Automatic Detection of Software Failures with Hierarchical Supervisors

by

Tony Savor

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical Engineering

Waterloo, Ontario, Canada, 1997

0-612-22233-0

Acknowledgments

First and foremost, I would like to thank my supervisors, Prof. R.E. Seviora and Prof. P. Dasiewicz whose efforts and contributions to this work went well beyond the call of duty. Gratitude also goes to all members of my committee: Prof. J.M. Atlee, Prof. B.R. Preiss, Prof. A. Singh and Prof. M.A. Vouk for their guidance throughout the course of this work. In addition, appreciation goes to all members of the Software Engineering Group for their many contributions over the years. Finally, I would like to thank Claudia and the members of my family who stood behind me during this period.

Funding for this work was provided by Bell Canada, the Natural Sciences and Engineering Research Council of Canada, the Province of Ontario and the University of Waterloo.

Abstract

As the size and complexity of modern software systems grows, it becomes increasingly difficult to determine whether they operate as specified. Presently, the process is excessively dependent on human observation, limiting its scalability and accuracy. Accurate and reliable detection of software failures would aid in the management and improvement of software reliability. An automated approach to detection of software failures is needed.

This thesis addresses software supervision, an approach to specification-based, automated detection of software failures. The work is focused on real-time reactive systems specified in a formalism based on communicating finite state machines. The supervisor, a separate unit, observes the inputs and outputs of a target software system. It makes use of the target systems' requirements specification. Discrepancies between specified and observed behaviors are reported as failures by the supervisor.

Supervision involves a number of difficult issues. A prominent one is the handling of specification nondeterminism. Specification nondeterminism permits the target system to generate several legal output behavioral alternatives for a single input behavior. The supervisor must be able to consider all behavioral alternatives so that unwarranted failure reports are not generated. In some cases, the exhaustive consideration of all behavioral alternatives results in an excessive supervisor time and space cost.

This thesis presents a novel approach to supervision, called hierarchal supervision, that reduces the time and space cost of supervising systems whose specifications contain large amounts of nondeterminism. In a hierarchal supervisor, failure detection is carried out at two levels of abstraction: the path detection level and the base level. The path detection level determines the path or trajectory through the specification that corresponds with observed target system behavior. Effectively, at the path detection level, the behavioral alternative chosen by the target system is identified. At the base level, a detailed check of observed behavior along the path identified is made.

This thesis presents the underlying concepts of hierarchal supervision, the architecture of a hierarchal supervisor, the derivation of the supervisor model from the requirements specification, the definition of the interpreters for both the path detection and base supervisor levels and describes the derivation of the time and space complexities for both. The major research contributions of the thesis include splitting of supervision into two sub-problems (path detection and detailed behavior checking), making use of both target system input and output signals to track target system behavior, discussion of tradeoffs between the latency of failure detection vs the computational cost of supervision, development of an approach to prune behavioral alternatives from consideration and development of a base supervisor aimed at detailed behavior checking.

To evaluate hierarchical supervision, a demonstration supervisor was implemented. It supervised the control program of a small telephone exchange. Two key aspects, failure detection and time/space complexity, were evaluated.

The failure detection evaluation included both optimistic and pessimistic reporting. Pessimistic reporting refers to unwarranted generation of failure reports, while optimistic refers to not generating warranted failure reports. Experimental observations revealed that all failures were reported and no failures were missed. The time and space cost was evaluated by measuring the number of behavioral alternatives considered by the supervisor, which is indicative of its time and space cost. Experimental measurements showed improvements of over two orders of magnitude over the direct single-layer approach.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Failure Detection

This thesis addresses automatic detection of software failures. It is well known that state-of-the-art software development processes yield imperfect software. Thus it is common for large systems such as telecommunication switches, avionics flight systems etc. to contain several thousands of software faults. Automatic failure detection is the first step to dealing with failures arising from software faults.

1.2 Software Supervision

Software supervision is an approach aimed at automatically detecting externally observable software failures. A software supervisor monitors the inputs and outputs of the target system (figure 1.1) and makes use of the target system's requirements specification.

