ont été produits [1]. Les cas d`épreuve peuvent être produits pour les plusieurs buts. Dans cette thèse, le cas dépreuve est produit pour la conformance de la configuration dans le modèle de la Machine de l'État Finie Étendue (EFSM). Ce genre de cas d'épreuve est défini comme Configuration qui Confirme la Séquence (CCS) dans [5].

Plusieurs approches pour générer automatiquement des cas d'épreuve ont été étudiées. Une d'elles est basés sur l'analyse de l`atteinte de la spécification formelle du modèle EFSM du système étudié. Cette thèse adopte une telle approche pour la génération de CCS, mais avec quelques extensions — après avoir produit, par analise formelle, des séquences de séparation pour la configuration, un programme de sélection est exécuté pour choisir les séquences de séparation adéquates (comme CCSs).

Aujourd'hui, les techniques de description formelles (FDT), y compris le Langage de Description de Spécifications (SDL), sont utilisées dans la spécification et l'analyse du modèle EFSM. Un système FDT est une description formelle, non équivoque, hiérarchique d'un ensemble de modèles EFSM. De nombreuses librairies FDT sont commercialement disponibles. L'usage adéquat de celles-ci permet la génération de cas d'épreuve plus facilement. Dans cette thèse, ObjectGeode SDL a été choisie comme la FDT pour le design et l'analyse du système.

Dans cette étude, la méthode de génération CCS utilisant la spécification SDL est présentée en premier lieu. Ensuite, un générateur CCS pour le 'Responder' du protocole INRES est développé pour vérifier l'efficacité de cette méthode. Finalement, les résultats des expériences sont présentés. Nous concluons en signalant des pistes de travaux futurs.

Acknowledgments

I would like to express my deep gratitude to Prof. Alexandre Petrenko who supervised me in this research. Without his academic advice my research would be impossible. His numerous research papers lead me to the interesting field of automatic testing. He influences me not only by his knowledge but also gentle personality.

I also thank Prof. Monroe Newborn as my second supervisor. His suggestions were very helpful for me to finish the thesis.

My gratitude should also be given to Dr. Sergiy Boroday. The discussions with him help me clear many issues. His comments assist me to shape my thesis into its current form. Thanks to his profound knowledge in formal methods, he is never failed by any challenging problem risen from this thesis research.

Finally, a financial support of CRIM is greatly acknowledged.

Contents

Chapter 1

Introduction

1.1. Motivation

It is widely acknowledged that the model of Extended FSM (EFSM) is a very powerful model for verification and test derivation. There exist a number of tools that support development activities around specifications based on the EFSM model. In particular, the commercial tools supporting SDL now offer test case generation facilities. Such tools may resorts to reachability analysis to compute tests that cover transitions of the EFSM and provide test preambles to reach specific configurations of the EFSM enabling the transitions to be tested. However, they currently do not check the tail state of transitions or the configurations reached after a test. This casts doubt on the confidence that the tests have really assessed the corresponding behavior of the tested machine (known as IUT or Implementation Under Test), let alone that they would reach a significant coverage of the faults in tail states and configurations [Part I, 12].

This problem can be abstracted as configuration distinguishability for the EFSM model that has been studied in [5]. The authors of [5] introduce a new concept — Configuration Confirming Sequence, or CCS. Generally speaking, CCS is an input sequence that can distinguish a state or configuration from a set of other states or configurations. Given this concept, the question on how to generate CCS automatically becomes an interesting problem. In this thesis, a CCS generator with ObjectGeode toolset is developed and studied.

1.2. Thesis Contributions

The existence of CCS is based on configuration distinguishability. The problem of configuration distinguishability for the EFSM model has been investigated in [5]. Given an EFSM model, a configuration and an arbitrary set of configurations, determine an input sequence such that the EFSM in the given configuration produces an output sequence different from that of the configurations in the given set or at least in a maximal proper subset. Such a sequence can be used in a test case to confirm the destination configuration [5].

Authors of [5] demonstrate that the problem of configuration distinguishability problem could be reduced to the EFSM traversal problem, so that the existing methods and tools developed in the field of model checking can be applied to CCS generation. Based on this result, the thesis presents a CCS generating approach relying on exhaustive simulation provided by protocol verification tool ObjectGeode.

This thesis investigates a general way to construct a CCS generator with ObjectGeode. During the study we find a feature of ObjectGeode simulator that hinders correct generation of CCS, that is, the simulator normally explores just a single path, as a result some CCSs will be lost. The solution is to introduce a post-processing program using C++ to explore the paths that have not been explored in ObjectGeode. This results in a hybrid system of ObjectGeode and C++. The structure of CCS generator is illustrated in Figure 1.1.

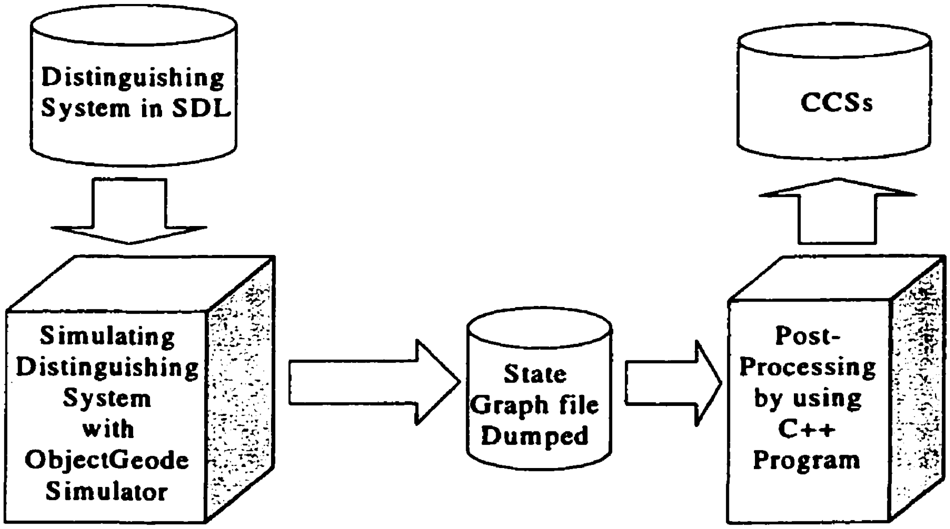

Distinguishing System in SDL
CCSs
Simulating Distinguishing System with ObjectGeode Simulator
State Graph file Dumped
Post-Processing by using C++ Program

Figure 1.1 CCS Generator

In post-processing two tasks will be fulfilled, one is generation of all minimal separating sequences (for an EFSM model, a minimal separating sequence is an input sequence that separates a configuration from another) from state graph dump file, another is selection of proper minimal separating sequences as CCS.

In this thesis, we demonstrate that the CCS generated in the method described above is correct and complete. Theoretical proof and positive results of the experiment performed with a benchmark protocol, called INRES, is also presented in the thesis.

1.3. Thesis Layout

The remainder of the thesis is organized in the following way.

Chapter 2 and Chapter 3 discuss the concepts of the EFSM model and protocol verification tool ObjectGeode, especially the exhaustive simulation performed by ObjectGeode simulator. Exhaustive simulation is tightly related to the reachability analysis, which is a crucial part of CCS generation.

Chapter 4 explains how a CCS can be generated with ObjectGeode simulator. First, we give a method to build an SDL model that is used in reachability analysis, this model is called a *distinguishing system.* Then we discuss the design and implementation of the post-processing program.

Chapter 5 demonstrates the applicability of the proposed approach to a communication protocol called INRES.

Chapter 6 concludes thesis and presents possible directions for further study.

Chapter 2

EFSM Model

2.1. Extended Finite State Machine

The model of a Mealy (finite state) machine extended with input and output parameters, context variables, operations and predicates defined over context variables and input parameters, is what is understood by an extended FSM in this thesis. The definition of EFSM model is given below:

Definition 2.1 [5]: An *extended finite state machine (EFSM) M* is a pair (S, T) of a set of states S and a set of transitions T between states from S, such that each transition t in the set T is a tuple (s, x, P, op, y, up, s'), where

- $s, s' \in S$ are the initial and final states of the transition, respectively;
- $x \in X$ is input, X is a set of inputs, and D_{inp_x} is the set of input vectors, each component of an input vector inp_x is an input parameter associated with x;
- $y \in Y$ is output, Y is a set of outputs, and D_{out_y} is the set of output vectors, each component of an output vector out_y is an output parameter associated with y;
- P, op, and up are functions defined on input parameters and context variables V, namely;
- P: $D_{inp_x} \times D_V \rightarrow \{\text{True}, \text{False}\}$ is a predicate, where D_V is the set of context vectors $\vec{v}$;
- op: $D_{inp_x} \times D_V \rightarrow D_{out_y}$ is an output parameter function;
- up: $D_{inp_x} \times D_V \rightarrow D_V$ is a context update function.

To define the operation of an EFSM, we first introduce some additional definitions.

Definition 2.2 [5]: Given input x and a (possibly empty) set of input vectors D_{inp_t}, a pair of input x and input vector from D_{inp_t} is called a *parameterized input.* A sequence of parameterized inputs is called a *parameterized input sequence.*

Similarly, we can define parameterized outputs and their sequences.

Definition 2.3 [5]: A context vector $\vec{v} \in D_V$ is called a *context* of *M*. A *configuration* of *M* is a pair of state *s* and context $\vec{v}$.

Note that in case of an empty set of context variables, which is the case for a pure FSM, a configuration coincides with a state. In this thesis configuration will be represented as a tuple $(S; p_1, p_2 ... p_n)$, where *S* is a state, and $p_1, p_2 ... p_n$ are context variables.

Definition 2.4 [5]: A transition is said to be *enabled* for a configuration and parameterized input if the transition predicate evaluates to true.

The EFSM operates as follows. The machine receives input along with input parameters (if any) and computes the predicate that is satisfied for the current configuration. The predicate identifies enabled transitions. A single transition among those enabled is fired. Executing the chosen transition, the machine produces output along with output parameters, which, if they exist, are computed from the current context and input parameters by the use of the output parameter function. The machine updates the current context according to the context update function, and moves from the initial to the final state of the transition. Transitions are atomic and cannot be interrupted. The machine usually starts from a designated configuration, called the *initial* configuration. A pair of an EFSM *M* and the initial configuration is called a *strongly initialized* EFSM [5].

To simplify the notations for transitions of EFSMs, we present a few conventions. Specifically, we normally use (*s*—*x, P/op, y, up*→*s'*) to denote a transition $t \in T$. If, in *t*, *P* is a True constant, *P* can be dropped from the transition. Similarly, when the transition does not change the context, the update function *up* can be omitted. At the same time, the output parameter function can only be absent when output *y* has no output parameters at all. Notations (*s*—*x, P/op, y, up*→*s'*), (*s*—*x / y, up*→*s'*), (*s*—*x / y*→*s'*) are examples of notations used for such situations. If present a transition, the update and output parameter functions can take forms of operations on separate variables, such as assignments. Figure 2.1 gives an example of a machine specified using these notations [Page17, 5].

It has four states and ten transitions that are labeled with two inputs *a* and *b*, three outputs *x*, *y* and *z*, the latter has a parameter, and four predicates.

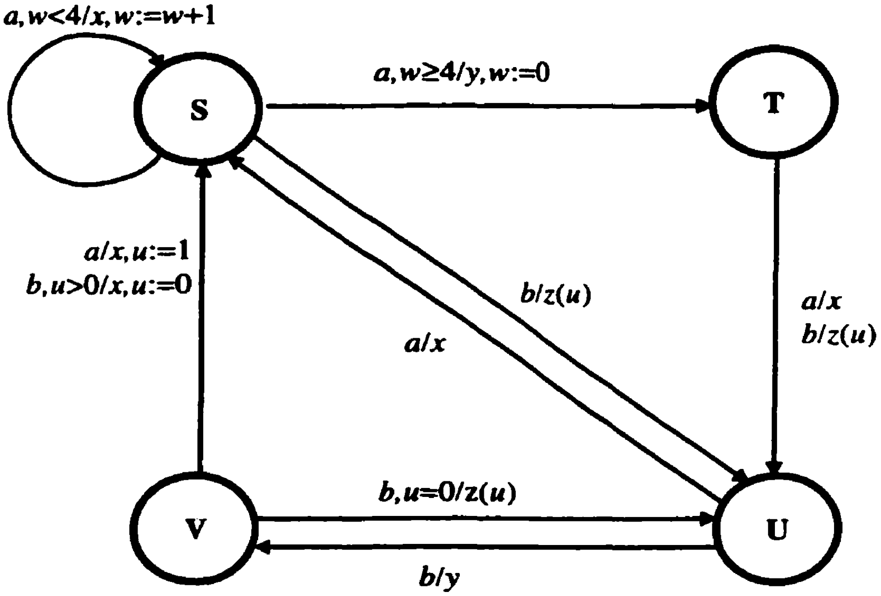


Figure 2.1 The EFSM Model *M*

Restricted EFSM (REFSM) is a type of EFSMs which meets a number of requirements.

Definition 2.5 [5]:

An *REFSM* is an EFSM meeting following requirements:

- *Consistent*: if for each transition *t*, every element in $D_{inp_t} \times D_V$ evaluates exactly one predicate to True among all predicates guarding transitions with the start state and the input of t. In other words, the predicates are mutually exclusive and their disjunction evaluates to true.

- *Completely specified*: if for each pair of state and input $(s, x) \in S \times X$, there exists at least one transition leaving state s with input x.
- *Deterministic*: if any two transitions outgoing from the same state with the same input have different predicates.
- *Observable*: if for each state and input, every outgoing transition with the same input has a distinct output.

Considering Figure 2.1, the machine has two context variables, u and w. It is consistent, completely specified, deterministic, and observable.

In this thesis, we study only REFSMs.

2.2 Configuration Confirming Sequence

The problem we are dealing with can be informally stated as follows. We know the configuration reached in the REFSM M in response to some parameterized input sequence applied to the initial configuration (this is the tail configuration of the test up to that point). Our goal is to determine a *single* parameterized input sequence that can increase our confidence in the correctness of the configuration reached in any implementation under test derived from M. To that end, we try to ensure that the correct configuration has been reached in the implementation, or at least that no suspicious configuration has been reached. Typically, we might allow an implementation to have different values from those specified for non relevant context variables, but pay special attention to crucial variables or the control state [5].

For a classical deterministic FSM, UIO sequence [4] is a solution to the problem. Assuming that faults in any implementation under test neither increase the state number nor mask each other, such a sequence can ensure correctness of the tail state of any transition once it is executed immediately after the transition. The problem of UIO generation was studied in a number of works. In case of the EFSM model, we are dealing with a more general problem, which we call here a configuration confirming sequence (CCS) generation. The key issue here is configuration distinguishability [5].

Finally, for practical reasons again based on experience in protocol testing (and on acceptability by test experts), we should try to find confirming sequences that are not "too long". Basically, it would not make sense to claim that a 100-input-long sequence would bring enough added confidence to justify appending it to test preambles of length 4 or 5, all the more so as a confirming sequence, to be fully trustable, has yet to be applied in all the other configurations, since faults may mask each other. Therefore, an arbitrary limit l on the length of the sequence will be set up for CCS generation [5].

We define distinguishing ability of configurations based on the following notion. Let M and N be two EFSMs defined over the same inputs and input parameters. We assume that the output alphabets of the two machines intersect, but the sets of output parameters associated with each common output in M and N are not necessarily identical.

Definition 2.6 [5]: Two parameterized outputs of M and N are said to be *compatible* if the output symbols coincide and very common output parameter has the same value in both parameterized outputs. Two parameterized output sequences, $y_1 \ldots y_k$ of M and $y'_1 \ldots y'_k$ of N, are *compatible* if for all $i = 1 \ldots k$, y_i and y'_i are compatible.

Based on this notation, we now define distinguishability of configurations.

Definition 2.7 [5]: Given a parameterized input sequence x, configuration c and c' of M are *distinguishable by* x if the parameterized output sequence that can be produced by (M, c) in response to x, is not compatible with any parameterized output sequence that can be produced by (N, c') in response to x. x is said to be a *separating sequence* that *separates* c from c'. Given the length $l = |x|$, configurations c and c' are said to be *l-distinguishable.*

Indistinguishable configurations of REFSMs are also referred to as *equivalent* configurations. Two REFSMs are *equivalent* if their initial configurations are equivalent.

Note that in this study, we assume that the given REFSM M may have indistinguishable (i.e., *equivalent*) configurations.

One should note that if an input signal in an input sequence is not acceptable by (M, c) but acceptable by (N, c'), then such an input sequence is a separating sequence. Because the input signal that is not acceptable produces, in fact, a NULL output signal by (M, c) which is different from all other outputs, the input sequence with this input signal is in definitely is a separating sequence.

The definition of Configuration Confirming Sequence immediately follows the Definition 2.7.

Definition 2.8 [5]: Given configuration c and a configuration set C of a REFSM M, a parameterized input sequence x is said to *confirm* c in the set C if x separates c from every $c^{'} \in C$ distinguishable from c.

Definition 2.9 [5]: Given configuration c and a configuration set C of a REFSM M, a parameterized input sequence x is a *Configuration Confirming Sequence (CCS) for configuration c and configuration set C* if x confirm c in the set C, or called *CCS for configuration c and configuration set C.*

Example. Consider Figure 2.1, assume the requirement is to find a CCS (if it exists) for configuration c=(S; 1,0) and configuration set a, that contains configurations (S; 2,4), (T; 0,0), (U; 0,0) and (V; 1,0). Here S, T, U and V are states in M, and (2,4) means u=2 and w=4.

Then the input sequence b is a CCS since different configurations give different outputs. (S; 1,0) outputs $z(1)$; (S; 2,4) outputs $z(2)$; (T; 0,0) outputs $z(0)$; (U; 0,0) outputs y; (V; 1,0) outputs x.

Chapter 3

SDL FDT

3.1. Introduction

SDL (Specification and Description Language) is an FDT (Formal Description Technique) promoted by ITU Z.100 [13]. Extended finite state machine can be specified formally and unambiguously in SDL and a set of such descriptions of the EFSM models is called an SDL system. The SDL system provides a solid base for automated analysis of EFSM model.

ObjectGeode is an FDT toolset that provides design and analysis facilities. SDL is its main description language. It provides a set of tools required at every step of system modeling, simulation, targeting and testing, as listed below [4]:

- Modeling tools, for analysis and design: Object Editor, MSC Editor, State Chart Editor, SDL Editor and SDL&MSC Checker.
- Simulation tools: SDL&MSC Interactive Simulator and SDL&MSC Exhaustive Simulator.
- Targeting tools: OMT C++ Code generator, SDL C Code Generator, SDL C Runtime Library.
- Testing tool: DesignTracer.

In this thesis, only SDL Editor, State Chart Editor and SDL Exhaustive Simulator are used.

There are two forms in which an SDL system can be represented in graphic form and textual form. Although they are different in appearance, they are same in semantics.

The following sections are a brief overview of SDL and ObjectGeode. For detail information, the reader can refer to [2][6].

3.2. Interpreting SDL

3.2.1. Static Concepts of SDL

Making a static description of a system amounts to defining its architecture. SDL uses a hierarchical structure for system specification. The description always begins with the *system* object, which is the object of the highest hierarchical level in the description. Creating a system entails creating a boundary between the system to be modeled and the exterior of the system (*environment*) [9].

The aim of an SDL system is to model a consistent set of communicating extended finite state machines grouped as *blocks*. Blocks, or subsystems, are the main conceptual components of the system. Blocks are arranged in a hierarchical structure. There is no limit to the number of hierarchical levels for block [9].

Only leaf block can contain processes. A *process* is an extended state machine describing a unit of dynamic system behavior [9].

Example: The following is an illustration of SDL system structure.

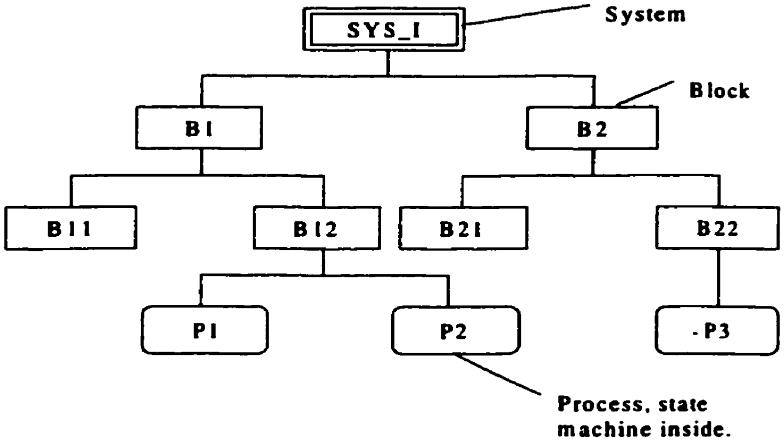


Figure 3.1 SDL System

3.2.2. SDL Communication

In SDL, communication means signals (discrete signal) sending and receiving through media (channel or route) between peer objects (system and its environment, blocks, processes), or signals (continuous signal) sending and receiving inside a process.

The system communicates with its environment by exchanging signals through channels. Blocks can communicate with each other by exchanging signals through channels. Processes communicate with the environment of the block they are contained in by exchanging signals through routes (route is the channel inside a block). Processes in the same block communicate with each other by exchanging signals through route [9].

There are two kinds of signals:

- *Discrete Signal* (which we often referred to as *signal*)

    Discrete Signals are sent or received through channels and/or routes. The reception of discrete signal can be followed immediately by a validation condition (PROVIDED). If the validation condition is not hold (FALSE) then the signal will be implicitly saved (Conditional Reception of Signal), otherwise signal input will be consumed and fire a transition [9].

- *Continuous Signal*

    Continuous signals are local variables of a process instance. Continuous signals are only evaluated if the queue is empty [9].

Since the work in this thesis does not touch continuous signals, in the discussion that follows we only deal with discrete signal.

To better understand ObjectGeode SDL communication, now we have a look on the whole process of transmission and consumption of a signal as illustrated in Figure 3.2. The events happen in following order:

- The signal X sent by a sender process,
- X is conveyed through routes or channels,
- X is stored in the queue of the receiver process,

Signals to be sent simultaneously along the same route are conveyed in random order. And each process instance that can communicate has a FIFO queue to store the signals received.

- Signal X is consumed or saved by the receiver process.

  For a signal to be consumed and processed by a process instance, the signal must have been declared to be acceptable in this state, and the receiving process instance must be waiting in the given state to consume or save it.

  When the required signal arrived in the queue (before consumption), the transition becomes *enabled*, or called *fireable*. The parameters carried by the input signal consumed will be assigned to the variables corresponding to the INPUT.

  Signals that are placed in the queue but are not declared in the state for consumption or save will be lost. Saved signals remain in the queue in the order in which they arrive.