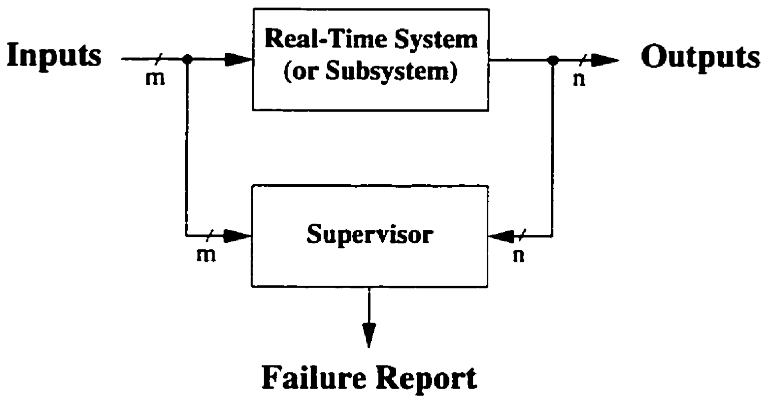


Figure 1.1: Software Supervisor

Internally, the supervisor generates a set of *expected behaviors* from the requirements specification and target system input and/or output signals. The expected behaviors are compared with actually observed behaviors. A failure is reported if a match between the two cannot be made.

The supervisor may be attached to either the entire system or a sub-system, provided that the inputs and outputs of the sub-system are observable. In the latter case, a supervisor could be used to detect *errors* before they manifest themselves as externally observable failures.

A number of challenges exist in the development of a software supervisor. One major challenge is dealing with *specification non-determinism.* Specification non-determinism permits several legitimate output behaviors for a single input behavior. A supervisor that uses a specification containing non-determinism must be able to consider all legitimate behavioral alternatives so that false failure reports are not generated.

For some specifications, the number of legitimate behavioral alternatives can be large. Explicit consideration of each alternative can result in a very large supervisor time and/or space complexity.

1.3 Why Supervision

There are three principal categories of application areas for supervision: software development, on-line supervision and software reliability instrumentation. This section outlines several uses of a supervisor within each application area.

A. Software Development: During software development, a supervisor may be used to report software failures prior to release. Two specific application areas are:

1. **Fault-localization tool.** Large software systems can be partitioned into several sub-systems. If a supervisor is attached to each sub-system, a fault may be automatically localized to a sub-system. In this case, supervision has the potential of reducing costs associated with software debugging.

2. **Test tool.** Systems exhibiting non-determinism are difficult to test due to a number of possible outcomes for a given test case. As a result, testing, in most cases, is restricted to specific cases with few, known outcomes.

   A supervisor is able to report a relatively complete set of failures. It would serve to improve the effectiveness of testing and indirectly improve the reliability of developed software.

B. On-Line Supervision: The presence of faults in software systems during field-operation makes supervision an attractive approach for detecting failures. On-line supervision presents a number of advantages including:

- Early reporting of failures allows a company to repair underlying faults before more serious consequences occur.

- Minor failures are, in some cases, indicative of more serious future problems. Accurate reporting of failures often gives early warning of potential future catastrophes.
- The supervisor maintains a more global perspective of the system than any individual user. It is thus able to report failures not visible to individual users.
- The supervisor is able to provide more accurate and detailed failure reports than non-technically oriented users.

C. **Software Reliability Instrumentation:** A major impediment to the advancement of the software reliability engineering discipline are the difficulties associated with collection of software failure data. At present, the process is excessively dependent on human intervention both for the detection of failures and collection of relevant descriptors. The software supervisor may be used to automate this process.

1.4 Objectives

The primary objective of this work is the research of an efficient approach to automatic detection of software failures in the presence of specification non-determinism. The intended application of the failure detection unit is real-time reactive telecommunications software.

1.5 Summary of Research Contributions

- Partitioning supervision into two subproblems: target system tracking and detailed behavior checking.
- Definition of a framework to track the target system operation. The tracking unit consists of a model and an interpreter.
  - Formalization of the semantics of the tracking-unit model.
  - Research of a derivation procedure for the tracking unit model.
  - Definition of algorithms for a suitable tracking system model interpreter.
  - Development of a prototype implementation of the tracking unit interpreter.
- Definition of a framework for a detailed behavior checking. A detailed behavior checking unit consists of a model and an interpreter.
  - Formalization of the semantics of the detailed behavior checking unit.
  - Development of algorithms for a suitable interpreter of the detailed behavior checking unit.
  - Development of a prototype implementation of the detailed behavior checking unit.
- Computational complexity assessment of the proposed approach.

1.6 Organization of Thesis

This thesis is organized as follows: Chapter 2 outlines the major issues related to automated failure detection and overviews existing approaches.

Chapter 3 presents an overview of *hierarchical supervision*. A hierarchical supervisor consists of two layers: (1) the tracking layer (called the path-detection layer) and (2) the detailed behavior checking layer (called the base supervisor layer). Each layer makes use of a unique target system model and interpreter. Target system models are derived from the target system requirements specification.

Chapter 4 describes the transformation of the model used by the path detection layer. The transformation accepts as input the target system's requirements specification and generates a suitable model to be used by the path detection layer. Chapter 5 describes an interpreter for the aforementioned model.

Chapter 6 presents the model transformation and interpreter for the base supervisor layer. Evaluations of the approach based on a prototype supervisor and a small telephone exchange that served as a target system are presented in chapter 7. Conclusions are drawn in chapter 8.

Chapter 2

Issues & Related Work

This chapter outlines six major issues that arise in software supervision. Existing work that may be used for automatic detection of software failures is described next. The chapter concludes with an overview of the focus of this thesis in light of the issues and existing work.

2.1 Definition of Correct Behavior

The objective of supervision is to detect failures in the operation of a target system. The supervisor requires a definition of legitimate target system behavior. The definition is required to be complete and expressed using a formal notation.

One possibility is that the supervisor uses the target software system's requirements specification, typically developed as part of the software life cycle [46]. The requirements specification defines the externally observable behavior of the software system. A multitude of formal specification languages exist with formally defined semantics to minimize semantic ambiguities.

This work is focused on communicating finite state machine (CFSM) based formalisms. Many internationally standardized formalisms are based on a CFSM model. Examples include the Specification and Description Language (SDL) [58], Estelle [23], and Lotos [22].

2.1.1 Target System Response Time

Physical systems are typically specified as having finite response times. Thus an event, E will be serviced by the target system after R units of time.

For actual systems, R may be different for different events. Furthermore, for a single event, R may vary depending on several factors such as the target system load and the availability of resources. The exact response time may be impossible to determine analytically.

An approximation of the individual event response times can be made by considering the best and worst-case response times. The actual response time will fall within this interval. T^r_{min} is defined as the best case response-time of any event under any specified condition of the target system. Similarly, T^r_{max} is defined as the worst case response time. For the remainder of the thesis, each event will be considered to have a response time that falls within the interval $[T^r_{min}, T^r_{max}]$.

This thesis considers the case where the requirements specification consists of two components. The behavioral specification appears in a CFSM-based formalism. The behavioral specification is supplemented by a declarative specification of best and worst-case response times.

2.2 Specification Non-Determinism

Before non-determinism is defined, a definition of determinism is presented first. The definition originally appeared in the encyclopedia of philosophy [18].

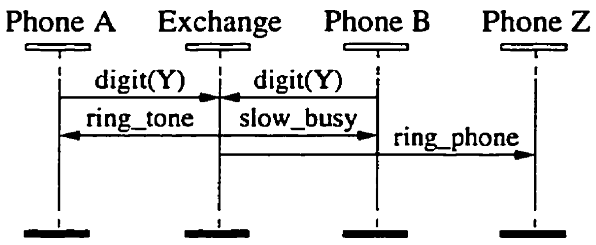
> Determinism is the general philosophical thesis which states that for everything that ever happens there are conditions such that, given them, nothing else could happen. (...) an event might be said to be determined in this sense if there is some other event or condition or group of them, sometimes called its cause, that is a sufficient condition for its occurrence, the sufficiency residing in the effects following the cause in accordance with one or more laws of nature

From this definition, non-determinism may be defined as the theory or doctrine that for each cause, there may be two or more legitimate effects.

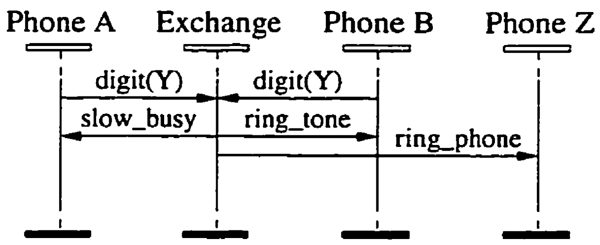
Non-determinism is an important part of many specification formalisms. It allows the specification writer to omit portions of the specification that are not relevant. This reduces the specification effort and gives the software designer more design freedom to choose the behavioral alternative (or alternatives) that would result in a less costly or otherwise desirable implementation.

Specifications having non-determinism allow systems to exhibit non-deterministic behavior during field operation. Consider a telephone exchange and the scenario where two parties, A and B simultaneously attempt to call a third party, Z. The exchange will typically exhibit non-deterministic behavior in that either A or B can connect to Z. The two behavioral alternatives arising are shown in figure 2.1.

A software supervisor must be able to consider all behavioral alternatives arising out of the non-determinism in the requirements specification. A supervisor that is not able to consider all behavioral alternatives may generate erroneous failure reports. Specification non-determinism is one of the major challenges of supervision



(a)



(b)

Figure 2.1: Non-Deterministic Behaviors

as the number of behavioral alternatives required to be considered by the supervisor may be large resulting in a large supervisor time and space complexity [47].