The following is an illustration of signal transmission:

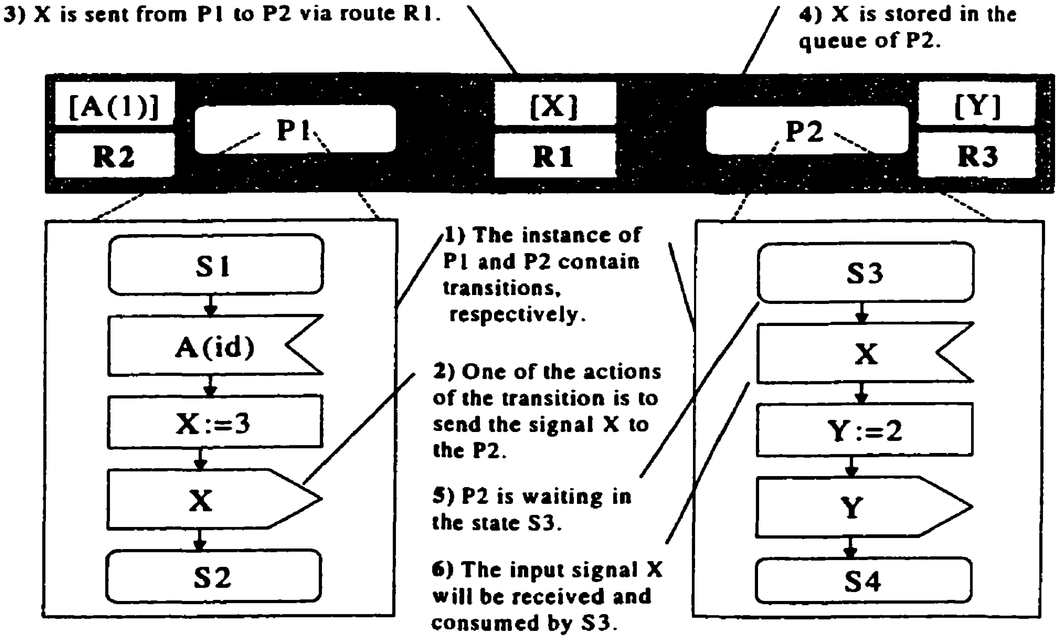

3) X is sent from P1 to P2 via route R1.
4) X is stored in the queue of P2.
[A(1)]
R2
P1
[X]
R1
P2
[Y]
R3
S1
A(id)
X:=3
X
S2
1) The instance of P1 and P2 contain transitions, respectively.
2) One of the actions of the transition is to send the signal X to the P2.
5) P2 is waiting in the state S3.
6) The input signal X will be received and consumed by S3.
S3
X
Y:=2
Y
S4

Figure 3.2 Signal Transmission in SDL

3.2.3. Dynamic Concepts of a Model

3.2.3.1. Introduction

The dynamic description of a SDL system represents the system's *operation*, including SDL transition and instantiation of process. The system is *valid* dynamically if it verifies the rules of SDL. This validity verification is called *interpretation* of the system. A valid system is a system that has been correctly interpreted [9].

3.2.3.2. SDL Transitions

The operating principle supposes that the states inside a leaf process instance are interconnected by transition. A transition is a series of actions executed by a state machine, triggered after consumption of a given signal in an initial state, and possibly leading to another state or to the death of the process instance [9].

There are two main kinds of transitions in SDL:

- Initial transitions,
- Transitions that define the processing performed by the state machine.

A transition in ObjectGeode SDL process has three parts:

- *Transition header*, which is the description the transition fireable conditions.
- *Transition action*, which is the description of the actions performed by the transition.
- *Transition terminator*, which is the description of what happens when the transition has been executed.

The following is a detail description of the transition header and transition action:

1) *Transition Header*

Initial State and Initialization Transitions

When the system is initialized, all leaf processes are set to an initial state (START). When a process is created, only the initial transition is fireable. Each initial transition is only executed once, when the process instance is created [9]. The initial transition is only used to:

- initialize variables,

- determine the initial state of the process.

Initial transitions differ from other transitions in that they can not consume signals.

Declaration of States

The transition definition must contain the declaration of at least one state characterizing the operation of state machine [9], that is to say the transition must start from a state.

Signal Reception

After each state declaration, it is possible to specify [9]:

- The reception of one or more signals, with or without validation conditions.
- The reception of a mixture of signals (with or without validation conditions).
- One or more continuous signals.
- A save mechanism for the signals received.
- A mixture of the above items.

2) *Transition Actions*

Actions are processed when a signal is consumed. The actions of the corresponding transition are executed sequentially. There are no semantic rules governing the order in which the actions of the transition are performed [9].

The following is a list of usually seen actions that can be executed in a transition [9]:

- Tasks: a task can be a variable operation action or comment text.
- Process Instance Creation.
- Signal Sending: OUTPUT action is used to send a signal.
- Time Set and Reset: SET and RESET actions are used to set or reset timer.

3) *Transition Terminator*

Usually the end of a transition is specified by the following instructions [9]:

- JOIN, indicating that a transition branch is completed, with transfer of interpretation to a part of another transition branch indicated by a LABEL IN CONNECTOR.

- NEXTSTATE, used to specify the state the process instance must be in after the transition branch has been executed.
- STOP, indicating that a transition branch is completed (the corresponding process instance will be killed).

3.2.3.2. Life Span of a Process

We are now going to take a look at the life span of a process. The process is the main link object between the static part of the description of an SDL system and its dynamic behavior. It is the only structural object that can be instantiated [9].

A state machine represented by a process reacts to *stimuli*, or signal inputs. A stimulus, after consumption, normally will trigger a series of transition action [9].

A system handles a large number of simultaneous signal transmissions and receptions. A number of stimuli can arrive over a very short period or simultaneously at the same process [9]. To receive different signals simultaneously at the same process SDL provides CREATE REQUEST action to dynamically create a new process instance.

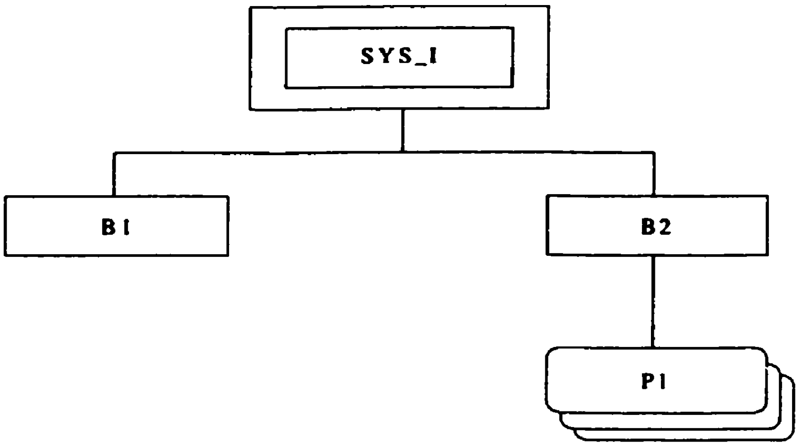


Figure 3.3 Process Instances in One Process

Instantiation also means the different processes can be activated in parallel [9], as shown in Figure 3.4.

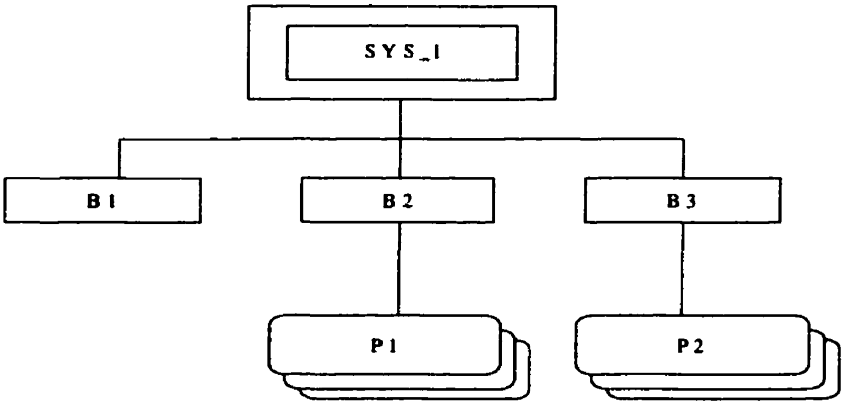


Figure 3.4 Process Instances in Different Processes

Process instance can be created statically when system is initialized, or dynamically in running time. The parallel instantiation of a number of processes also means that a number of transitions can be executed simultaneously, and they can react to the stimulus simultaneously.

The *life* of a process instance begins from the instantiation, and *death* of a process instance is the result of a STOP action at the end of a transition [9].

3.3. ObjectGeode SDL Semantics

3.3.1. Global States of the Model

This and next section present the elements used by ObjectGeode SDL simulator to interpret SDL language, especially when this interpretation does not strictly follow the Z.100 standard.

Control States of a state machine are the objects declared by the user in STATE clauses. The states defined in the EFSM model are control states [2].

Global states of a state machine are the resulting states of the model being executed, they are composed of the current values of all information that varies dynamically [2]:

- The current control state.
- Current data values which can be decomposed into:
  - Variables,
  - Formal parameters,
  - Predefined Pid (Process identifier),
  - Current timer values.

Actually a global state of an EFSM is a possible configuration of that machine.

Global states of a model are composed of [2]:

- The global states of all state machines (process instances and active procedures).
- The queue status of all processes.

All global states of a model can be explored by exhaustive simulation and can be shown in a state graph like the one presented in the next section.

3.3.2. Possible Behaviors of the Model

State machines in a SDL model are executed in parallel. In exhaustive simulation, parallelism among state machines is simulated by interleaving the execution of their transitions [2].

Let S_0 be the initial global state, and *trans*(S) be the set of fireable transitions for a global state S. A global state S is said to be ***reachable*** from the initial state S_0 if there is a list of transitions ($T_1 \ldots T_n$) such that, whatever i will be between 1 and n, T_i belongs to *trans*(S_{i-1}) and $T_i(S_{i-1}) = S_i$. The path represented by such a list of transition is a possible model behavior [2].

The set of possible behaviors of a model forms the model's *state graph*. The nodes are the reachable states and the arcs are the fireable transitions. It is a graph and not a tree, as two different behaviors can lead to the same state [2].

3.4. ObjectGeode Simulator

3.4.1. Simulation Mode

For ObjectGeode simulator, random simulation and exhaustive simulation are two automatic simulation modes as described below:

- *Random simulation*: this simulation mode automatically fires a series of transitions at random. Random transition selected by the SDL simulator is a uniform choice in the list of all fireable transitions of all processes: all the fireable transitions have the same probability [2]. This simulation mode has a little relation with the thesis topic due to its uncertainty and we do not give its detail here.

- *Exhaustive simulation*: During exhaustive simulation, the simulator creates the state graph of the SDL model according to fireable transitions. The simulator can exhaustively traverse the state graph in breadth first mode, depth first mode [2].

Exhaustive simulation, if it succeeds, provides 100% state coverage of state graph, while random simulation does not.

During simulation, ObjectGeode can find following errors in the system specification [2]:

- Deadlocks.
- Livelocks.
- Dead code: The parts of the model that are never executed (corresponding to behavior types that can never be activated and that are useless or that are a result of a modeling error).
- Signal loss: Unexpected signal inputs resulting in signal loss.
- Queue overflow.
- Dynamic errors, or "exception": type overflows, limits of an array exceeded, illegal output, non-existent answers to a decision, process stopping errors, assertions violation, etc.

To detect these errors, stop condition or observer should be defined in the simulator.

An example of deadlock and livelock is given in Figure 3.5. Each circle represents a global state reachable in the SDL model. Since no transition is fireable from S2, the model is in a deadlock. This situation may correspond to a serious error or may have been created on purpose by the designer. If the model has reached state S3, it can no longer leave the livelock containing states S3, S4 and S5. Such a behavior is difficult to detect because the system continues to function (livelock may contain thousands of scenarios) but in degraded mode (it can not return to the beginning state). Livelock may be a serious error or intentional.

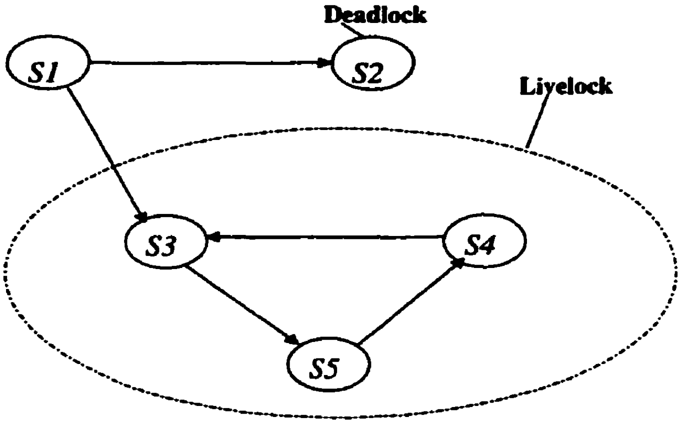


Figure 3.5 Deadlock and Livelock

For an SDL system under simulation without stop condition and observer, the simulation will end when all fireable transitions are fired and lead to deadlocks. Then the state graph can be dumped out in a file, as shown in Figure 1.1, for further analysis. Exhaustive simulation guarantees the generation of states for all possible scenarios.

3.4.2. Simulation Concepts

3.4.2.1. Basic Concepts

Simulation is done in simulation steps. A simulation step is started by actions that change the state from one to another. These actions step are composed of smaller execution steps.

The smallest execution step handled by the SDL Simulator is the elementary transition, i.e. instruction sequences located between one declared or intermediate state and another declared or intermediate state [2].

The following actions can modify the current state of the model:

- **Fire** executes a transition.
- **Wait** makes time progress.
- **Let** modifies a variable.
- **Create** creates a process instance.
- **Stop** deletes a process instance.
- **Output** outputs a signal.

The execution of any of the commands above forms a simulation step. Other commands, such as **go**, execute several simulation steps in sequence [2].

A simulation step is composed of the following operations [2]:

- Execution of the command that changes the current model state: if a dynamic error occurs, the model is left in the exception state.
- Updating coverage tables.
- Updating SDL time and internal variables such as, step number (NOW and STEP).
- Log of events.
- Computing the list of fireable transitions.
- Taking filtering conditions into account, to reduce the transition list.
- Taking stop condition into account.

The simulator always knows the list of the simulation steps that led from the initial state to the current state, it is the current scenario [2]. After each simulation step, deadlock condition, exception, livelock, stop condition and/or observer etc., will be evaluated and if satisfied, the current scenario will be saved in file corresponding to these conditions.

3.4.2.2. Exhaustive simulation in ObjectGeode

Different exploration modes are available in exhaustive simulation [2]:

- *Breadth first exploration.* If the state graph is explored in breadth first mode, then the simulator fires all the transitions at a given level before passing to the level below. This simulation will produce a minimal length scenario.
- *Depth first exploration.* If the state graph is explored in depth first mode, the simulator will fire all the transitions at the "leftmost" path before passing to the path in the right.
- *Supertrace.* This is a variant of depth first mode in which exploration is usually not exhaustive, but is less demanding on processing and memory.
- *Liveness.* This mode can detect endless loops and can check liveness properties.

For state graph generation, both breadth first mode and depth first mode are applicable since they make no difference in the state graph.

Example. This example illustrates the simulation of an SDL system, how the simulator explores this system in an exhaustive way and what a state graph is.

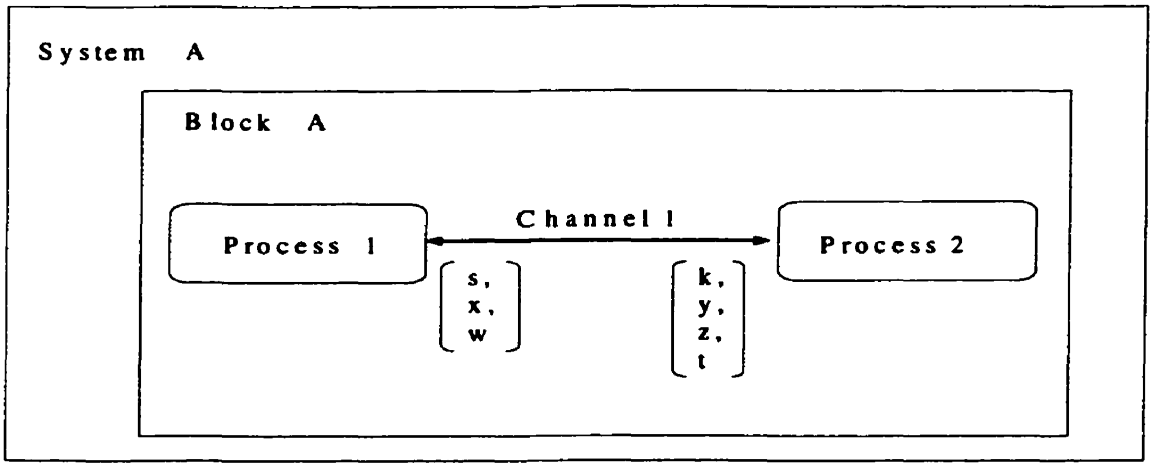
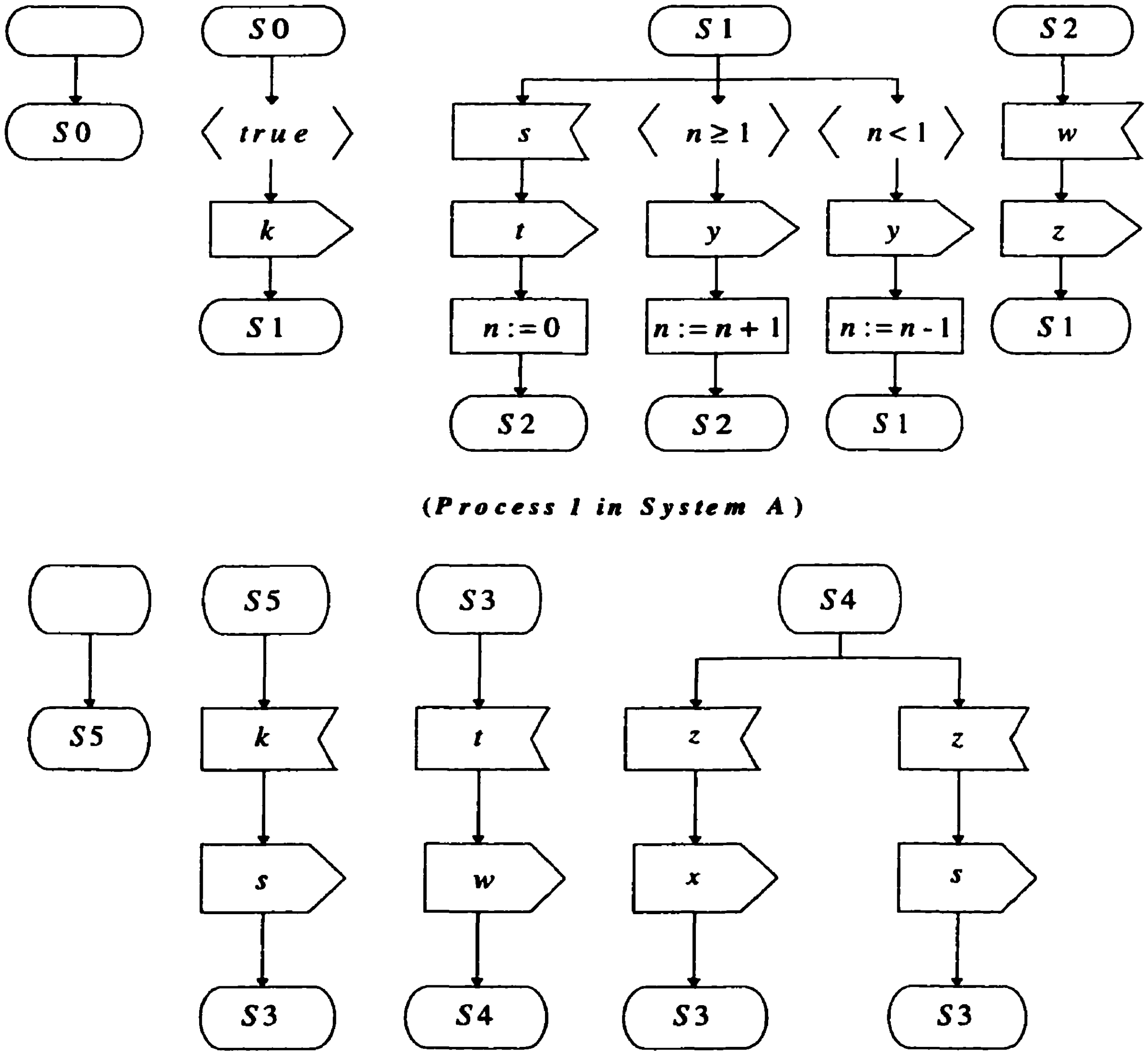


Figure 2.6 System A

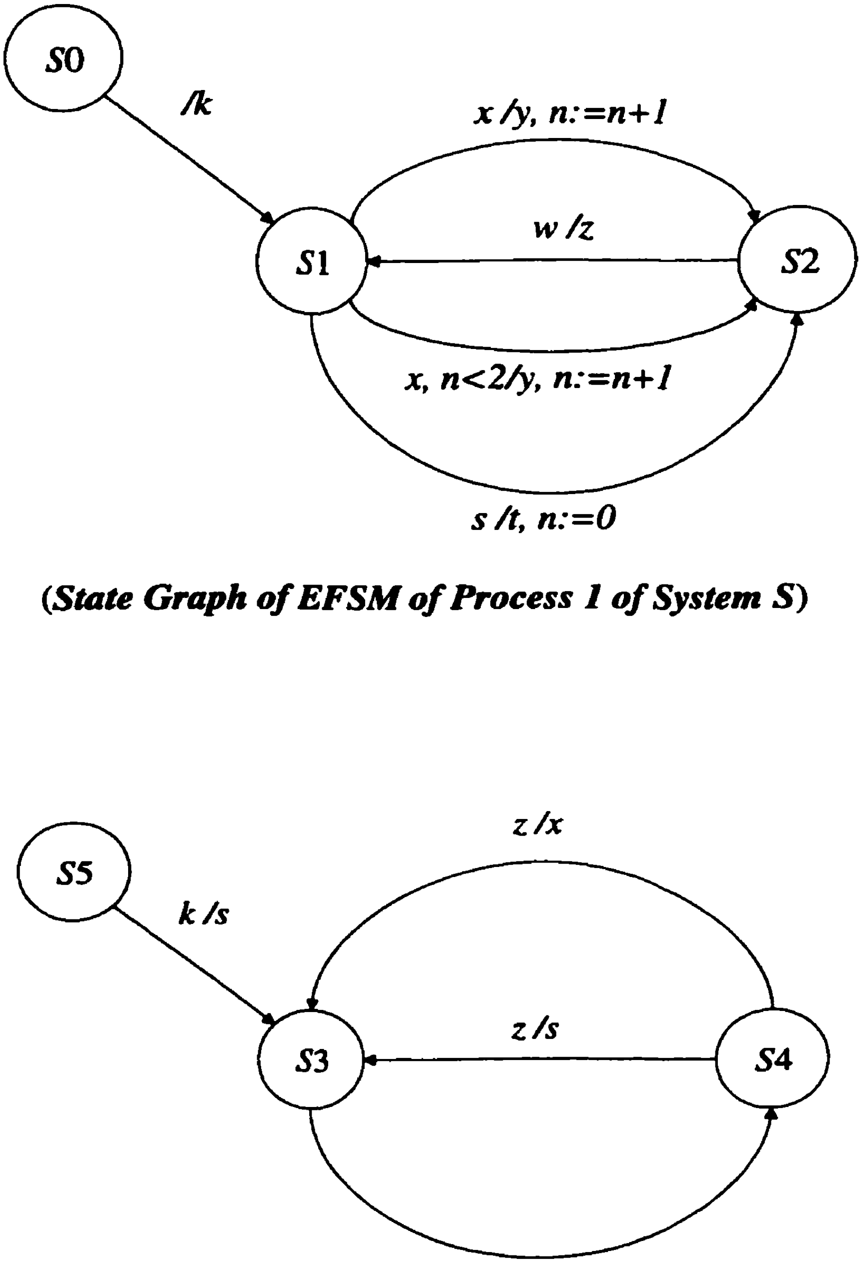
Figure 2.6 is the SDL system used in this example. SDL processes are shown in Figure 3.7. The EFSM models of each process are given in Figure 3.8:



(*Process 1 in System A*)

(*Process 2 in System A*)

Figure 3.7 Process 1 and Process 2 in System A



(*State Graph of EFSM of Process 1 of System S*)

(*State Graph of EFSM of Process 2 of System S*)

Figure 3.8 State Graphs of Process 1 and 2 of System A

Every global state can be represented as shown in Figure 3.9. During exhaustive simulation, the simulator creates the state graph as shown in Figure 3.10.