Specification non-determinism may refer to several categories of non-determinism. Descriptions of many of these can be found in [44]. The principal ones dealt with here are non-deterministically delayed communication paths and the non-determinism associated with the precise time of a local clock. For our purposes, the latter will refer to the difference between the values of the supervisor clock and the target system clock.

2.2.1 An Execution Path Interpretation of Non-Determinism

Many specification formalisms support different types of non-determinism. A common framework can be used to represent most types of non-determinism. The framework shall be referred to as the *execution path* (EP) interpretation.

An EP is defined as a series of state transitions through a finite state machine. Essentially, non-determinism permits two or more legitimate EPs for a possibly empty set of stimuli directed to a CFSM-based specification.

As an example, consider the single-FSM specification in figure 2.2a. On the arrival of stimulus a, either path $S_0 \rightarrow S_1$ or $S_0 \rightarrow S_2$ could be taken. The choice of EPs is non-deterministic despite both paths producing an identical observable output, X.

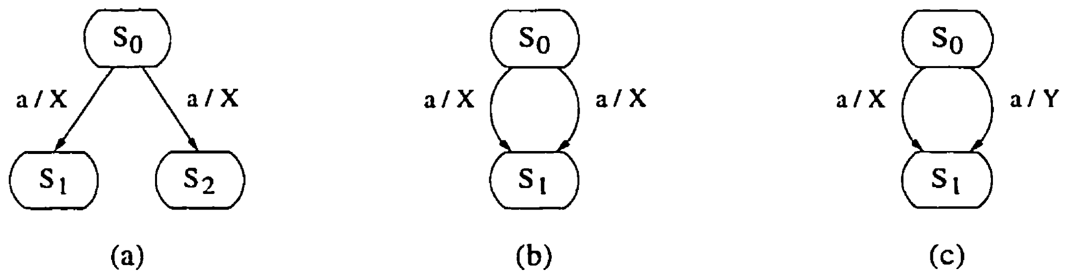


Figure 2.2: Example Finite State Machines

2.2.2 Categories of Non-Determinism

Non-determinism refers to choices in the EP. Specification non-determinism may be categorized into *don't care non-determinism* and *don't know non-determinism.* In this section, an informal definition of the two types of non-determinism is presented. A formal definition will appear later.

Don't care nondeterminism refers to two or more alternate EPs that if followed for a finite number of state transitions will leave the system in an *identical* global state. For communicating extended finite machine (CEFSM)-based specifications, global state refers to the collective state of all FSMs including the contents of communication channels and input ports.

A trivial example of don't care non-determinism is illustrated in figure 2.2b. Assume that the FSM is initially in state S_0 and stimulus a is consumed by the FSM. Regardless of the EP chosen, the FSM will output signal X and terminate in state, S_1.

Don't know non-determinism refers to or more alternate EPs that if traced will leave the system in a *different* global state. FSMs containing examples of don't know non-determinism are illustrated in figures 2.2a and 2.2c. The two paths in figure 2.2a leave the FSM in two different symbolic states, while the paths in figure 2.2c output different signals.

From the perspective of a software supervisor, all behavioral alternatives must be considered so that erroneous failure reports are not generated, as described in section 2.2. If the don't care non-determinism could be separated from the don't know non-determinism, the supervisor would only have to consider don't know non-determinism. This would have the desirable effect of reducing the time and/or space complexity of the supervisor.

2.3 Supervisor Signal Processing Latency

Signals to and from the target system are directed to the supervisor. Signals may be processed, by the supervisor, an arbitrary time after their occurrence. Two general categories of supervisors are *in-time* and *out-of-time*. The supervisors differ principally in the time at which signals are processed by the supervisor. In other words, the relation between the clocks of the supervisor and target system. A loosely bound definition of in and out-of-time supervision is presented below. This definition will be refined later as more issues are presented.

Consider an event, E, generated by the environment to be processed by the target system. Assume that E was generated at time T. A supervisor will process E at some time, $T + \Delta$. An in-time supervisor must be able to process, event E such that $\Delta = 0$ while an out-of-time supervisor must be able to process event E such that $\Delta > 0$.

The time at which events in a supervisor are processed is dependent on the response time of the target system. Consider two events, E_1 and E_2, representing requests for service (e.g. two telephones going offhook). E_1 and E_2 are generated at times, T_1 and T_2 respectively. If $|T_1 - T_2| < T^r_{max}$, then the order in which the events are serviced by a non-deterministically specified system may be arbitrary. For example, if telephone A goes offhook before telephone B, it may be possible (and legitimate) for B to receive dialtone before A.