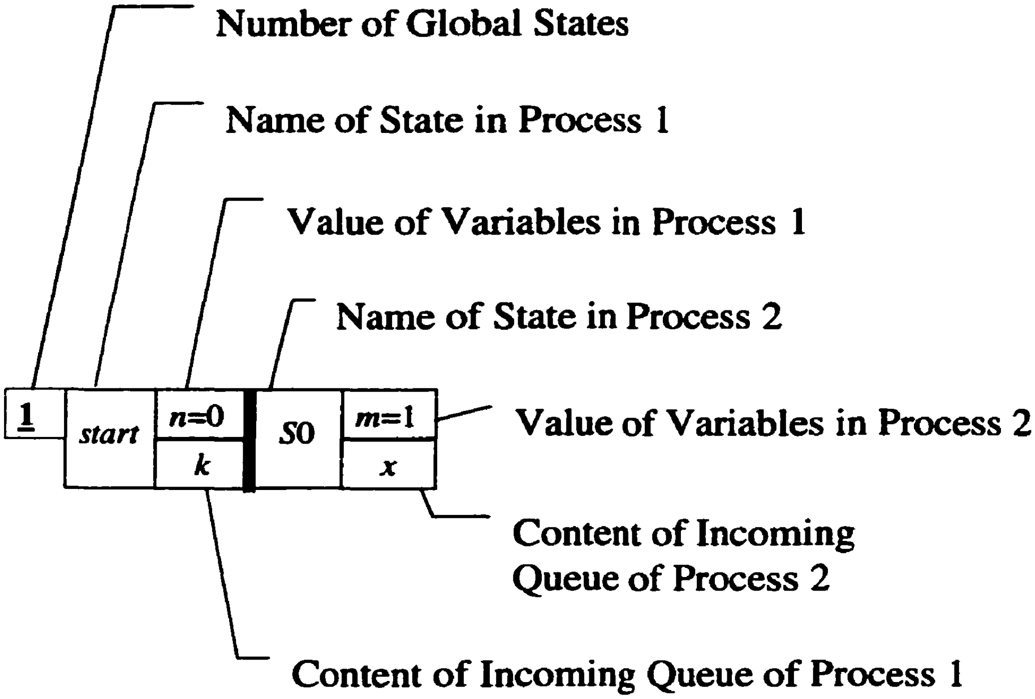
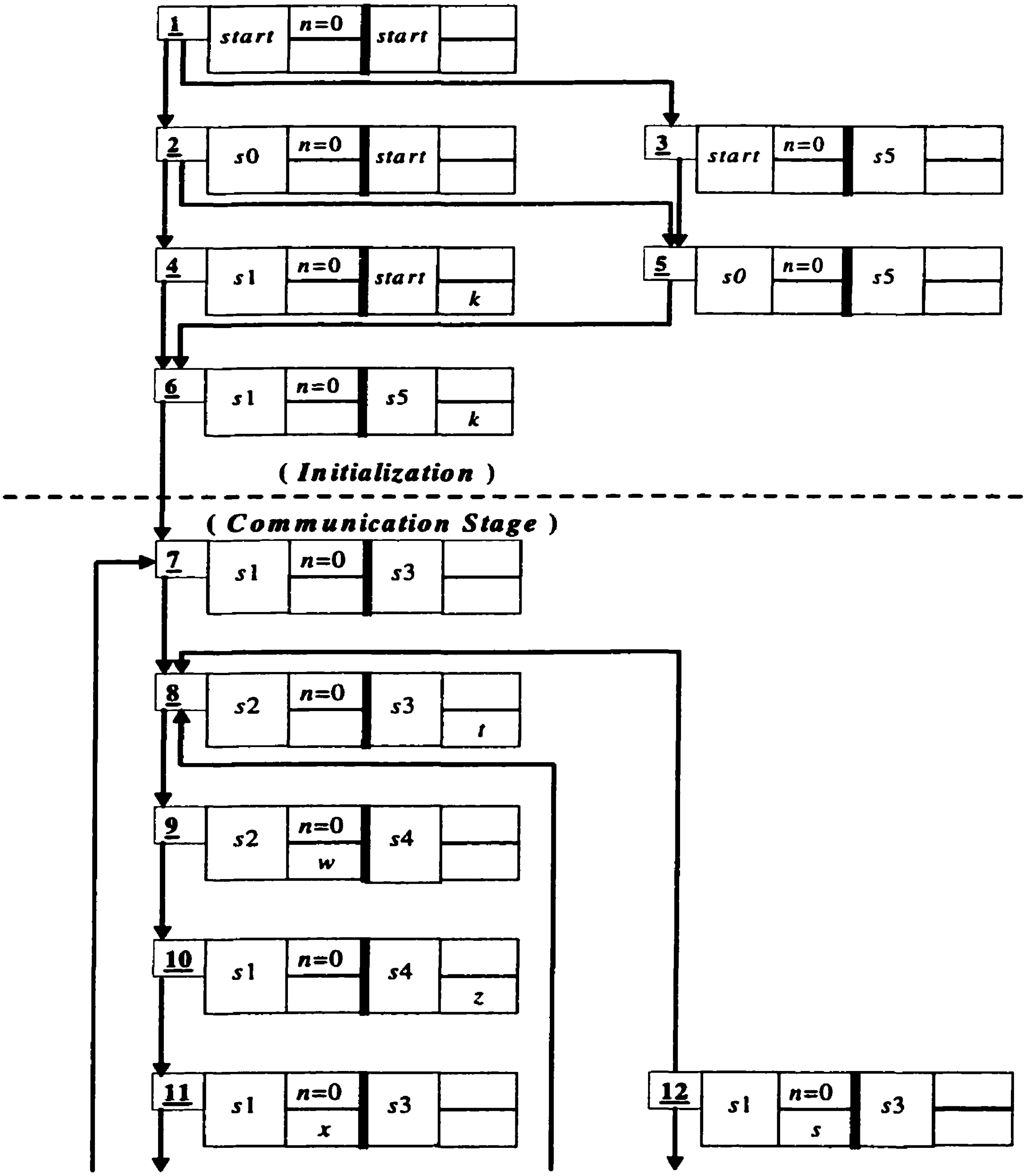

Number of Global States
Name of State in Process 1
Value of Variables in Process 1
Name of State in Process 2
1
start
n=0
k
S0
m=1
x
Value of Variables in Process 2
Content of Incoming Queue of Process 2
Content of Incoming Queue of Process 1

Figure 3.9 Syntax of Global State


1 start n=0 start
2 s0 n=0 start
3 start n=0 s5
4 s1 n=0 start k
5 s0 n=0 s5
6 s1 n=0 s5 k
(Initialization)
(Communication Stage)
7 s1 n=0 s3
8 s2 n=0 s3 t
9 s2 n=0 w s4
10 s1 n=0 s4 z
11 s1 n=0 x s3
12 s1 n=0 s s3

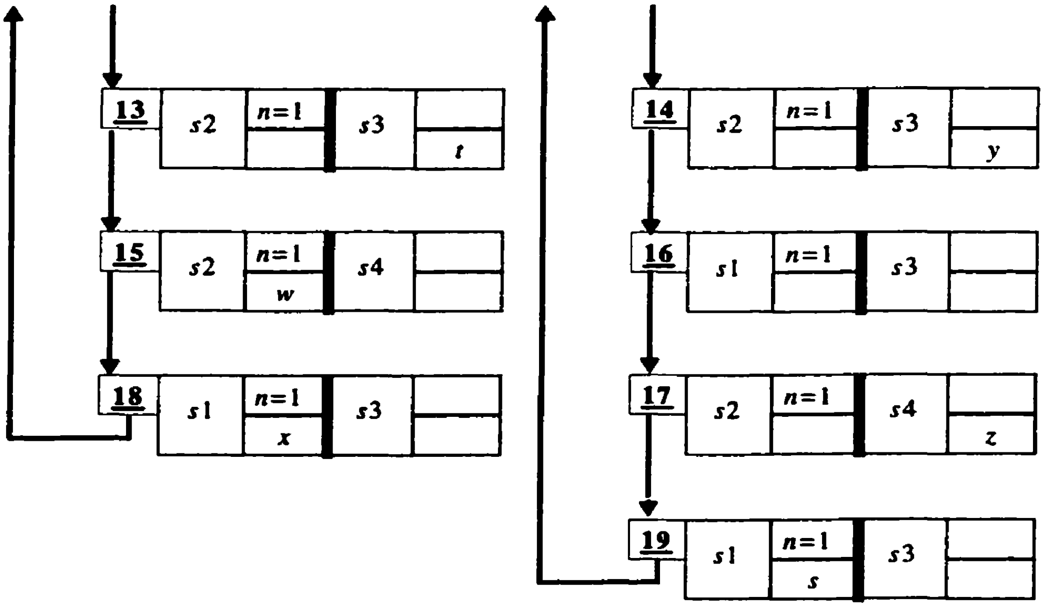


Figure 3.10 State Graph of *S*

Each global state in the state graph of the example is numbered. The number shows the order in which the graph is explored. This state graph can be obtained by exhaustive simulation in breadth first mode or depth first mode.

Note that in Figure 3.10 states 5, 6, 8 and 12 are visited several times but only one copy exists respectively. That is to say, next time simulator detects a same state that has already been generated, it will not explore transitions that have already been explored.

3.5. Summary

In this chapter, we first gave an introduction to SDL, Specification and Description Language. When the user specifies a system by using SDL, the system can be simulated with SDL simulator. Simulator can run in different simulation modes. Among them, the most important is exhaustive simulation, simulating in exhaustive simulation mode explores all the possible scenarios which can occur in real execution of the model and can be saved in a state graph dump file.

Chapter 4

CCS Generation with ObjectGeode Simulator

4.1. Relevant Work

Configuration Confirming Sequence is first defined in [5] and [12]. It is a single parameterized input sequence that can distinguish the expected tail configuration from a set of suspicious configurations.

The existence of CCS has been studied in [5] and [12]. This problem can be transformed into the problem of configuration distinguishability. For more detail reader can refer to [5] or [12].

Generation of CCS can be done in several ways. In [5] authors present an approach based on the concept of "distinguishing machine". The problem authors are dealing with in [5][12] can be stated as following:

Problem 1 [12]. Given an EFSM *E*, an "expected" configuration, a set of *k* sets of suspicious configurations, each of which is represented as a pair of state and partial context, and an integer *l*, we are required to find an input sequence of length at most *l* that confirms the expected configuration in a *maximal* number of sets of configuration sets (among given *k* sets). The sequence is called CCS.

A solution to this problem in previous chapters in term s of EFSM model has been presented in [Part I, 12]. The basic idea of this solution is to construct an EFSM (a distinguishing machine) such that a desired CCS is a transfer sequence taking the machine from the initial state to a state which indicates that a maximal number of configuration sets is separated from the given configuration. Such a transfer sequence can be determined using existing reachability methods and tools. In this solution, the global *configuration* space of the EFSM has not always to be constructed, at the same time, the solution still requires an explicit representation of the whole *state* space, which is exponential in the number of suspicious configuration sets. The question arises whether one can solve Problem 1 without explicitly constructing a distinguishing machine, as its size may be prohibitively huge. The solution is to build a system of k+2 communicating processes. One process is a given EFSM M (SDL spec) initialized in the expected configuration, each of the k processes represents an EFSM projection [5] initialized in a suspicious configuration and slightly modified to support communications with other processes. In addition, a monitor is required to ensure all the communications between processes and to terminate communications when it is necessary [12].

Problem 2 [12]: Given an EFSM E, an "expected" configuration, a set of k sets of configurations, each of which is represented as a pair of state and partial context, and an integer l, we are required to find an input sequence of length at most l that confirms the expected configuration in a *given* number of configuration sets.

Once a method for solving the Problem 2 is available, the initial problem can also be solved using a *dichotomy* method as follows.

Algorithm 2.1 [12]. Dichotomy search.

Input. The EFSM, an "expected" configuration, a set of k sets of "suspicious" configurations, and an integer l.

Output. An input sequence CCS.

```
1. $i_{min}$:=1, $i_{max}$:=$k$.
2. $i$:= $i_{max}$.
3. If there exists a sequence of at most $l$ symbols that separates the expected
   configuration from $i$ configuration sets, then do
   $i_{min}$:=$i$, $i$:=$i_{max}$-[( $i_{max}$- $i_{min}$)/2] else $i_{max}$:=$i$, $i$:= $i_{min}$+[( $i_{max}$- $i_{min}$)/2]
4. If $i_{min}$< $i_{max}$, then go to 2.
5. Stop.
```

Here [x] denotes the integral part of x. Step 3 calls for a method for Problem 2 explained above [12].

In what follows, we discuss the approach for solving Problem 1 different from that in [12].

4.2. CCS Generating with Protocol Verification Tool

4.2.1. Minimal Separating Sequence

Minimal separating sequence is a separating sequence with some restrictions.

Definition 4.1: Given an EFSM machine *M*, configuration *a* and configuration *b* for *M*. Parameterized input sequence $x=x_1x_2...x_n$ is a separating sequence. x is a ***minimum separating sequence*** if $\forall i<n \Rightarrow \{x_1x_2...x_i$ is not a separating sequence$\}$.

Minimal separating sequences will be generated from state graph dump file by a post-processing C++ program as explained in Figure 1.1. Then they are used as candidates from whom CCS is selected. In practice we are only interested in configurations that are ***l-distinguishable***, so there is an upper limit l ($0<l<\infty$) for the length of minimal separating sequences.

4.2.2. CCS Generation

4.2.2.1. Distinguishing System

The method used for CCS generation is derived directly from Definition 1.7, 1.8 and 4.1. According to these definitions, given an EFSM machine *E*, a "expected" configuration *c* and a "suspicious" configuration set *C*, a CCS is a separating sequence that can separate *c* from each configuration in set *C*. Thus the first step of CCS generation is to find all separating sequences for each suspicious configuration in *C*. In the second step, CCS will be selected among the separating sequences.

To find all separating sequences, we first construct an SDL system *D*. In *D*, there are three processes *P*, *S* and *M*, both *P* and *S* are derived from *E*. The major difference between *P* and *S* is that they have different initial configurations. *P* starts from the configuration that we want to confirm and *S* starts from a suspicious configuration.

Then we simulate *D* by using ObjectGeode simulator. During the simulation *P* and *S* compare their inputs and outputs. If a discrepancy is found, *D* goes to a deadlock state and the simulation along this path will be terminated, and a specific signal "SequenceFound" will be produced. Simulator will explore all the paths with a given length *l* and then dump out state graph in a file. This state graph contains all the information about separating sequences.

After that a C++ post-processing program will be executed to extract all separating sequences from the state graph and select a separating sequence as CCS.

The SDL system *D* is called a ***distinguishing system*** in this thesis. The key point of constructing *D* is how to detect discrepancy of inputs and outputs of *P* and *S*. The idea is to add a comparison mechanism into *P* and *S*, as described below.

Assume *D* is constructed to confirm configuration *c* from a set of configurations *C*. Let *P* is derived from the EFSM *E*, with following changes:

1) *P* starts from the configuration *c*.
2) For every output action in *E*, *P* makes the same output. Then it waits for a *resume* or *invalid* message.
3) For every input action in *E*, *P* first outputs a message that requests an input, then waits for the input feedback:
   - If the input feedback is *resume*, then:

     If the output received is the same as the original input in *E*, it continues the next action.

     If the output received is not the same as the original input in *E*, then it outputs a message called *Sequencefound* and then deadlocks.

- If the input feedback is *invalid,* then:

  If the output is the same as the original input in *E*, then it outputs a *Sequencefound* message and then deadlocks.

  If the output is not the same as the original input in *E*, then it outputs a message called *TwoInvalidSignals* and then deadlocks.

Let *S* be the same as the original EFSM *E*, except for the following changes:

1) *S* starts from a set of suspicious configurations *C*.
2) For every output action in *E*, input this message first. If the input message is the same as the message to be output in *E*, output a signal called ***resume***, then continue the next action; else output a signal called ***invalid*** and deadlock.
3) For every input action in *E*, input a message. If the input message is one of the original inputs of *E*, then output a ***resume*** signal and continue the next action. At the same time input all other messages and output an ***invalid*** signal, and then continue waiting for the correct input in the same state.
4) The timeout signal is processed the same way as ordinary input message.

Finally, let *M* be an EFSM machine communicating with *P* and *S*. *M* performs the following actions:

1) Set a simulation depth counter. When simulation goes beyond the upper limit *l*, *M* will be deadlocked, the simulation among this path will be stopped.
2) When an output from *P* is received, *M* passes the message to *S* and wait for a signal (***resume*** or ***invalid***) from *S*. When it receives a message from *S*, *M* passes the message to *P*.
3) After receiving a ***resume*** or ***invalid*** from *S*, *M* send the same message to *P*.
4) Signal *TwoInvalidSignals* will be discarded.

The above modifications are represented in CSP (Communicating Sequential Processes [10]) as shown in Table 4.1. The following is a brief explanation of some CSP operators [10]:

- **Process operator**

  $Q!x$ — on channel Q output (value of) x.

  $Q?x$ — from channel Q input to x.

  $A \| B$ — process A in parallel with process B.

  $a \rightarrow B$ — a then B.

  $(a \rightarrow A \mid b \rightarrow B)$ — a then A choice b then Q (provided $a \neq b$).

  P sat S — (process) P satisfies (specification) S.

- **Logic operator**

  $A = B$ — A equals to B.

  $A \Rightarrow B$ — if A then B.

  $\neg P$ — not P (P is not true).

- **Set operator**

  $A \cup B$ — A union B.

  $A \cap B$ — A intersect B.

  $A - B$ — A minus B.

Assume E_{input} is the set of all possible input messages in E, $k(x_1 \ldots x_n)$ is a parameterized signal, where k and $x_1, \ldots, x_n$ are constants, *resume*, *invalid* and *SequenceFound* are all constant signals, $msg(p_1 \ldots p_n)$ are parameterized signal variable, then,

- G is a channel connecting environment in E;
- Q_p stands for the channel connecting process M and P;
- Q_s for the channel connecting S and M;
- $K=G!k(x_1 \ldots x_n)$; where k and $x_1, \ldots, x_n$ are constants.
- $L=G?msg(p_1 \ldots p_n)$;
- $P(K)=Q_p!k(x_1 \ldots x_n) \rightarrow Q_p?x \rightarrow$

  ```
      if ($x=invalid$) then $Q_p!SequenceFound$;
  ```

- $S(K)=Q_s?msg(p_1 \ldots p_n) \rightarrow$

  ```
      if($msg = k$) then
          (if ($p_1=x_1 \wedge \ldots \wedge p_n=x_n$) then ($Q_s!resume$)
          else ($Q_s!invalid$))
      else ($Q_s!invalid$);
  ```

```
• $M(K)=(Q_p?msg(p_1...p_n)\rightarrow$
      if($msg = resume$) then ($Q_s!resume$)
      else (if($msg = invalid$) then ($Q_p!invalid$)
          else ($Q_s!msg(p_1...p_n)$)));
• Let $PA=(Q_p!k(x_1...x_n)\rightarrow(Q_p?x)\rightarrow$
      if($x = invalid$) then $Q_p!SequenceFound$)
  Let $PB=(Q_p!msg2(p_1...p_n)\rightarrow(Q_p?x)\rightarrow$
      if($x = resume$) then $Q_p!SequenceFound$);
      Where $msg2(p_1...p_n) \in E_{input}- msg(p_1...p_n)$;
  Then $P(L)=PA \parallel PB;$
• $S(L)=Q_s?msg(p_1...p_n)\rightarrow$
      if($msg = k$) then
          (if ($p_1=x_1 \wedge ... \wedge p_n=x_n$) then ($Q_s!resume$)
          else ($Q_s!invalid$))
      else ($Q_s!invalid$);
• $M(L)=(Q_p?msg(p_1...p_n)\rightarrow$
      if($msg = resume$) then ($Q_s!resume$)
      else (if($msg = invalid$) then ($Q_p!invalid$)
          else ($Q_s!msg(p_1...p_n)$)));
```

	Original Action in *E*	**Action in *P***	**Action in *S***	**Action in *M***
Output	$K=G!k(x_1...x_n)$	$P(K)$	$S(K)$	$M(K)$
Input	$L=G?msg(p_1...p_n) \rightarrow$ **If**($msg=k$) **then** L'	$P(L)$	$S(L)$	$M(L)$

Table 4.1 Modification of original actions in *M* using CSP description

Definition 4.2: An SDL system is called a *distinguishing system of the EFSM model E* if it is derived from *E* by following the instructions described in Table 4.1.

D will generate a *SequenceFound* signal if *P* generates some I/O sequences that cannot be reproduced by *S*, and this makes the finding of separating sequence possible. The reason for this statement is described informally as follows:

1) If *S* cannot generate the same output as *P*, it sends out the message *invalid* to *P*. If *S* can generate the same output message as *P*, *S* will send the message *resume* back to *P* and go to a next state. After *M* transfers this message from *S* to *P*, *P* will check the input and decide what to do next. If it is the message *invalid*, then *P* will send out the message *Sequencefound* and a minimal separating sequence is found. If the input message is the message *resume*, then the message output by *P* is same as the message input by *S*, *P* will continue execution.

2) If *P* and *S* are waiting for the different inputs, then there are two cases:

- If the message received is the one that is expected by *P* but not by *S*, then *S* will send out the message *invalid* and then go blocked. *P*, after receiving the message *invalid*, will send out the message *Sequencefound* indicating that a minimal separating sequence is found.
- If the message received is the one that is expected by *S* but not by *P*, *P* will send this message to *S* through *M* and then wait for the message *resume* to be sent back by *S*. *P*, after receiving this message *resume*, will send out the message *Sequencefound* indicating that a minimal separating sequence is found.

3) If *S* and *P* are waiting for the same input, then there are two cases:

- If the message input is the one expected by *P* and by *S*, *S* will send the *resume* signal back to *P*. Upon receipt of this *resume* signal, *P* will continue execution of simulation.
- If the message input is the one expected by *P* and by *S*, *S* will send the *invalid* signal back to *P*. Upon receiving this *invalid* signal, *P* will release a *TwoInvalidSignals* message and continue execution trying to find a longer separating sequence.

The interaction among *M*, *P* and *S* is shown in Figure 4.2. Figure 4.2(a) illustrates the case when *E* is waiting for some input other than *msg*. According to the action in Table 4.2, *P* sends out all possible inputs including *msg*. *M* passes *msg* to *S*. If *msg* is the signal *S* wants, then *S* will send an *resume* back. *M* passes *resume* to *P*. Since receipt of the message ***resume*** indicates ***msg*** is been accepted by *S*, it sends out a message *SequenceFound*. Figure 4.2(b) illustrates the case when *E* outputs ***msg***. ***P*** sends a signal ***msg*** to ***M***, ***M*** passes it to *S*. If ***msg*** is not a message that *S* intends to output, *S* will send an ***invalid*** back. ***M*** passes ***invalid*** to ***P***. When the message ***invalid*** is received, *P* judges that the signal is not wanted by *S*, so *P* sends out a *SequenceFound*. Figure 4.2(c) corresponds to an input not acceptable in both states of *E*. Finally *TwoInvalidSignals* will be issued by *P*.

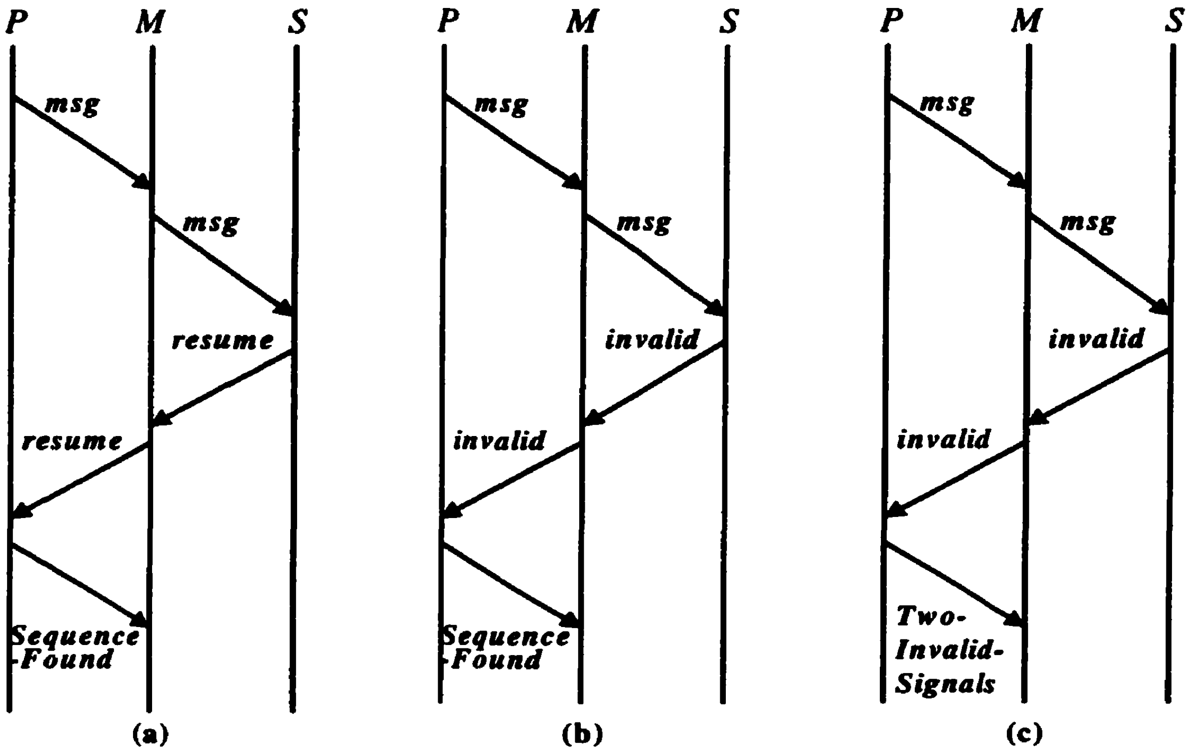


Figure 4.2 MSC for CCS Generation

Finally we note that to find a CCS for the set *C* of suspicious configurations, we have to construct, strictly speaking, $|C| = k$ distinguishing systems. These systems differ in the initial configuration of the S module, therefore, in fact, we can always use just a single system, and ObjectGeode will nondeterministically initialize the system in all possible initial global states.

4.2.2.2. Extension to Depth First Search

After a state graph has been generated, we will enhance a traditional depth first search, as described, for example, in [11] to find all separating sequences. Compared to the Depth First Searching Mode in ObjectGeode simulator, which does not explore the global states that have been explored, the Depth First Search after enhancement will explore all global states that have been visited. The algorithm for it is defined in pseudo-code as follows:

The dump state graph consists from the set of all nodes *U*, and the set of edges *V*, each edge is a triple (*u*, *u'*, *a*), where *u* and *u'* are nodes and *a* is a label (message).

Here we present an algorithm that finds all the paths in the graph that end with an edge with the *SequenceFound* label.

Algorithm [4.1]:

Input: V; l - the upper limit; E_{input}; u_0 – start node.

Output: a set of minimal separating sequences.

```
Main( )
        EDFS(u_0, ε)

EDFS(u, x)
        if |x ↑ E_input| ≤ l then
                for each (u, u', a) ∈ V
                                if (a = SequenceFound) then print x ↑ E_input
                                else EDFS(u', xa)
                                endif
                endfor
        endif
```

ε is an empty string, $x \uparrow E_{input}$ is obtained from x by deleting symbols not from E_{input}.

The syntax of state graph dump file can be found in Chapter 5 of [2]. The code of Algorithm 4.1 can be found in Appendix C, and here is a brief explanation of the code:

Class TREE: is the data structure stores the node of state graph.
Color: is an array used to mark the visited node in the state graph.
ConstructAdjList: is the function used to construct adjacent list.
EDFS: is the function used to explore the paths in dumped state graph.
GetMSS: is the function to find minimal separating sequences.
FindCCS: is the function to find CCS among minimal separating sequences.
PrintMSS: is the function to print out all minimal separating sequences.

In the *PrintMSS*, a filter has been set to filter out the output signals of *P*. Leave the remainder to be pure input signal sequence, which is minimal separating sequences.

4.2.2.3. Selecting CCS from Minimal Separating Sequences

For configuration c and the set C of all given suspicious configurations, if there is a common separating sequence x for c and every configuration pair(c, c'), where $c' \in C$, then x is a CCS. This thesis deals with only this case, the code for this selection is function *FindCCS* in Appendix C. Otherwise, we may decide to find a minimum number of separating sequences that take together separate the expected configuration from a maximum number of suspicious configurations. To this end one can use algorithms for solving the classical set cover problem. In this thesis, however, will simply indicate that no CCS exists in this case.

4.2.3. Correctness of CCS Generation Algorithm

Here we prove correctness of the method for restricted class of plain SDL processes that always have a necessary output on the transitions (branches). However, according the SDL semantics and practice, inputs, that are not declared in a particular state defines transitions without outputs. We assume that such "silent" outputs can be observed during the testing.

Here we use next notations, borrowed from [10]:

$x \upharpoonright A$ is restriction to the alphabet A;

$s \downarrow G$ is a sequence of events (trace) on communication channel G.

Now we show that all the separating sequences can be obtained by restriction on E_{input} of traces ended with signal *SequenceFound.*

For simplicity consider the case when C is a singleton, and the process M is eliminated for the sake of simplicity.

Proposition 4.1: A trace is corresponds to deadlock in $P \parallel S$ if and only if it ended with *SequenceFound*.

Proposition 4.2: Let (s_i, v_i) and (s_j, v_j) be two configurations, x is a minimal separating sequence for (s_i, v_i), (s_j, v_j) if and only if $\exists$ *trace* $\in$ traces$(P_i(v_i) \parallel S_j(v_j))$.(trace[#*trace*] = *SequenceFound* ^ *trace*$\upharpoonright E_{input} = x$.

Proof: We prove this by induction.

1) Base of induction: for $|x| = 1$ statement holds clearly from definition.
2) Step of induction. Let the proposition holds for all x of length r. We will show that proposition holds for all the sequences of length $r + 1$.

Necessity: Let $x = ax'$ is a minimal separating sequence, $a \in E_{input}$, $x' \in E_{input}{}^{r}$. a is not a separating sequence. This means there exists *trace*1of $(P \parallel S)$ such that *trace*1$\upharpoonright E_{input} = a$.

Let (s_i', v_i') is a-successor of the (s_i, v_i) and that of (s_j, v_j) is (s_j', v_j'), then by assumption of the step of math induction, there is such *trace*2 $\in$ traces$(P_i'(v_i') \parallel S_j'(v_j'))$ that the last signal of *trace2* is *SequenceFound* and *trace*2 $\upharpoonright E_{input} = x'$. (restriction on E_{input} is equal to x'). By definition of distinguishing system, there is such *trace*1 of $(P_i'(v_i') \parallel S_j'(v_j'))$ that *trace*1 $\upharpoonright E_{input} = a$. Hence, *trace*1 ^ *trace*2 $\upharpoonright E_{input} = x$ and the last signal in *trace*1 ^ *trace*2 is the *SequenceFound* sumbol. Necessity is proven.

Sufficiency: Let *trace* $\in$ (P_i'(v_i') $\|$ S_j'(v_j')) such that the last element is equal to *SequenceFound* and restriction of the trace on E_{input} is sequal to x $\in$ E_{input}, Let $x = ax$', where $a \in E_{input}$, x'$\in$ E_{input} is a–successor [7] of (s_i, v_i) and x-successor of (s_j, v_j) is (s_j', v_j'). By definition of the distinguishing system, there is a trace1, trace2 such that *trace*1 ^ *trace*2 = *trace*, *trace*1 $\upharpoonright$ E_{input} = a, *trace*1 $\upharpoonright$ E_{input} = x'. Hence, *trace*2 $\in$ traces(P_i'(v_i') $\|$ S_j'(v_j')).

From the main assumption of the step of induction x' is a minimal separating sequence for guarantee (s_i', v_i') and (s_j', v_j'). Hence ax' is a separating sequence for (s_i, c_i) and (s_j, c_j). From Proposition 4.1, it is a minimal separating sequence.

Chapter 5

Generating CCS for INRES protocol

5.1. Distinguishing System for INRES Protocol

The INRES protocol, Initiator-Responder protocol, is an abridged version of the Abracadabra protocol used for academic studies and illustrative purposes. It is a connection-oriented, asymmetrical communication protocol featuring many OSI concepts [6].

To demonstrate our approach, in this thesis a part of INRES protocol, Responder process, has been transformed into a distinguishing system *D*. *D* is shown in Figures 5.1, 5.2, 5.3 and 5.4. This distinguishing system is built to confirm configuration c = (*Connect*; *number=un*) from a set *C* of configurations $\{s_1, s_2, s_3, s_4\}$, where s_1 = (*Wait*; *number=un*), s_2 = (*Wait*; *number=zero*), s_3 = (*Disconnected*; *number=zero*) and s_4 = (*Connected*; *number=zero*). Figure 5.1 shows the original EFSM model for Process Responder (*P'*) in INRES. Figure 5.2 shows a part of primary EFSM *P* (transition between state *Disconnected* to *Wait*) modified from the original machine *P'* with the configuration *c*. Figure 5.3 shows a part of the secondary EFSM *S* (transition between state *Disconnected* to *Wait*) modified from the original machine *P'* with the configuration set *C*. Monitor *M* is presented in Figure 5.4.

In Figure 5.1, 5.2, 5.3 and 5.4, *D* stands for state *Disconnected* in original EFSM model *E*, *W* stands for state *Wait* in *E*, *C* for *Connected* in *E*. Other states are new ones for the distinguishing system.

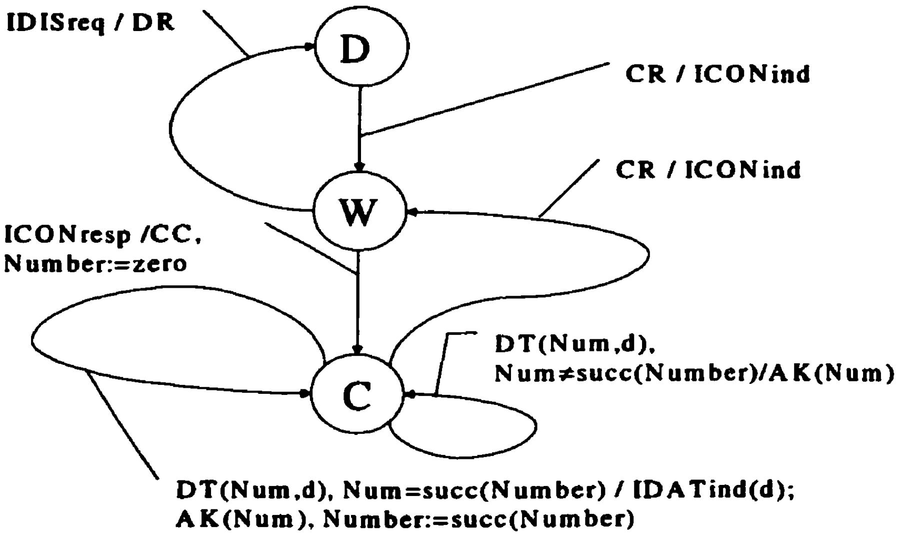

IDISreq / DR
D
CR / ICONind
CR / ICONind
W
ICONresp /CC, Number:=zero
DT(Num,d), Num≠succ(Number)/AK(Num)
C
DT(Num,d), Num=succ(Number) / IDATind(d); AK(Num), Number:=succ(Number)

Figure 5.1 EFSM Model of INRES

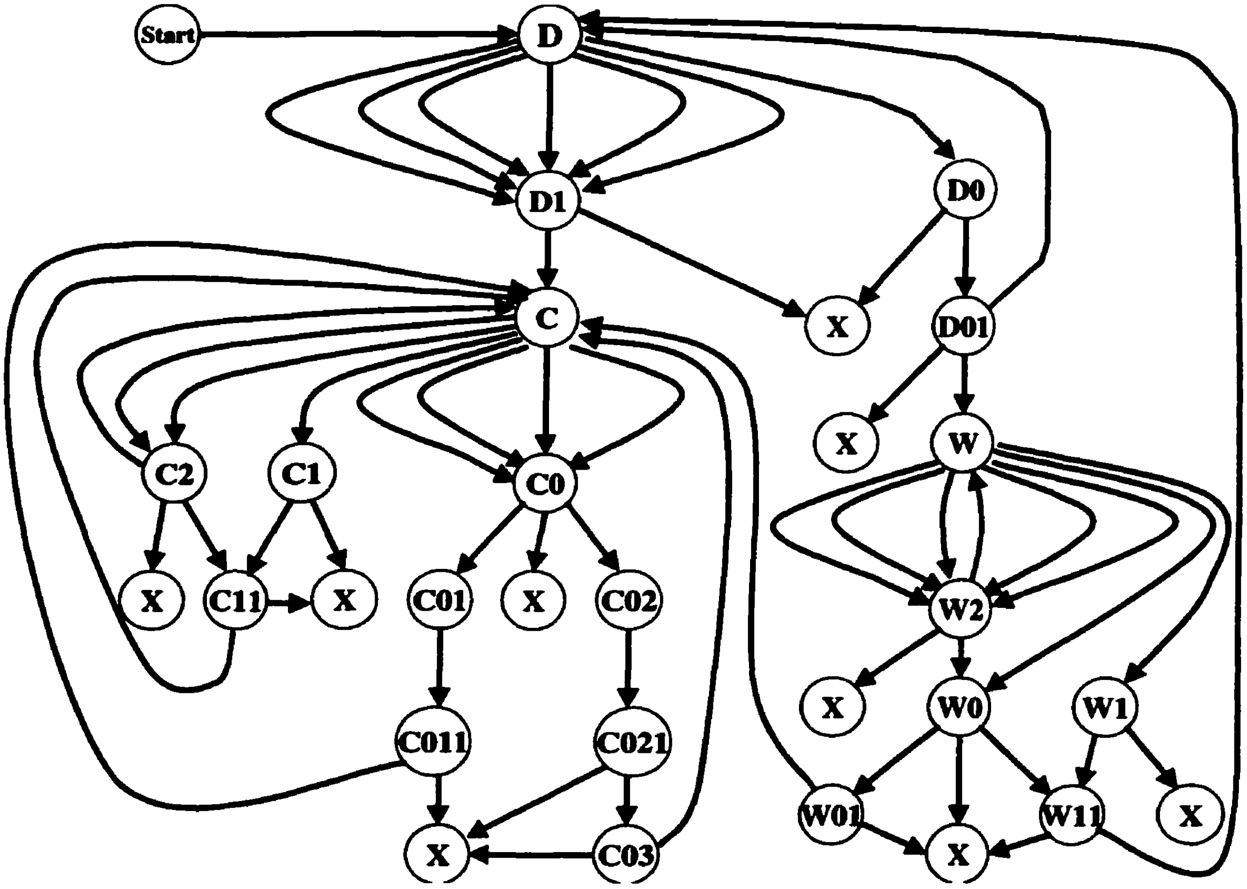


Figure 5.2 Process *P* in Distinguishing System

Appendix B presents the SDL description of process *P* of the distinguishing system. Here we give its state transition graph in Figure 5.2, where D stands for state Disconnected in *P* as shown in Appendix B, D0 for state Disconnected_0, D1 for Disconnected_1, D01 for Disconnected_0_1, C for Connected, C0 for Connected_0, C1 for Connected_1, C2 for Connected_2, C01 for Connected_0_1, C02 for Connected_0_2, C011 for Connected_0_1_1, C021 for Connected_0_2_1, C03 for Connected_0_3, C11 for Connected_1_1, W for Wait, W0 for Wait_0, W1 for Wait_1, W2 for Wait_2, W01 for Wait_0_1, W02 for Wait_0_2. Here X stands for deadlock. There should be only one X in Figure 5.2, we have several of them for convenience. Start node in Figure 5.2 stands for the start state of process *P*.

Transitions of process *P* in the distinguishing system are defined as following:

t(Start→D), stands for the transition (Start → D);

t(D→D0), stands for the transition (D — /CR → D0);

t(D→D1)^{1}, for the transition (D — /ICONresp → D1);

t(D→D1)^{2}, for the transition (D — /IDISreq → D1);

t(D→D1)^{3}, for the transition (D — /DT(Num,d), Num:=un, d:=FALSE → D1);

t(D→D1)^{4}, for the transition (D — /DT(Num,d), Num:=un, d:=TRUE → D1);

t(D→D1)^{5}, for the transition (D — /DT(Num,d), Num:=zero, d:=FALSE → D1);

t(D→D1)^{6}, for the transition (D — /DT(Num,d), Num:=zero, d:=FALSE → D1);

t(D0→D01), for (D0 — resume/ICONind → D01);

t(D0→DEADLOCK), for (D0 — invalid/SequenceFound → DEADLOCK);

t(D01→W), for (D01 — resume/Wait → W);

t(D01→DEADLOCK), for (D01 — invalid/SequenceFound → DEADLOCK);

t(D1→D), for (D1 — invalid/TwoInvalidSignals → D);

t(D1→DEADLOCK), for (D1 — resum/SequenceFound → DEADLOCK);

t(C→C0)^{1}, for (C — true/DT(Num, d), Num:=un, d:=FALSE → C0);

t(C→C0)^{2}, for (C — true/DT(Num, d), Num:=un, d:=TRUE → C0);

t(C→C0)^{3}, for (C — true/DT(Num, d), Num:=zero, d:=FALSE → C0);

t(C→C0)^{4}, for (C — true/DT(Num, d), Num:=zero, d:=TRUE → C0);

t(C→C1), for (C — /CR → C1);

t(C→C2)^1, for (C — /ICONresp → C2),

t(C→C2)^2, for (C — /IDISreq → C2);

t(C0→C01), for (C — resume, Num≠succ(Number)/ → C01);

t(C0→C02) , for (C — resume, Num=succ(Number)/ → C02);

t(C01→C011), for (C01 — /AK(Num) → C011);

t(C0→DEADLOCK) , for (C — invalid/SequenceFound → DEADLOCK);

t(C1→C11), for (C1 — resume/ICONind →C11);

t(C1→DEADLOCK), for (C1 — invalid/SequenceFound →DEADLOCK);

t(C11→C), for (C11 — resume/ →C);

t(C11→C11), for (C11 — invalid/SequenceFound →C11);

t(C2→DEADLOCK), for (C2 — resume/SequenceFound → DEADLOCK);

t(C2→C), for (C2 — invalid/TwoInvalidSignals → C);

t(C01→C011), for (C01 — /AK(Num) → C011);

t(C02→C021), for (C02 — /IDATind(d) → C021);

t(C11→C), for (C11 — resume/ → C);

t(C11→DEADLOCK), for (C11 — invalid/SequenceFound → DEADLOCK);

t(C011→C), for (C011 — resume/ → C);

t(C011→DEADLOCK), for (C011 — invalid/SequenceFound → DEADLOCK);

t(C021→C03), for (C021 — resume/AK(Num) → C03),

t(C021→DEADLOCK), for (C021 — invalid/SequenceFound → DEADLOCK);

t(C03→C), for (C03 — resume/, Number:=succ(Number) → C),

t(C03→DEADLOCK), for (C021 — invalid/SequenceFound → DEADLOCK);

t(W→W0), for (W — ICONresp/ → W0);

t(W→W1), for (W — IDISreq/ → W1);

t(W→W2)^1, for (W — /DT(Num,d), Num:=un, d:= FALSE → W2);

t(W→W2)^2, for (W — /DT(Num,d), Num:=un, d:= TRUE → W2);

t(W→W2)^3, for (W — /DT(Num,d), Num:=zero, d:= FALSE → W2);

t(W→W2)^4, for (W — /DT(Num,d), Num:=zero, d:= TRUE → W2);

t(W→W2)^5, for (W — /CR → W2);

t(W0→W01), for (W0 — resume/CC, Number:=zero → W01);

t(W0→DEADLOCK), for (W0 — invalid/SequenceFound → DEADLOCK);

t(W01→C), for (W01 — resume/ → C);

t(W01→DEADLOCK), for (W01 — invalid/SequenceFound → DEADLOCK);

t(W1→W11), for (W1 — resume/CR → W11);

t(W1→DEADLOCK), for (W1 — invalid/SequenceFound → DEADLOCK);

t(W11→D), for (W11 — resume/ → D);

t(W11→DEADLOCK), for (W11 — invalid/SequenceFound → DEADLOCK);

t(W2→W), for (W2 — resume/TwoInvalidSignals → W),

t(W2→DEADLOCK), for (W2 — invalid/SequenceFound → DEADLOCK);

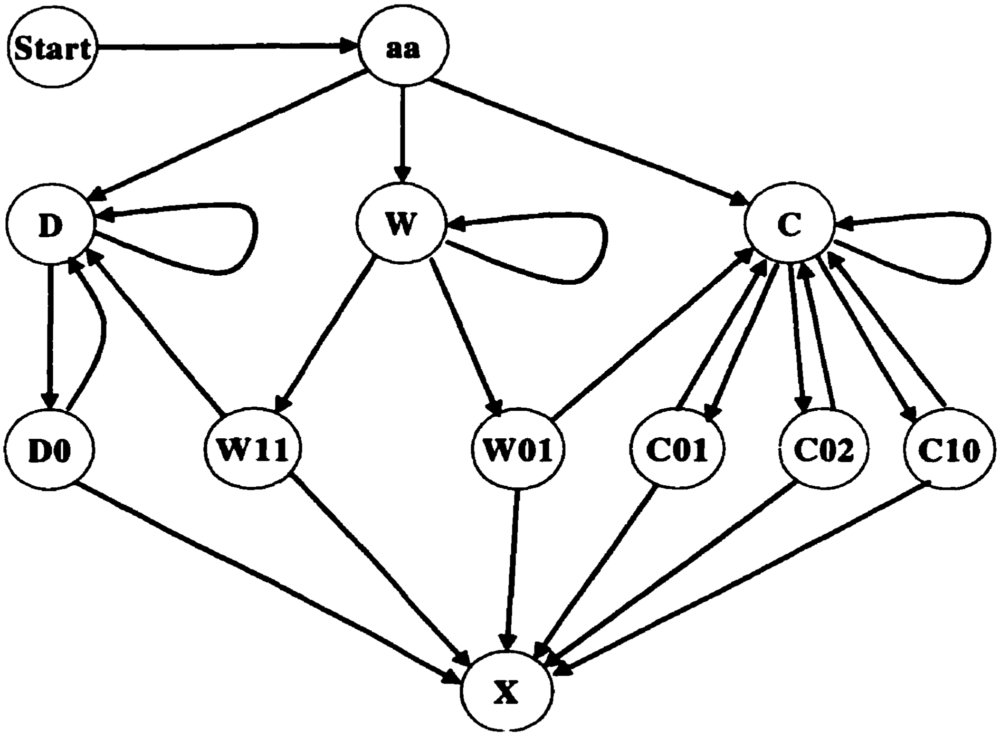


Figure 5.3 EFSM *S* in Distinguishing System

Appendix B also provides the SDL description of process *S*, In Figure 5.3 is the state graph of *S*, where aa stands for the state aa in Appendix B, D stands for Disconnected, W for Wait, C for Connected, D0 for Connected_0, W11 for Wait_1_1, W01 for Wait_0_1, C01 for Connected_0_1, C02 for Connected_0_2, C10 for Connected_1_0. X and Start node in Figure 5.3 have the same meanings as those in Figure 5.2.

Transitions in S are as following:

t(Start→aa), stands for the transition (Start → aa);

t(aa→D), for the transition (aa — /S3, Number:=zero → D);

t(aa→W)^{1}, for (aa — /S2, Number:=zero → W);

t(aa→W)^{2}, for (aa — /S1, Number:=un → W);

t(aa→C), for (aa — /S4, Number:=zero → C);

t(D→D0), for (D — CR/resume → D0);

t(D→D)^{1}, for (D — ICONresp/invalid → D);

t(D→D)^{2}, for (D — IDISreq/invalid → D);

t(D→D)^{3}, for (D — DT(Num,d)/invalid → D);

t(D0→D), for (D0 — ICONind/resume → D);

t(D0→DEADLOCK), for (D0 — */invalid → DEADLOCK);

t(C→C01), for (C — DT(Num,d), Num≠succ(Number)/resume, → C01),

t(C→C02), for (C — DT(Num,d), Num=succ(Number)/resume, → C02);

t(C→C10), for (C — CR/resume → C10);

t(C→C), for (C — IDISreq/invalid → C);

t(C01→C), for (C01 — AK(Num)/resume → C);

t(C01→DEADLOCK), for (C01 — */invalid → DEADLOCK);

t(C02→C), for (C02 — AK(Num)/resume, Number:=succ(Number) → C);

t(C02→DEADLOCK), for (D0 — */invalid → DEADLOCK);

t(C10→C), for (D0 — ICONind/resume → D);

t(C10→DEADLOCK), for (D0 — ICONind/invalid → DEADLOCK);

t(W→W01), for (W — ICONresp/resume, Number:=zero → W01);

t(W→W11), for (W — IDISreq/resume → W11);

t(W→W)^{1}, for (W — CR/invalid → W);

t(W→W)^{2}, for (W — DT(Num,d)/invalid → W);

t(W01→C), for (W01 — CC/resume → C);

t(W01→DEADLOCK), for (W01 — */invalid → DEADLOCK);

t(W11→D), for (W11 — DR/resume → D);

t(W11→DEADLOCK), for (W11 — */invalid → DEADLOCK);

Figure 5.4 is the state graph of Process Monitor in the distinguishing system.

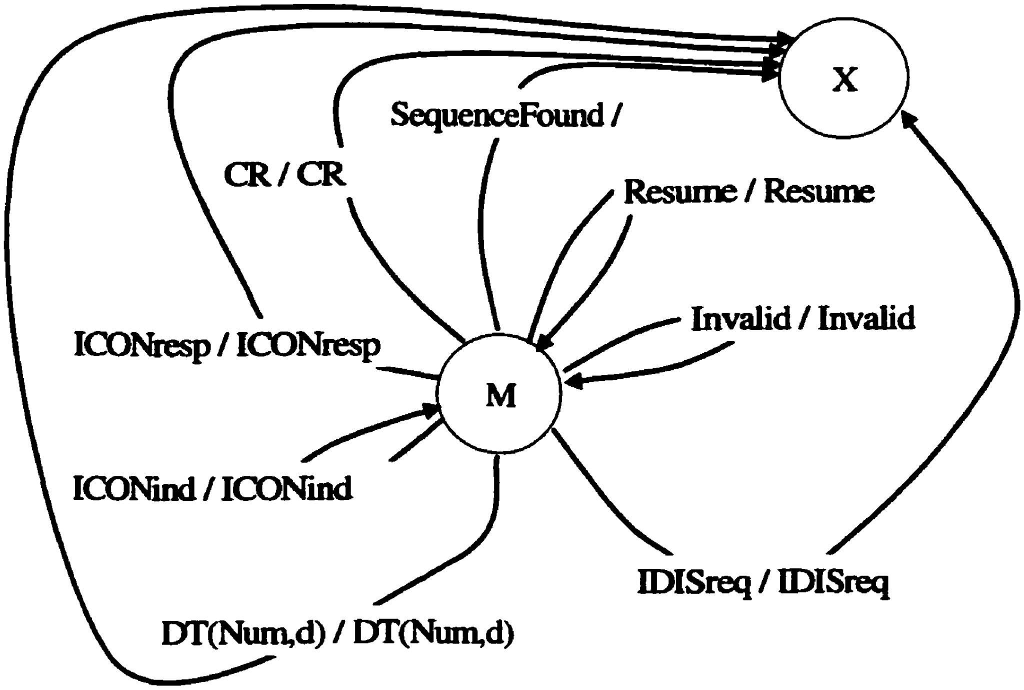


Figure 5.4 EFSM *M* in Distinguishing System

5.2. Results (CCS) and Analysis

The distinguishing system *D* is built to find CCSs which can confirm the configuration *c* in the set *C* of configurations, where:

$c = Connected(number = un)$

$C = \{ s_1, s_2, s_3, s_4 \}$, in which,

$s_1 = Wait(number = un)$

$s_2 = Wait(number = zero)$

$s_3 = Disconnected(number = zero)$

$s_4 = Connected(number = zero)$

From the distinguishing systems, where the process *S* has to be initialized in four different configurations.

Exhaustive simulation of *D* in Breadth First Mode with an upper bound set to 4 by using ObjectGeode simulator, the state graph will be generated and dumped into a file. Then we run post-processing program (set the upper bound of state number that will be searched along a path to 1000, this can be adjusted by the user) on the dump file and get all minimum separating sequences (total 118 sequences). Finally, we find 4 CCSs, they are: *DT(un, true)*, *DT(un, false)*, *DT(zero, true)* and *DT(zero, false)*.

The specification of the distinguishing system *D* of the Responder process in INRES protocol, post-processing program and minimal separating sequences found are given in Appendix B, C and D, respectively.

Chapter 6

Conclusion and Future Work

In this thesis the CCS generation problem has been analyzed and an approach to generate CCS for the configurations of a given SDL specification by using ObjectGeode simulator is presented. The main point is the exhaustive simulation provided by this tool. The correctness of the approach shown in Figure 2.1 is demonstrated on INRES protocol. The advantage of this approach is that a commercial tool can be used to solve the problem.

Future work is to automate the process of constructing a distinguishing system.

References

[1] C J. Wang, etc., "Protocol Validation Tools as Test Case Generators", Ohio State University, 1994

[2] Verilog Company, "ObjectGeode SDL Simulator Reference Manual, Version 3.2", France, 1997

[3] Verilog Company, "ObjectGeode Tutorial", France, 1997

[4] D. Lee & M. Yannakakis, "Principles and Methods of Testing Finite State Machines-A Survey", Bell Lab, 1996

[5] A. Petrenko, S. Boroday, R. Groz, "Confirming configurations in EFSM", CRIM, 1999

[6] K.J.Turner, "Using Formal Description Techniques", John Wiley & Sons, 1993

[7] A. Gill, "Introduction to the Theory of Finite State Machine", McGraw-Hill, 1962

[8] G. J. Holzmann, "Design and Validation of Computer Protocols", Prentice Hall Software Series, 1991

[9] Verilog Company, ObjectGeode On-line Help Documents

[10] C.A.R.Hoare, "Communicating sequential Processes", Prentice-Hall, 1985

[11] T.H.Corman, etc. "Introduction to algorithms", The MIT Press, 1997

[12] A. Petrenko, S. Boroday, "Final Report of the project VERA", CRIM, July, 1999

[13] ITU, "Z.100 Standard", 1992

[14] R.J.Linn, etc., Ïmprovements on UIO Sequence Generation and Partial UIO Sequence", IFIP 1992, Elsevier Science Publisher B.V., 1992

[15] O.Monkewich, "SDL-based Specification and Testing Strategy for Communication Network Protocol", Elsevier Science B.V., 1999

[16] A.Kerbrat, etc., "Automated test generation from SDL specifications", Elsevier Science B.V., 1999

[17] ETSI, "Use of SDL in European Telecommunications Standards – rules for testability and facilitating validation", ETS 300 414, 1995

[18] A.Cavalli, etc., "Test generation for the SSCOP-ATM networks protocol", Proceeding of SDL forum'97, Elsevier Science, 1997

[19] ITU-T Recommendation Q.2110, ATM Adaption Layer-Service Speicific Connection Oriented Protocol (SSCOP), 1994

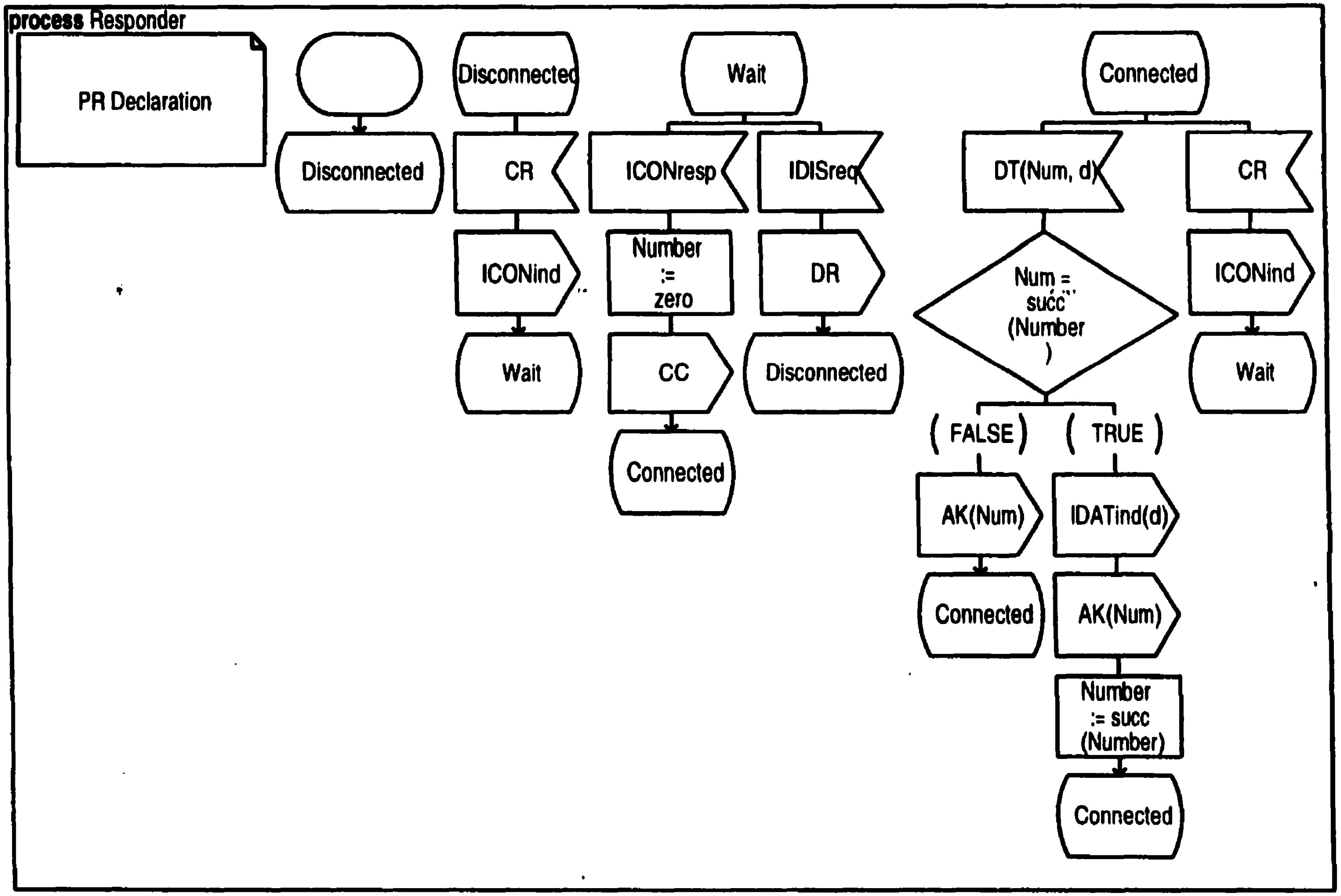
[20] D.Tasak, "Specification and validation of Q.2931 ATM signaling protocol using estelle", Master thesis, 1997

[21] J.L.Peterson, "Net theory and the modeling of systems", Prentice-Hall inc., 1981

[22] S.J.Hong, "Existence Algorithms for Synchronizing/Distinguishing Sequences", IEEE Transactions on Computer, Vol. C-30, No.3. March 1991

Appendix A

INRES Protocol Responder Process In SDL


process Responder
PR Declaration
Disconnected
Disconnected
CR
ICONind
Wait
Wait
ICONresp
Number := zero
CC
Connected
IDISreq
DR
Disconnected
Connected
DT(Num, d)
Num = succ (Number)
FALSE
AK(Num)
Connected
TRUE
IDATind(d)
AK(Num)
Number := succ (Number)
Connected
CR
ICONind
Wait

Appendix B

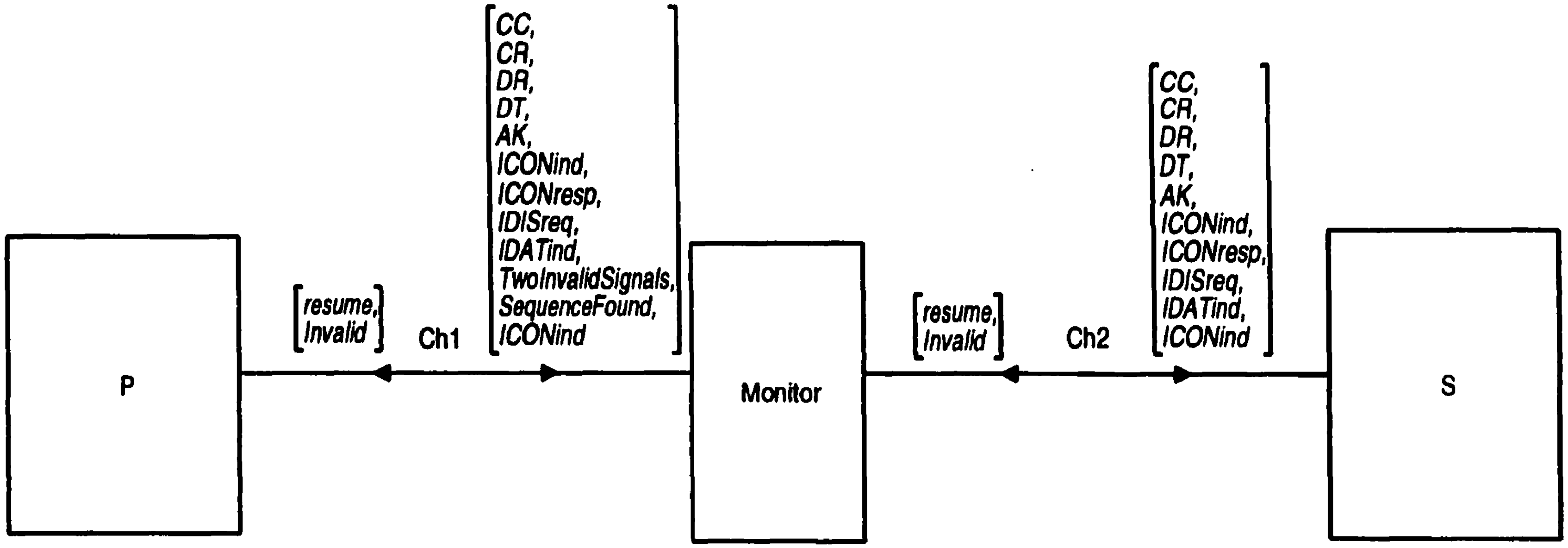
Distinguishing System of INRES Protocol
Responder Process
In SDL

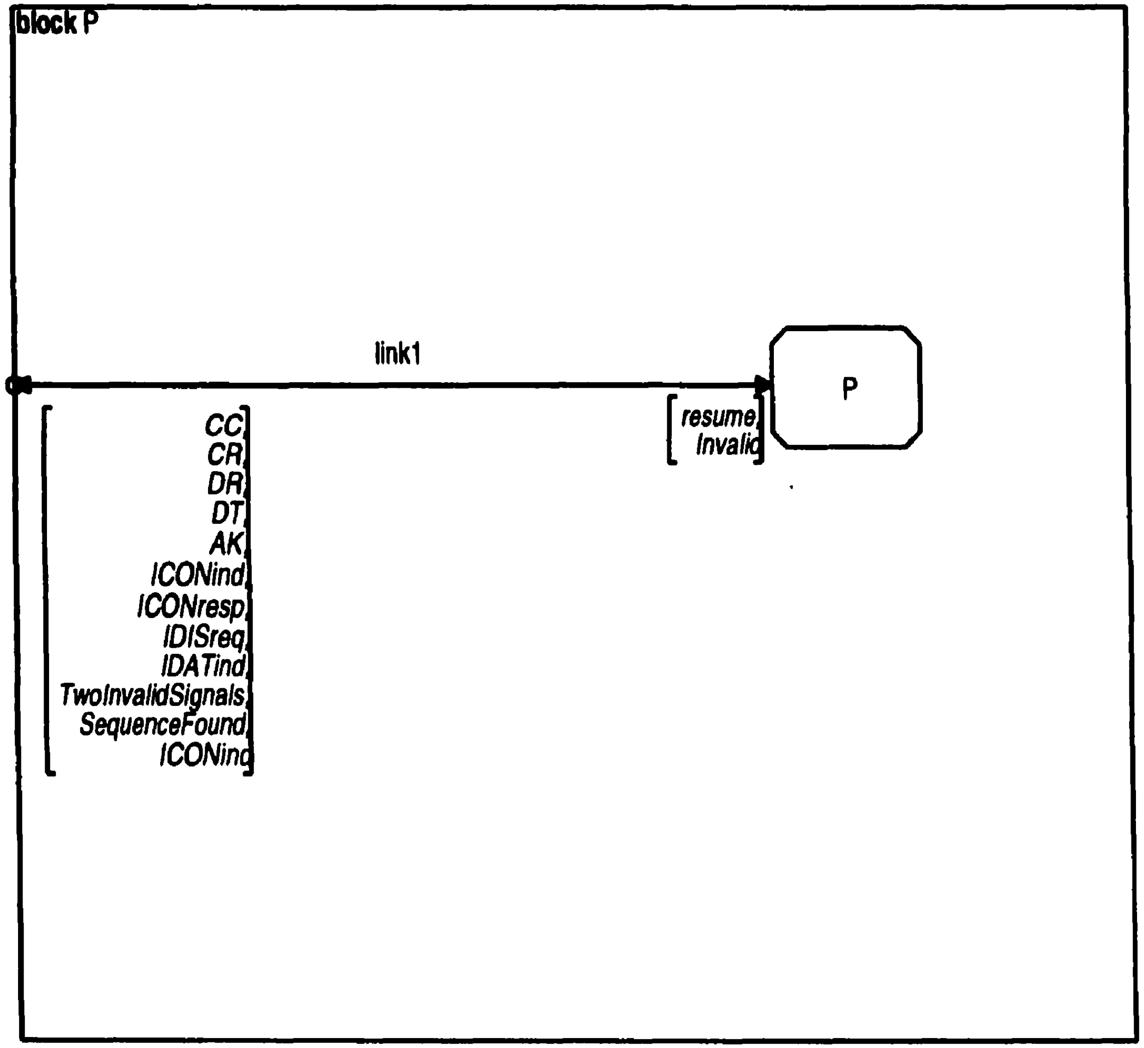
system DistinguishingSystem

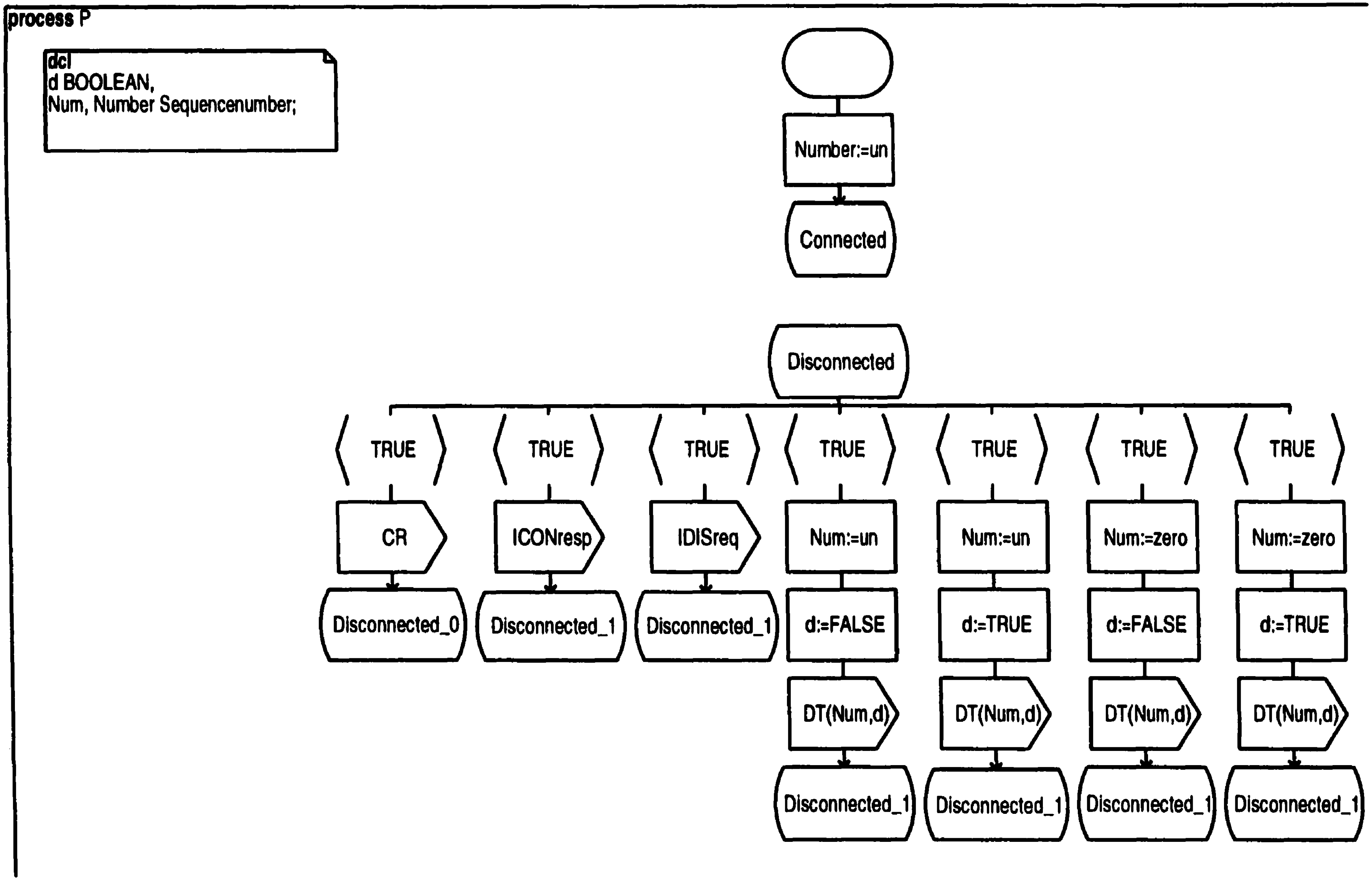
```
newtype Sequencenumber
literals zero,un
operators
 succ: Sequencenumber -> Sequencenumber
axioms
succ(un)==zero;
succ(zero)==un;
endnewtype Sequencenumber;
```

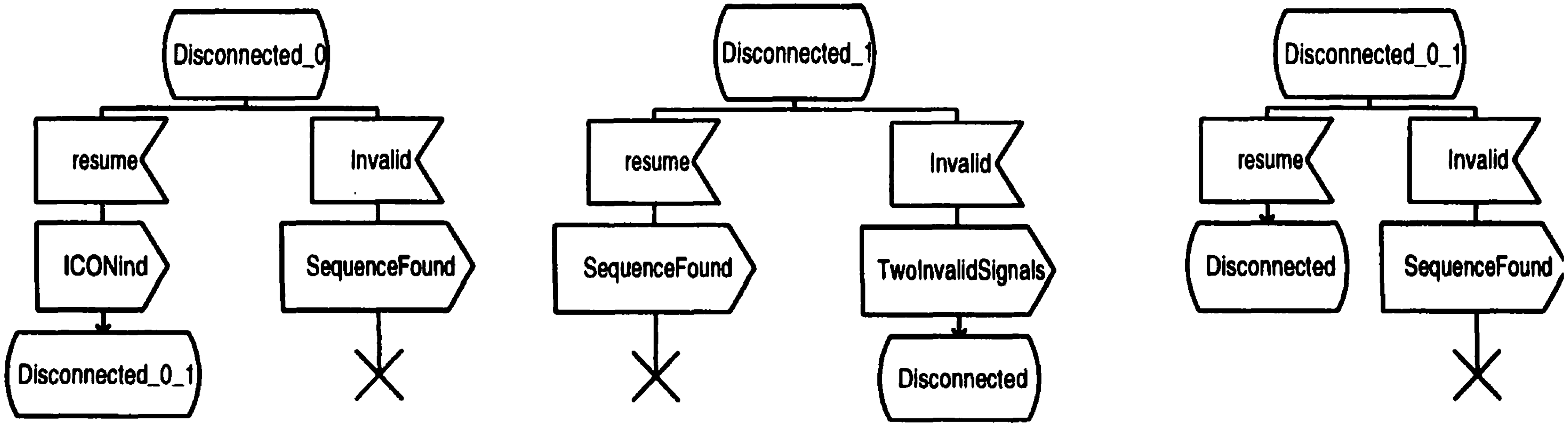
```
signal
resume,
Invalid,
TwoInvalidSignals,
SequenceFound,
CC,
CR,
DR,
DT(Sequencenumber, Boolean),
AK(Sequencenumber),
ICONresp,
ICONind,
ICONind,
IDISreq,
IDATInd(Boolean);
```

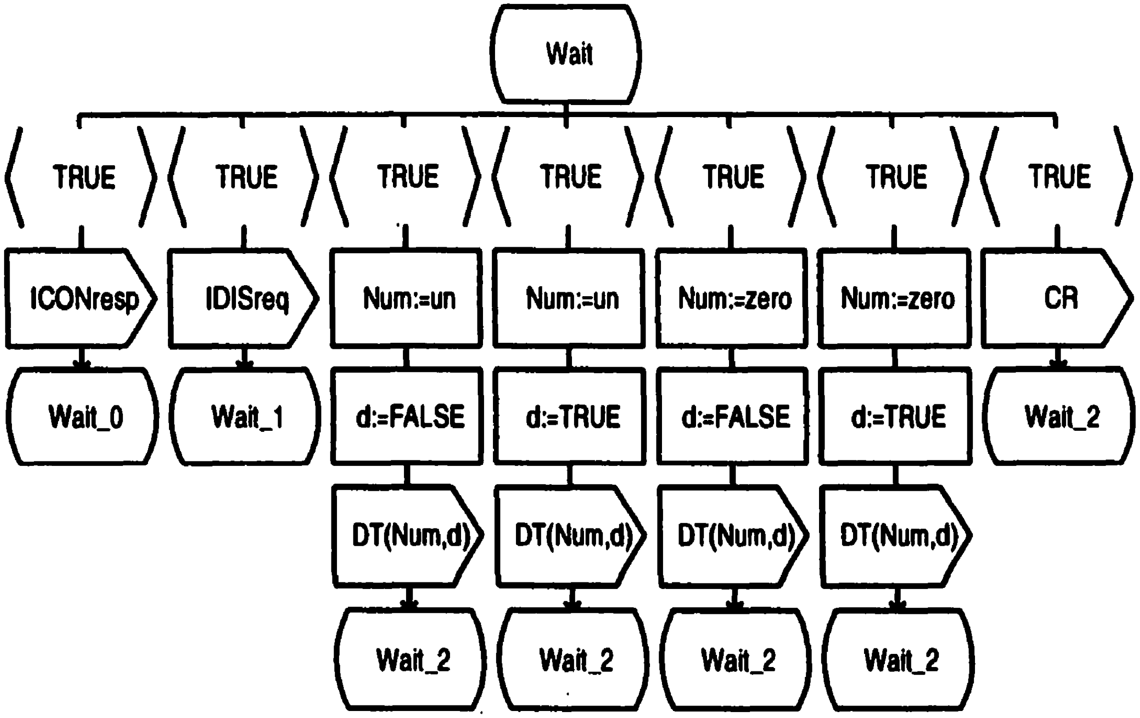
```
newtype IPDUTyp
literals CR, CC, DR, DT, AK
endnewtype IPDUTyp;

newtype MSDUTyp
struct
id IPDUTyp;
Num Sequencenumber;
Data BOOLEAN;
endnewtype MSDUTyp;
```

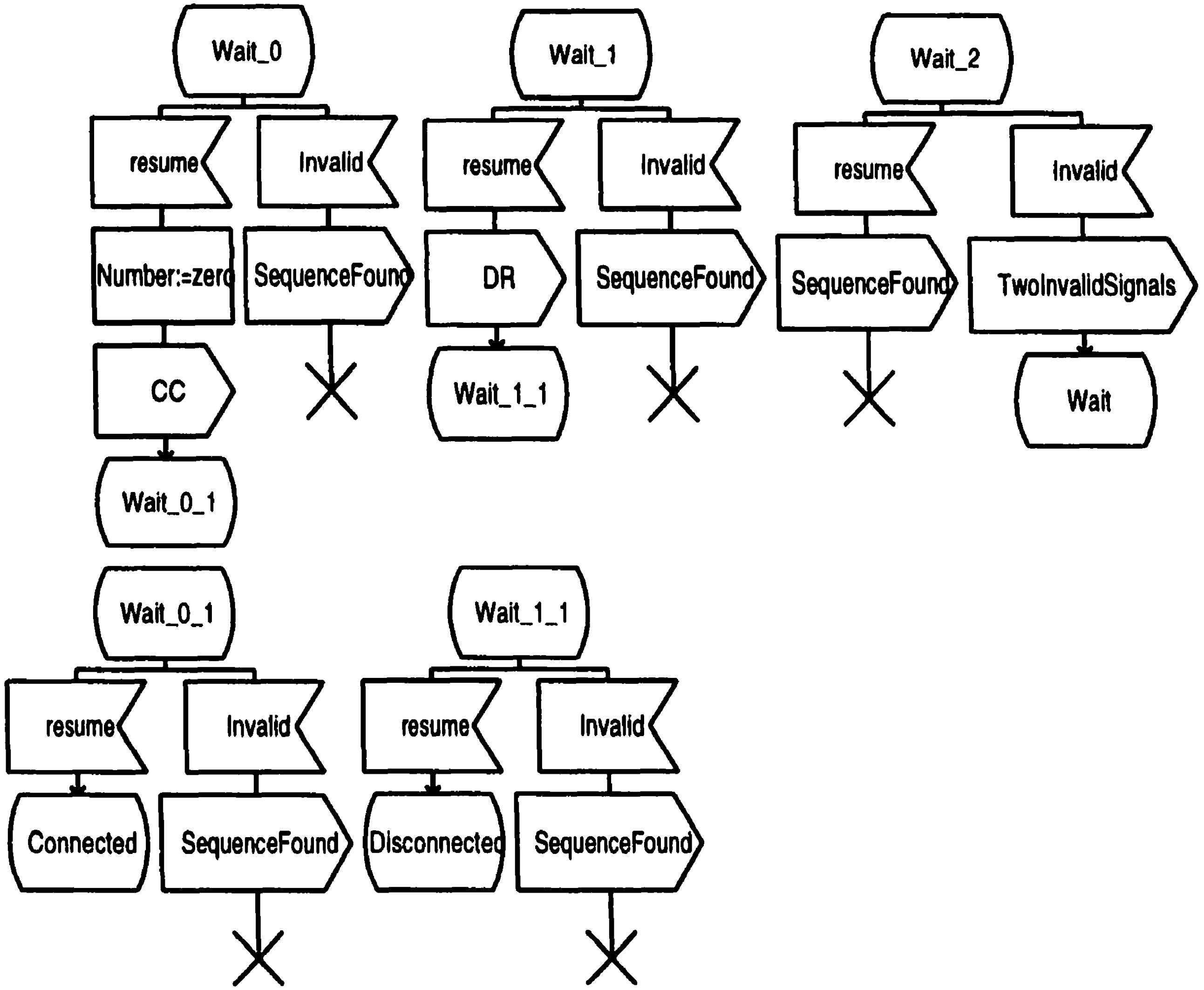

[resume, Invalid]
Ch1
[CC, CR, DR, DT, AK, ICONind, ICONresp, IDISreq, IDATind, TwoInvalidSignals, SequenceFound, ICONind]
P
Monitor
[resume, Invalid]
Ch2
[CC, CR, DR, DT, AK, ICONind, ICONresp, IDISreq, IDATind, ICONind]
S

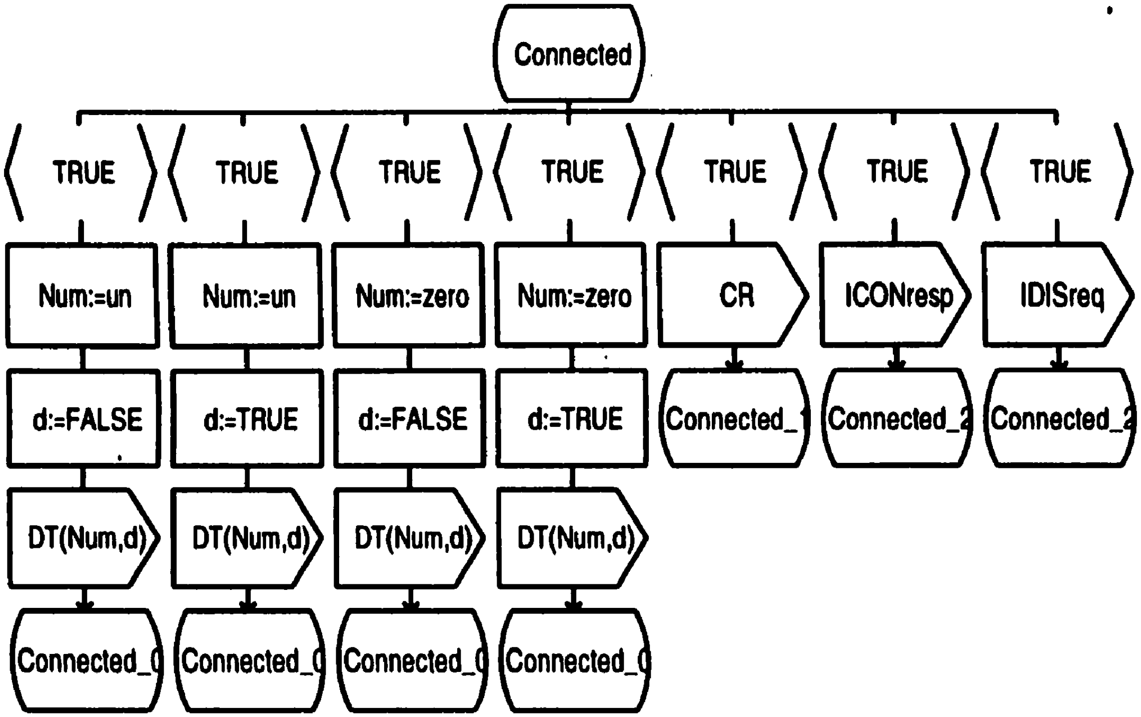

block P
link1
CC,
CR,
DR,
DT,
AK,
ICONind,
ICONresp,
IDISreq,
IDATind,
TwoInvalidSignals,
SequenceFound,
ICONinc
resume,
Invalic
P

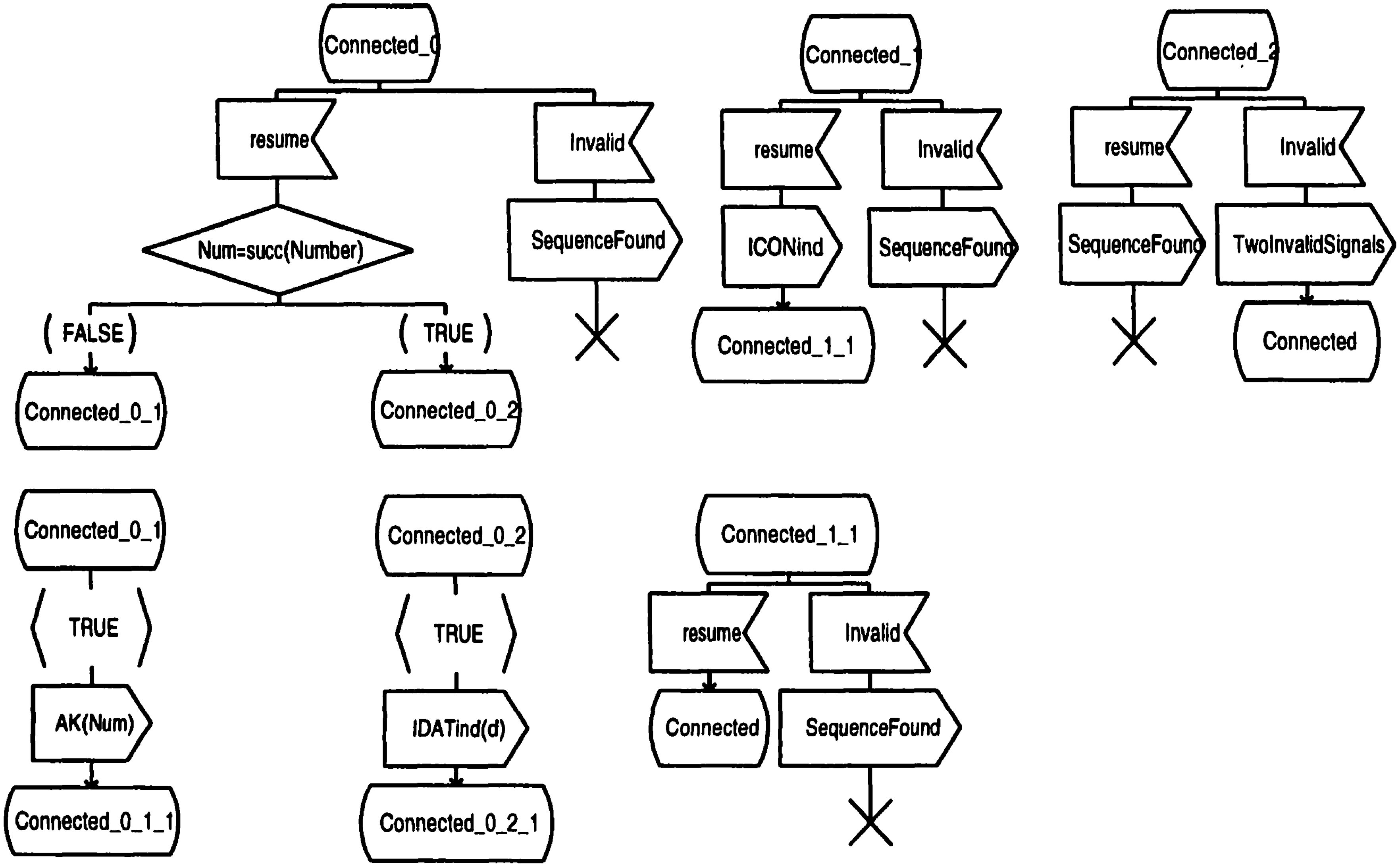

process P
dcl
d BOOLEAN,
Num, Number Sequencenumber;
Number:=un
Connected
Disconnected
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
CR
ICONresp
IDISreq
Num:=un
Num:=un
Num:=zero
Num:=zero
Disconnected_0
Disconnected_1
Disconnected_1
d:=FALSE
d:=TRUE
d:=FALSE
d:=TRUE
DT(Num,d)
DT(Num,d)
DT(Num,d)
DT(Num,d)
Disconnected_1
Disconnected_1
Disconnected_1
Disconnected_1

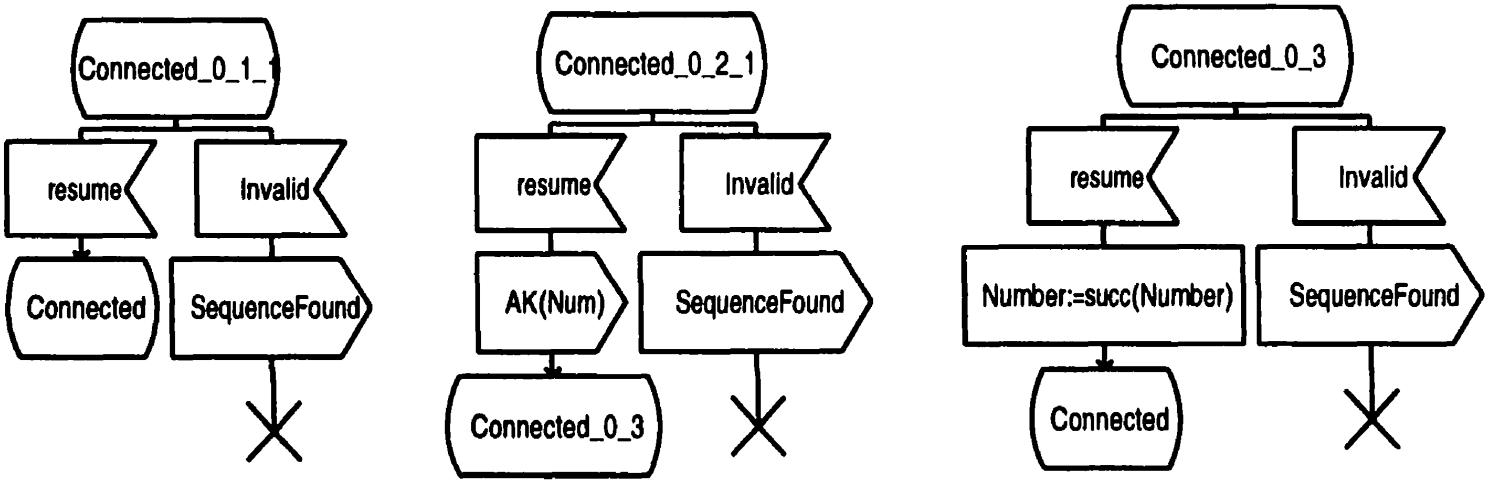

Disconnected_0
resume
Invalid
ICONInd
SequenceFound
Disconnected_0_1
Disconnected_1
resume
Invalid
SequenceFound
TwoInvalidSignals
Disconnected
Disconnected_0_1
resume
Invalid
Disconnected
SequenceFound


Wait
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
ICONresp
IDISreq
Num:=un
Num:=un
Num:=zero
Num:=zero
CR
Wait_0
Wait_1
d:=FALSE
d:=TRUE
d:=FALSE
d:=TRUE
Wait_2
DT(Num,d)
DT(Num,d)
DT(Num,d)
DT(Num,d)
Wait_2
Wait_2
Wait_2
Wait_2


Wait_0
resume
Invalid
Number:=zero
SequenceFound
CC
Wait_0_1
Wait_1
resume
Invalid
DR
SequenceFound
Wait_1_1
Wait_2
resume
Invalid
SequenceFound
TwoInvalidSignals
Wait
Wait_0_1
resume
Invalid
Connected
SequenceFound
Wait_1_1
resume
Invalid
Disconnected
SequenceFound

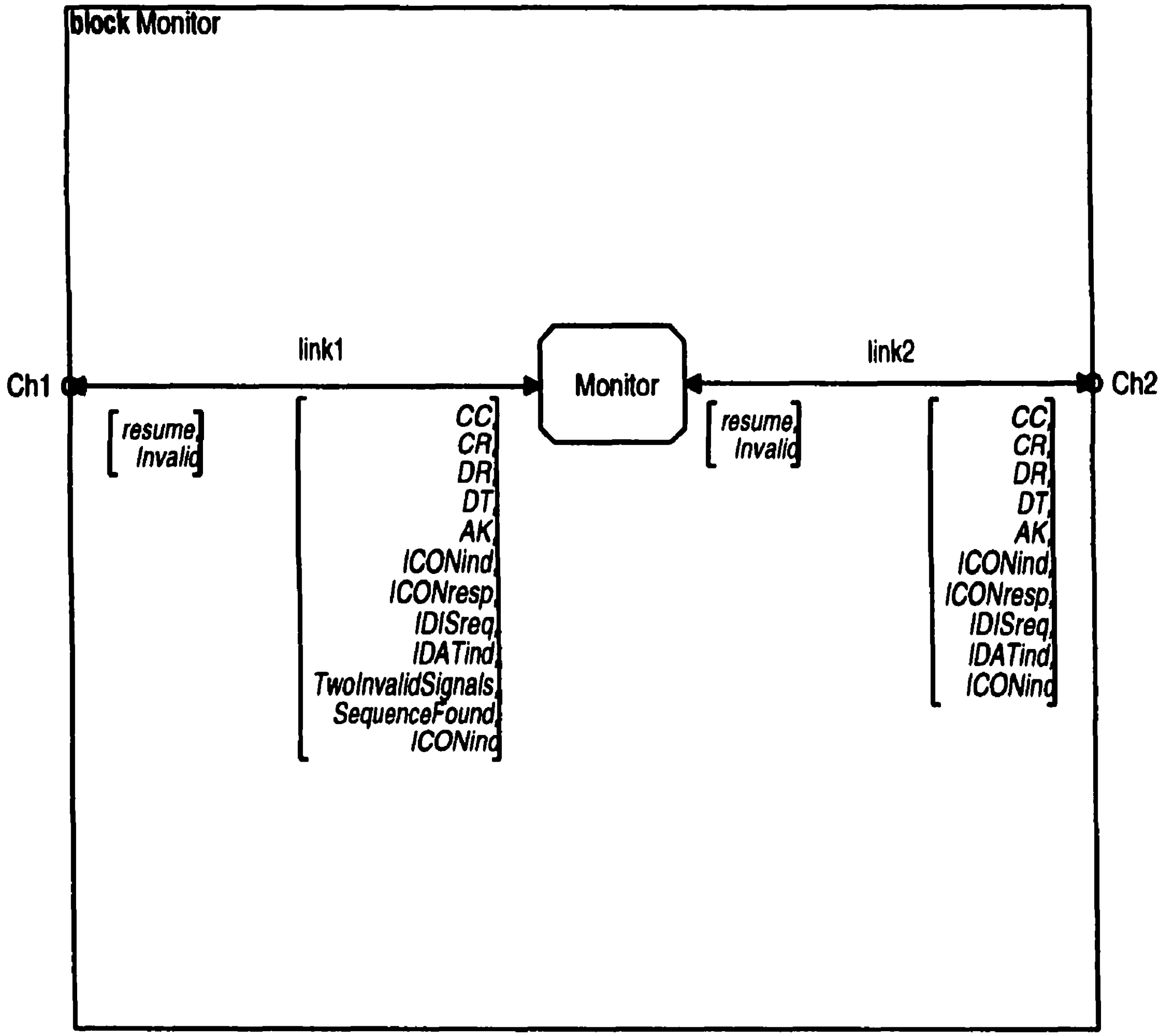

Connected
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
TRUE
Num:=un
Num:=un
Num:=zero
Num:=zero
CR
ICONresp
IDISreq
d:=FALSE
d:=TRUE
d:=FALSE
d:=TRUE
Connected_1
Connected_2
Connected_2
DT(Num,d)
DT(Num,d)
DT(Num,d)
DT(Num,d)
Connected_0
Connected_0
Connected_0
Connected_0


Connected_0
resume
Invalid
Num=succ(Number)
SequenceFound
FALSE
TRUE
Connected_0_1
Connected_0_2
Connected_1
resume
Invalid
ICONind
SequenceFound
Connected_1_1
Connected_2
resume
Invalid
SequenceFound
TwoInvalidSignals
Connected
Connected_0_1
TRUE
AK(Num)
Connected_0_1_1
Connected_0_2
TRUE
IDATind(d)
Connected_0_2_1
Connected_1_1
resume
Invalid
Connected
SequenceFound


Connected_0_1_1
resume
Invalid
Connected
SequenceFound
Connected_0_2_1
resume
Invalid
AK(Num)
SequenceFound
Connected_0_3
Connected_0_3
resume
Invalid
Number:=succ(Number)
SequenceFound
Connected

Monitor

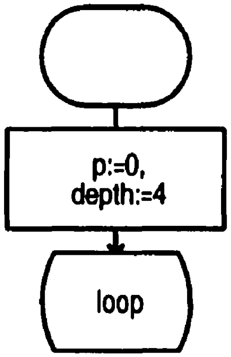
Sat Jul 17 04:05:37 1999

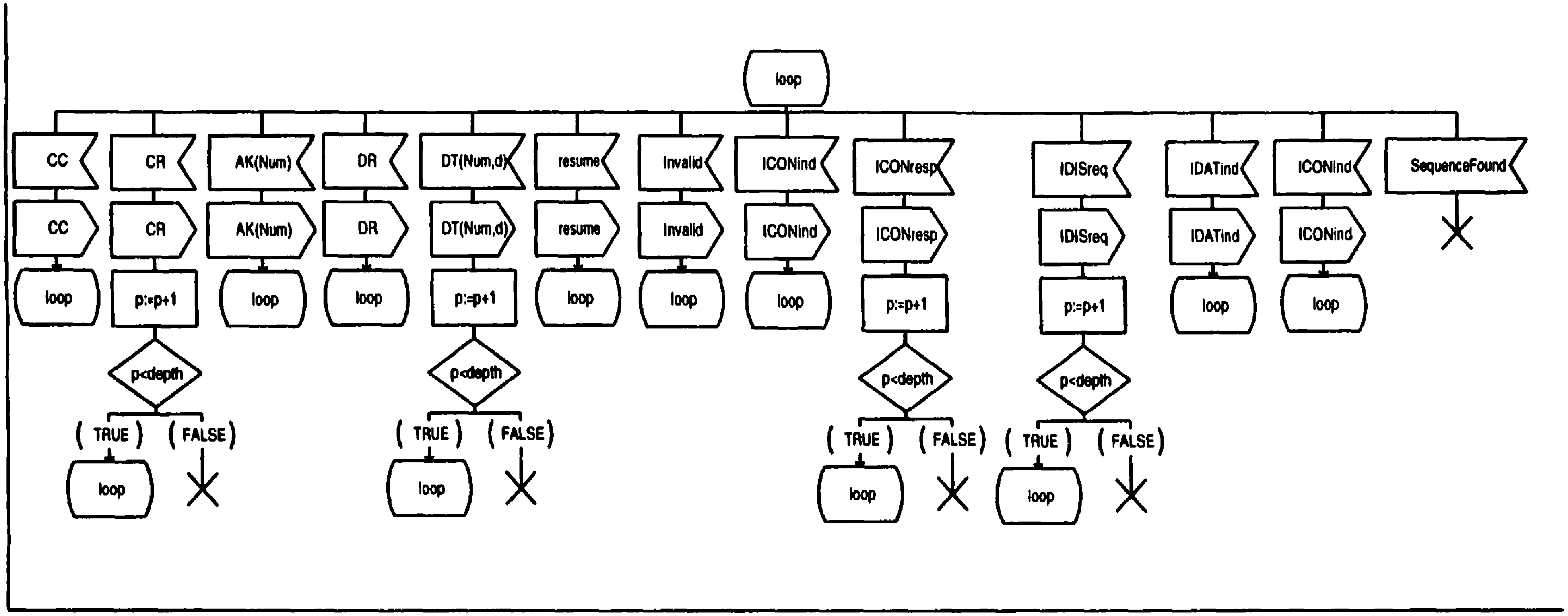


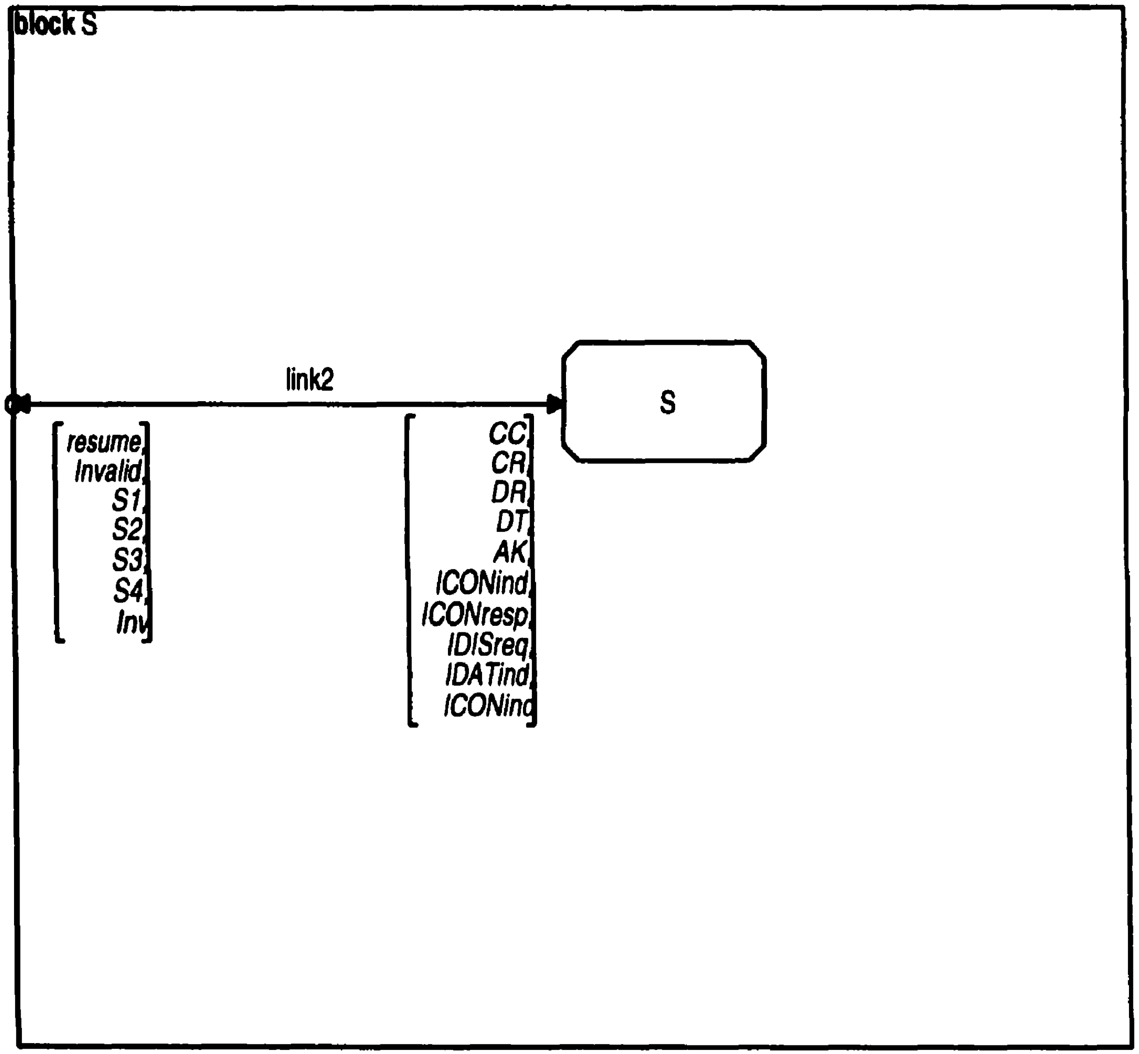
/home/poirot/zjhu/inres/testCCS.pr

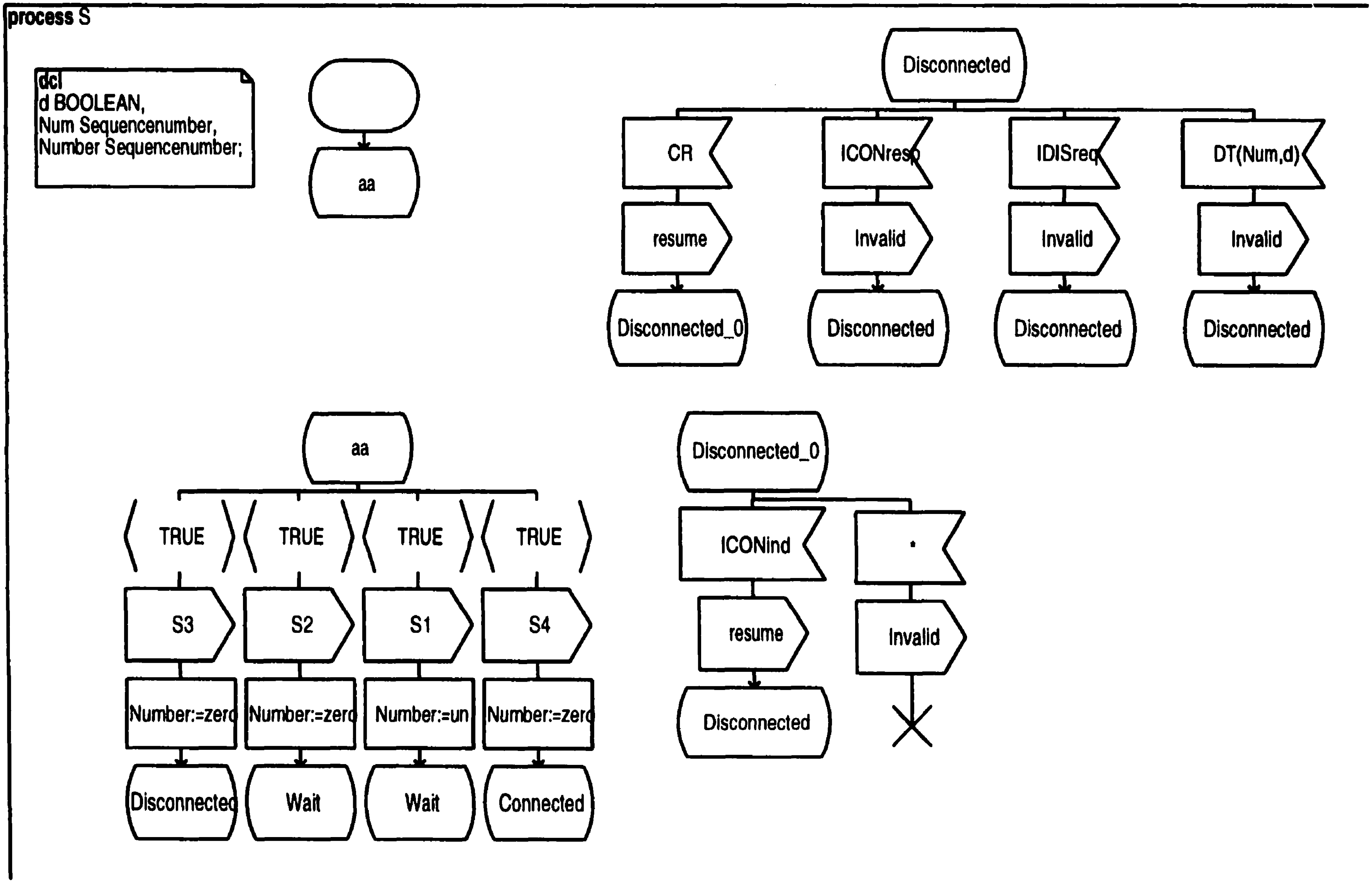
process Monitor

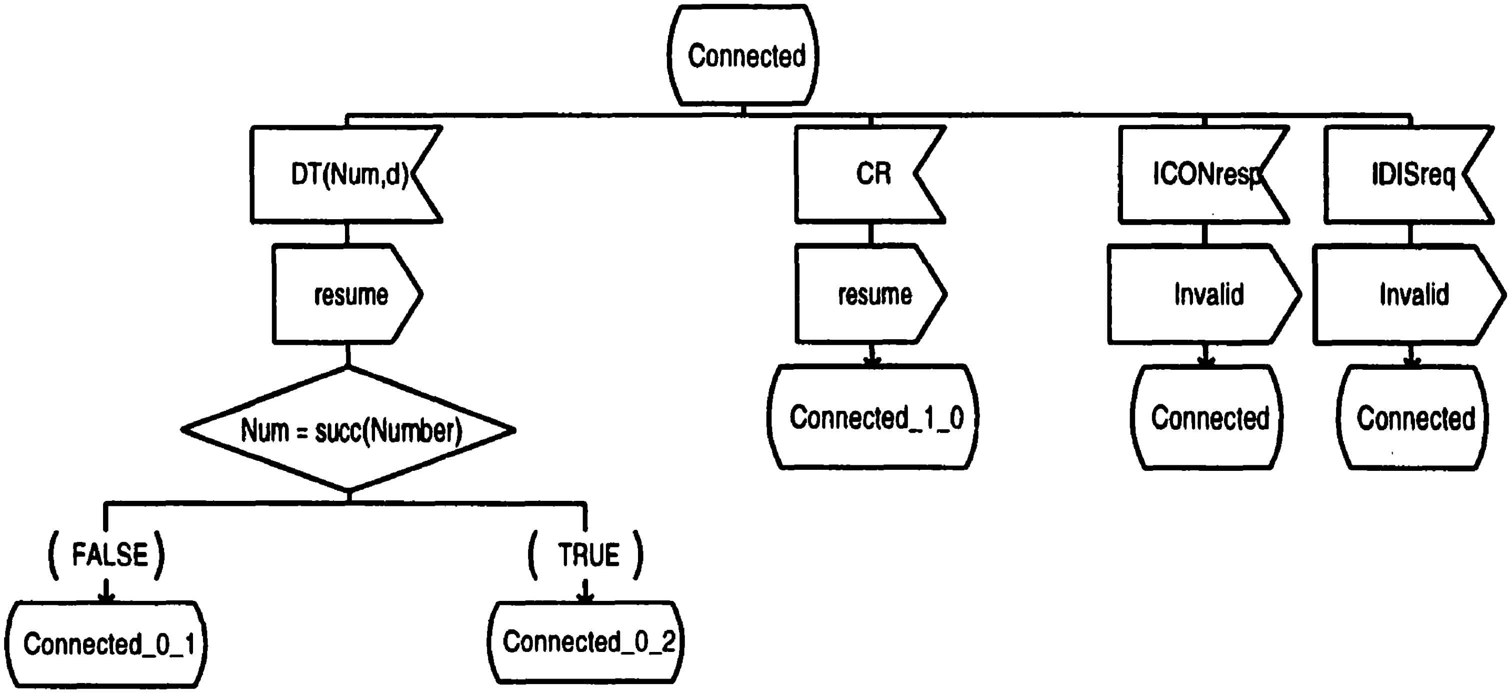
```
dcl
d BOOLEAN,
Num, Number Sequencenumber,
p integer,
depth integer;
```

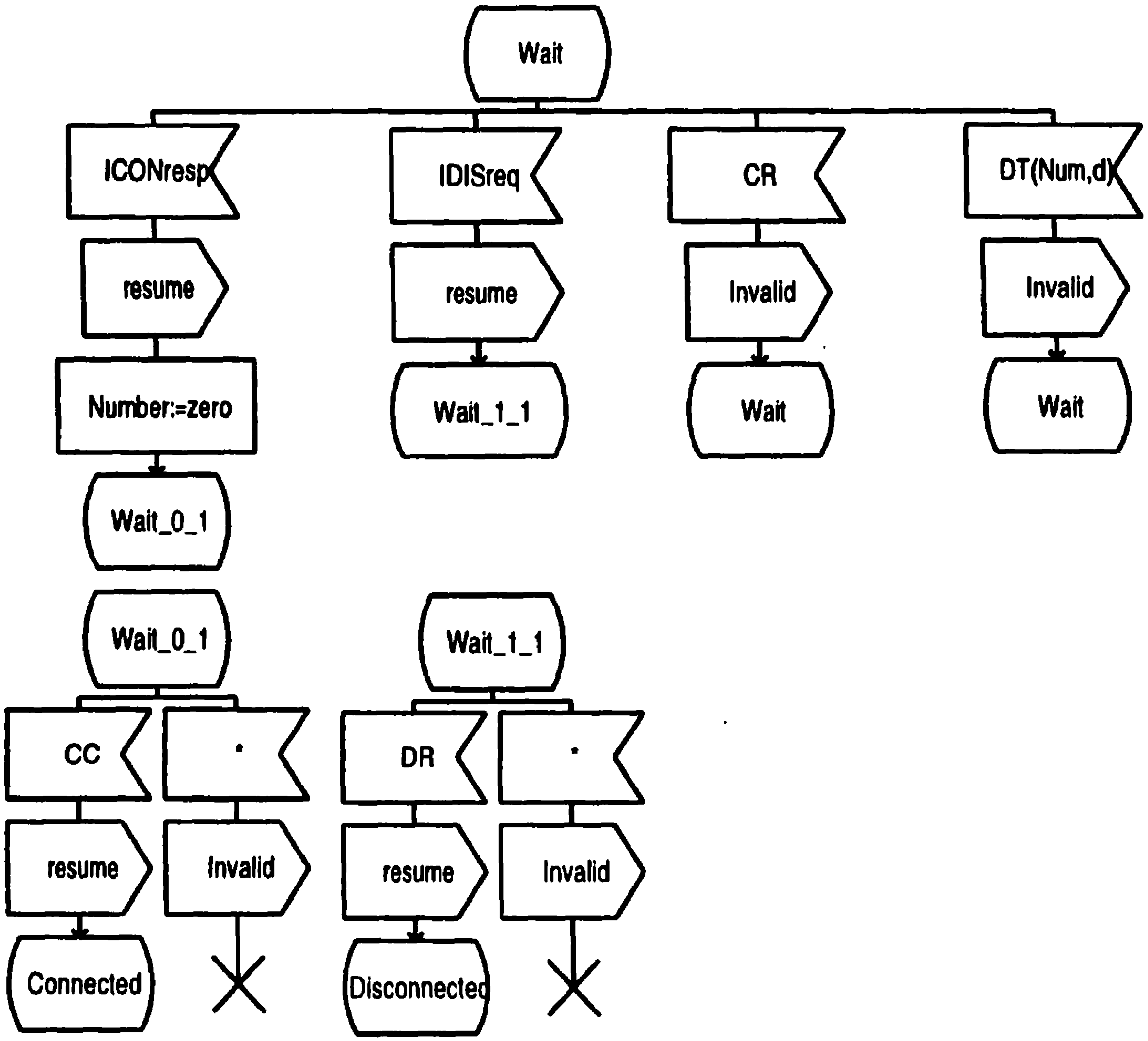


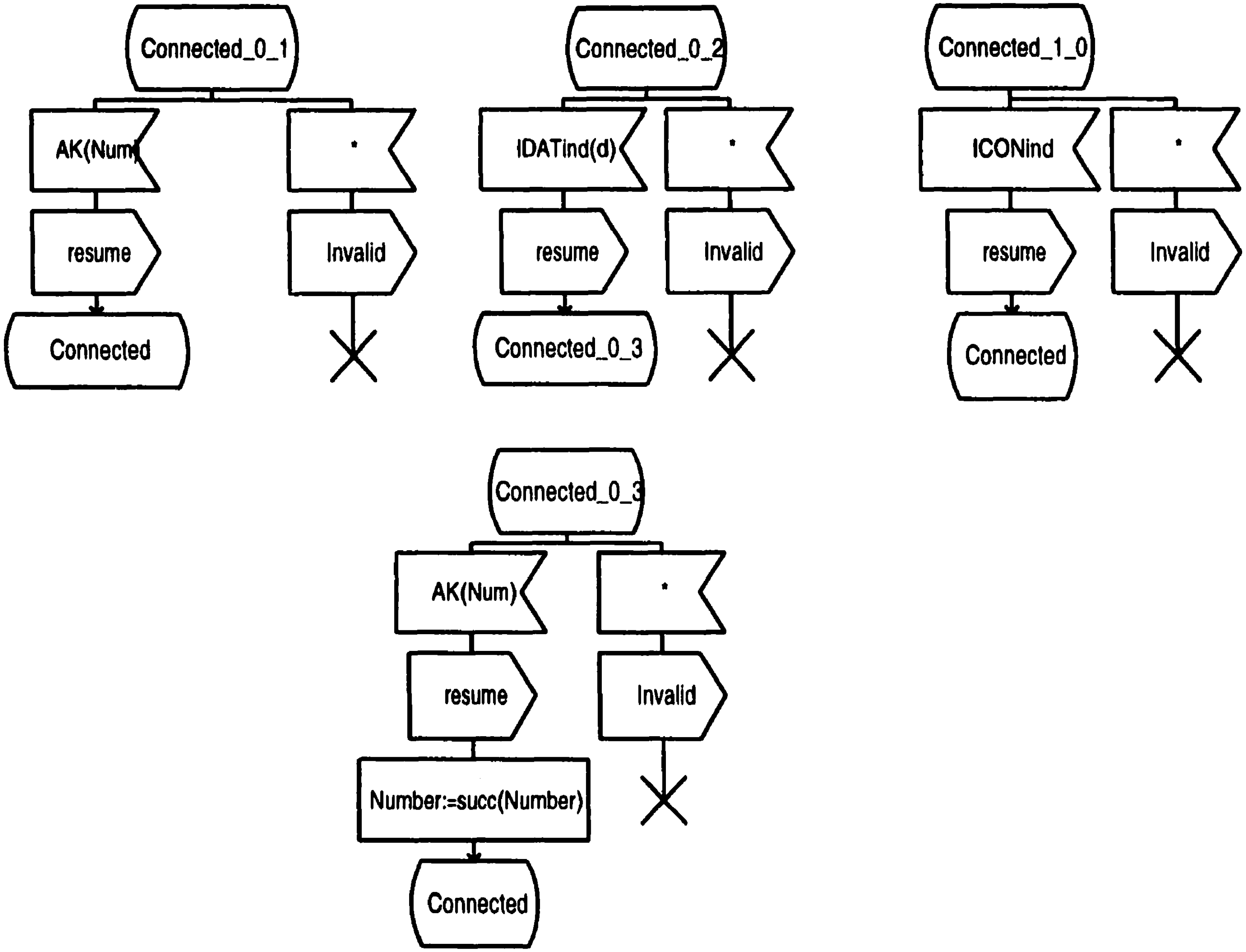

loop
CC
CC
loop
CR
CR
p:=p+1
p<depth
(TRUE)
loop
(FALSE)
AK(Num)
AK(Num)
loop
DR
DR
loop
DT(Num,d)
DT(Num,d)
p:=p+1
p<depth
(TRUE)
loop
(FALSE)
resume
resume
loop
Invalid
Invalid
loop
ICONind
ICONind
loop
ICONresp
ICONresp
p:=p+1
p<depth
(TRUE)
loop
(FALSE)
IDISreq
IDISreq
p:=p+1
p<depth
(TRUE)
loop
(FALSE)
IDATind
IDATind
loop
ICONind
ICONind
loop
SequenceFound


block S
link2
S
resume,
Invalid,
S1,
S2,
S3,
S4,
Inv
CC,
CR,
DR,
DT,
AK,
ICONind,
ICONresp,
IDISreq,
IDATind,
ICONind


process S
dcl
d BOOLEAN,
Num Sequencenumber,
Number Sequencenumber;
aa
Disconnected
CR
resume
Disconnected_0
ICONresp
Invalid
Disconnected
IDISreq
Invalid
Disconnected
DT(Num,d)
Invalid
Disconnected
aa
TRUE
S3
Number:=zero
Disconnected
TRUE
S2
Number:=zero
Wait
TRUE
S1
Number:=un
Wait
TRUE
S4
Number:=zero
Connected
Disconnected_0
ICONind
resume
Disconnected
*
Invalid


Connected
DT(Num,d)
resume
Num = succ(Number)
FALSE
Connected_0_1
TRUE
Connected_0_2
CR
resume
Connected_1_0
ICONresp
Invalid
Connected
IDISreq
Invalid
Connected


Wait
ICONresp
resume
Number:=zero
Wait_0_1
IDISreq
resume
Wait_1_1
CR
Invalid
Wait
DT(Num,d)
Invalid
Wait
Wait_0_1
CC
resume
Connected
*
Invalid
Wait_1_1
DR
resume
Disconnected
*
Invalid


Connected_0_1
AK(Num)
resume
Connected
*
Invalid
Connected_0_2
IDATind(d)
resume
Connected_0_3
*
Invalid
Connected_1_0
ICONind
resume
Connected
*
Invalid
Connected_0_3
AK(Num)
resume
Number:=succ(Number)
Connected
*
Invalid

Appendix C

Code For

Post-Processing Program

```
// CCS.cpp: find Configuration Confirming Sequence
//          1) First, find out Minimal Separating Sequences.
//          2) Second, select Configuration Confirming Sequence
//                   from Minimal Separating Sequences.
//
///////////////////////////////////////////////////////////////////////
#include "ctype.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

///////////////////////////////////////////////////////////////////////
// Construction/Destruction
//
//  TREE is the data structure to save adjacent list of state graph.
///////////////////////////////////////////////////////////////////////

class TREE
{
public:
        long    NodeNumber;
        char    Msg[100];
        TREE    *pSibling;
        TREE    *pSon;

        int     clr;

public:
        TREE();
        TREE(long);
        TREE(long, char*);
        virtual ~TREE();

        TREE * find(long x);
        TREE * addnew(long x);
        void appendson(long x, char* msg);
};

TREE::TREE()
{
        NodeNumber = -1;
        strcpy(Msg, "");
        clr = 0;
        pSibling = NULL;
        pSon = NULL;
}

TREE::TREE(long x)
{
        NodeNumber = x;
        strcpy(Msg, "");
        clr = 0;
        pSibling = NULL;
        pSon = NULL;
}

TREE::TREE(long x, char *strMsg)
{
        NodeNumber = x;
        strcpy(Msg, strMsg);
        clr = 0;
        pSibling = NULL;
        pSon = NULL;
}

TREE::~TREE()
{

}

TREE* TREE::find(long x)
{
        TREE    *pNode;

        for(pNode = this; pNode != NULL; pNode = pNode->pSibling)
                if(pNode->NodeNumber == x) return pNode;

        return NULL;
}

TREE* TREE::addnew(long x)
{
        TREE    *pNode;

        pNode = this;
        for(pNode = this; pNode->pSibling != NULL;
            pNode = pNode->pSibling);
        pNode->pSibling = new TREE(x);

        return pNode->pSibling;
}

void TREE::appendson(long x, char* msg)
{
        TREE    *pNode;

        pNode = this;
        for(pNode = this; pNode->pSon != NULL;
            pNode = pNode->pSon);
        pNode->pSon = new TREE(x, msg);
}

FILE *f;
TREE *pRoot;
char MSSes[1000][200];
int  MSSptr = 0;
long  MAXDEPTH = 50L;
char CCScand[1000][200];
int  CCSptr = 0;

int     getline();
long    getnumber();
void    getMSS(char*);
int     addMSS(char*);

int ConstructAdjList(char *fileName)
{
        int   c;
        long  x, y;
        char  *msg;
        TREE  *Tnode;

        msg = (char *)malloc(100);
        f=fopen(fileName,"r");
        if(f==NULL)
        {
                printf("\nerror!\n");
                return -1;
        }
        Tnode = new TREE();

        while(getline()>0)
        {
                x = getnumber();
                Tnode = pRoot->find(x);
                if(Tnode == NULL) Tnode = pRoot->addnew(x);

                y = getnumber();
                getMSS(msg);
                Tnode->appendson(y, msg);
                strcpy(msg, "");
        }

        fclose(f);
        free(msg);
        return 0;
}

int getline()
{
        int c;

        while(1)
        {
                if((c=getc(f)) == EOF) { c=0; break; }
                if(c!='$')      continue;
                else { c=1; break; }
        }

        return c;
}

long getnumber()
{
        int c, i;
        char num[10];

        c=getc(f);
        for(i=0; isdigit(c); i++)
        {
            num[i] = c; c=getc(f);
        }
        num[i]='\0';

        //printf("\ngetnum: %s", num);
        return atol(num);
}

void getMSS(char *msg)
{
```

```
int c, i;
char *opd1, *opd2, *opd3, *opd4;
char *opd21, *opd22, *opd23;
char *outpar;
long fLoc=0L;

opd1 = (char*)malloc(50*sizeof(char));
opd2 = (char*)malloc(50*sizeof(char));
opd3 = (char*)malloc(50*sizeof(char));
opd4 = (char*)malloc(50*sizeof(char));
opd21 = (char*)malloc(50*sizeof(char));
opd22 = (char*)malloc(50*sizeof(char));
opd23 = (char*)malloc(50*sizeof(char));
outpar = (char*)malloc(100*sizeof(char));

while(1)
{
  c=getc(f);
  if(c==EOF) { fseek(f, -1, SEEK_CUR); break; }
  if(c=='$') { fseek(f, -1, SEEK_CUR); break; }

  if(c=='o')
    {

      fscanf(f, "utput %s%s%s", opd2, opd3, opd4);

      if(strcmp(opd4, "responder(1)")==0)
      {
          // read output parameters
          strcpy(outpar, "(");
          for(i=0; ; i++)
          {
             for(c=getc(f); c!='$' &&
                   c!='\n' && c!=EOF; c=getc(f));

             if(c=='$' || c==EOF)
               { fseek(f, -1, SEEK_CUR); break; }

             fLoc = ftell(f);
             fscanf(f, "%s%s%s", opd21, opd22, opd23);
             if(*opd21=='p' && strcmp(opd22, "=")==0)
               {
                 strcat(outpar, opd23);
                 strcat(outpar, ",");
               }
             else
               {
                 if(fseek(f, fLoc, SEEK_SET)<0)
                   printf("error seek\n");

                 break;
               }
          }
          if(i>0) *(outpar+strlen(outpar)-1) = ')';
          else *(outpar+strlen(outpar)-1) = '\0';

          // record MSS
          sprintf(msg, "%s%s", opd2, outpar);
      }
      else if(*opd2=='s' && isdigit(*(opd2+1))
                      && strcmp(opd4, "responderminus(1)")==0)
                  strcpy(msg, opd2);

              break;
            }
          else
            for(c=getc(f); c!='\n'; c=getc(f));
        }

        free(opd1);
        free(opd2);
        free(opd3);
        free(opd4);
        free(opd21);
        free(opd22);
        free(opd23);
        free(outpar);
}

void EDFS(TREE *pNode, char *pS, char *pM, int depth)
{
        TREE    *pTemp;
        char    pSTemp[200], pMTemp[400];

        if(depth > MAXDEPTH) return;
        pNode->clr = 1;

        while(1)
        {
                pNode = pNode->pSon;
                if(pNode==NULL) break;

                strcpy(pSTemp, pS);  //erase old string of state-nu
                sprintf(pSTemp, "%s %d", pSTemp, pNode->NodeNumber)
                strcpy(pMTemp, pM);
                if(strlen(pNode->Msg) != 0)
                  sprintf(pMTemp, "%s %s", pMTemp, pNode->Msg);

                pTemp = pRoot->find(pNode->NodeNumber);

                if(pTemp != NULL)
                    EDFS(pTemp, pSTemp, pMTemp, depth+1);
                else
                {
                    if(addMSS(pMTemp))
                    {
                        strcpy(MSSes[MSSptr], pMTemp);
                        MSSptr++;
                    }
                }
        }
}

int addMSS(char * strMSS)
{
    int i;

    for(i=0; i<MSSptr; i++)
      if(strcmp(MSSes[i], strMSS)==0) return 0;
    if(strstr(strMSS, "sequencefound")!=NULL)  return 1;

    return 0;
}

void findCCS()
{
    int i, j, k, iScnt, c, isFound;

    //Right Trim
    for(i=0; i<CCSptr; i++)
    {
      for(j=strlen(CCScand[i]); j>0; j--)
          if(CCScand[i][j-1]==' ') CCScand[i][j]='\0';
          else break;
    }

    for(i=0; i<CCSptr; i++)
    {
      iScnt=0;
      for(j=i+1; j<CCSptr; j++)
      {
        isFound=1;
        if(strcmp(CCScand[i]+4, CCScand[j]+4)!=0) isFound=0;

        if(isFound==1) iScnt++;

        if(iScnt>=3) { printf(" %s (%d)\n", CCScand[i]+4, iScnt);
                       break; }
      }
    }
}

void Dump()
{
        TREE    *pTemp, *pNode;
        char    pSTemp[200*sizeof(char)];

        for(pNode = pRoot; pNode != NULL; pNode = pNode->pSibling)
        {
            for(pTemp = pNode; pTemp != NULL; pTemp = pTemp->pSon)
                  printf(" %d<%s>", pTemp->NodeNumber, pTemp->Msg);
        }
}

void printMSS()
{
        // PRINT OUT MINIMUM SEPERATING SEQUENCES
        int     c, i, k;
        char    *strMSS, *strTemp;

        strTemp = (char*)malloc(200*sizeof(char));

        for(c=0; c<MSSptr; c++)
        {
           CCSptr=c;
           k=0;
           strMSS = MSSes[c];
           while(*strMSS==' ') strMSS++;
```

```
            // filter out unnecessary outputs
            strTemp = strtok(strMSS, " ");
            if(*strTemp != 's') continue;
            for(i=1; i<strlen(strTemp); i++)
                 if(!isdigit(*(strTemp+i))) break;
            if(i<strlen(strTemp)) continue;

            strMSS += strlen(strTemp)+1;
            strcpy(CCScand[CCSptr], strTemp);
            k=strlen(strTemp);
            CCScand[CCSptr][k]=' ';
            CCScand[CCSptr][k+1]=' ';
            CCScand[CCSptr][k+2]='\0'; k+=2;

            for(i=0; ; i++)
            {
                if(i==(i/2)*2)
                  while(*strMSS != ' ' && *strMSS != '\0')
                  {
                      printf("%c", *strMSS);
                      CCScand[CCSptr][k]=*strMSS;
                      strMSS++;
                      k++;
                  }
                else
                  while(*strMSS != ' ' && *strMSS != '\0') strMSS++


                if(*strMSS == '\0') { CCScand[CCSptr][k]='\0'; brea
                else
                {
                  while(*strMSS==' ') strMSS++;
                  printf(" ");
                  CCScand[CCSptr][k]=' ';
                  k++;
                }

                if(strcmp(strMSS, "sequencefound")==0) break;
            }
           printf("\n");
        }
}

main(int argc, char *argv[])
{
        TREE    *pTNode;
        char    *pStr, *pMStr;

        MSSptr = 0;
        pRoot = new TREE();
        pStr = (char*)malloc(100*sizeof(char));
        pMStr = (char*)malloc(200*sizeof(char));

        if(ConstructAdjList(argv[1])) return -1;
        if(argc >= 3) MAXDEPTH = atol(argv[2]);
        pTNode = pRoot->pSibling;
        //Dump();

        while(1)
        {
            for(;;)
            {
                if(pTNode == NULL) break;
                if(pTNode->clr == 0) break;
                pTNode = pTNode->pSibling;
            }

            if(pTNode == NULL) break;
            sprintf(pStr, "%d", pTNode->NodeNumber);
            strcpy(pMStr, "");
            EDFS(pTNode, pStr, pMStr, 1);
        }

        printf("\n\n\nMinimal Separating Sequences:\n");
        printMSS();
        printf("\n\n\nCCSes: \n");
        findCCS();
        free(pStr);
        free(pMStr);
}
```

Appendix D

Minimal Separating Sequences Found By Distinguishing System

```
Minimal Separating Sequences:
dt(un,false)
dt(un,true)
dt(zero,false)
dt(zero,true)
cr  dt(un,false)
cr  dt(un,true)
cr  dt(zero,false)
cr  dt(zero,true)
cr  cr  dt(un,false)
cr  cr  dt(un,true)
cr  cr  dt(zero,false)
cr  cr  dt(zero,true)
cr  iconresp  dt(un,false)
cr  iconresp  dt(un,true)
cr  iconresp  dt(zero,false)
cr  iconresp  dt(zero,true)
cr  idisreq  dt(un,false)
cr  idisreq  dt(un,true)
cr  idisreq  dt(zero,false)
cr  idisreq  dt(zero,true)
iconresp  dt(un,false)
iconresp  dt(un,true)
iconresp  dt(zero,false)
iconresp  dt(zero,true)
iconresp  cr  dt(un,false)
iconresp  cr  dt(un,true)
iconresp  cr  dt(zero,false)
iconresp  cr  dt(zero,true)
iconresp  iconresp  dt(un,false)
iconresp  iconresp  dt(un,true)
iconresp  iconresp  dt(zero,false)
iconresp  iconresp  dt(zero,true)
iconresp  idisreq  dt(un,false)
iconresp  idisreq  dt(un,true)
iconresp  idisreq  dt(zero,false)
iconresp  idisreq  dt(zero,true)
idisreq  dt(un,false)
idisreq  dt(un,true)
idisreq  dt(zero,false)
idisreq  dt(zero,true)
idisreq  cr  dt(un,false)
idisreq  cr  dt(un,true)
idisreq  cr  dt(zero,false)
idisreq  cr  dt(zero,true)
idisreq  iconresp  dt(un,false)
idisreq  iconresp  dt(un,true)
idisreq  iconresp  dt(zero,false)
idisreq  iconresp  dt(zero,true)
idisreq  idisreq  dt(un,false)
idisreq  idisreq  dt(un,true)
idisreq  idisreq  dt(zero,false)
idisreq  idisreq  dt(zero,true)
dt(un,false)
dt(un,true)
dt(zero,false)
dt(zero,true)
```

```
cr
iconresp
idisreq
dt(un,false)
dt(un,true)
dt(zero,false)
dt(zero,true)
cr
iconresp
idisreq
dt(un,false)
dt(un,true)
dt(zero,false)
dt(zero,true)
cr  dt(un,false)
cr  dt(un,true)
cr  dt(zero,false)
cr  dt(zero,true)
cr  cr  dt(un,false)
cr  cr  dt(un,true)
cr  cr  dt(zero,false)
cr  cr  dt(zero,true)
cr  iconresp  dt(un,false)
cr  iconresp  dt(un,true)
cr  iconresp  dt(zero,false)
cr  iconresp  dt(zero,true)
cr  idisreq  dt(un,false)
cr  idisreq  dt(un,true)
cr  idisreq  dt(zero,false)
cr  idisreq  dt(zero,true)
iconresp  dt(un,false)
iconresp  dt(un,true)
iconresp  dt(zero,false)
iconresp  dt(zero,true)
iconresp  cr  dt(un,false)
iconresp  cr  dt(un,true)
iconresp  cr  dt(zero,false)
iconresp  cr  dt(zero,true)
iconresp  iconresp  dt(un,false)
iconresp  iconresp  dt(un,true)
iconresp  iconresp  dt(zero,false)
iconresp  iconresp  dt(zero,true)
iconresp  idisreq  dt(un,false)
iconresp  idisreq  dt(un,true)
iconresp  idisreq  dt(zero,false)
iconresp  idisreq  dt(zero,true)
idisreq  dt(un,false)
idisreq  dt(un,true)
idisreq  dt(zero,false)
idisreq  dt(zero,true)
idisreq  cr  dt(un,false)
idisreq  cr  dt(un,true)
idisreq  cr  dt(zero,false)
idisreq  cr  dt(zero,true)
idisreq  iconresp  dt(un,false)
idisreq  iconresp  dt(un,true)
idisreq  iconresp  dt(zero,false)
idisreq  iconresp  dt(zero,true)
idisreq  idisreq  dt(un,false)
idisreq  idisreq  dt(un,true)
idisreq  idisreq  dt(zero,false)
idisreq  idisreq  dt(zero,true)


CCSes:
 dt(un,false)
 dt(un,true)
 dt(zero,false)
 dt(zero,true)
```