In general, a violation of causality may result if events are processed as they are received by a supervisor. From the previous example, if event E_1 is processed before E_2 arrives (figures 2.3a and 2.3b). On the arrival of E_2, the supervisor determines that the order in which the events were processed does not correspond with the order chosen by the target system (figure 2.3c). The supervisor must revert to a previous state and reprocess the events in order $E_2 - E_1$ (figure 2.3d).

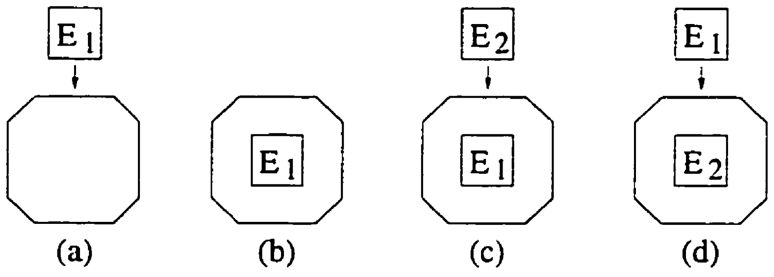


Figure 2.3: Causality Violation in Event Processing

Based on the issues in event processing latency, more precise definitions of in-time and out-of-time supervision follow.

2.3.1 In-time Supervision

An in-time supervisor is defined as one where events, generated at time T, are processed on the interval, $[T, T + T^{r}_{max}]$.

As outlined in section 2.3, causality violations may occur in an in-time supervisor. This category of supervisors must make provision for un-consuming consumed signals to un-do causality violations. Two approaches have been studied thus far. The signal-in-transit approach [25] pre-creates an explicit behavioral alternative for each possible signal that may arrive. The rollback-and-recovery approach [56] moves the global state of the supervisor back and re-orders the processing of events as required.

The principal advantage of in-time supervision is that failures are reported within T^{r}_{max} of their occurrence. The disadvantage is that the supervisor must be able to keep up with the target system (i.e. the supervisor cannot lag the target system by more than T^{r}_{max} units of time). In most cases, the supervisor is more computationally intensive than the target system due to the need to consider all behavioral alternatives. For systems with large amounts of non-determinism, the computational complexity of the in-time approach has been found to be a severe shortcoming [47].

2.3.2 Out-of-time Supervision

An out-of-time supervisor is defined as one where events generated at time, T are processed on the interval, $[T + T^{r}_{max}, T + \infty]$.

In an out-of-time supervisor, before an event is processed, the supervisor waits at least T^{r}_{max} units of time. The supervisor can thus guarantee that no further events will be generated that may precede the current one. Thus the out-of-time supervisor does not need a mechanism to un-do causality violations like its in-time counterpart.

The principal advantage of out-of-time supervision is that peaks in processing requirements can be amortized over an arbitrary amount of time. Thus the out-of-time supervisor requires a CPU that can process the *average* computational requirements of the target system rather than the *peak* as required by the in-time one. The disadvantage of the approach is the latency of failure reporting.

2.4 Tradeoffs Between Accuracy and Computational Cost

Specification non-determinism may result in large supervisor computational complexities as mentioned in section 2.2. This is currently one of the major impediments to the use of a supervisor. One possible approach of dealing with specification non-determinism is to use partial supervisor models [47]. A partial model would reduce the computational complexity of supervision at an expense of reduced failure detection capability.

Partial models may be derived from the requirements specification. There are two categories of approaches to devising partial models: pessimistic and optimistic. A partial supervisor model may be derived using a combination of the two approaches.

Pessimistic models can cause the supervisor to report failures while the target system is operating correctly. The failures reported by a pessimistic model are a superset of the actual set of failures. Pessimistic models are derived by eliminating alternative EPs representing don't know non-determinism from the requirements specification [47].

Optimistic models can cause the supervisor to miss reporting some failures. Failures reported by an optimistic model are a subset of the actual set of failures. Optimistic models are derived by eliminating m EPs representing don't know non-determinism from the specification and replacing them with n new EPs such that $m > n$ [51].

The effects of reduced model supervision have been studied in [45, 47]. It was determined that the savings are proportional to the number of encountered behavioral alternatives. As system loads get larger, more non-determinism was encountered and more savings in computational complexity were realized. For one particular experiment, reductions in computational complexity of several orders of magnitude were observed with approximately three quarters of failures reported [47].

2.5 Attachment of a Supervisor to a Target System

To minimize the interference with the target system software, a supervisor typically executes on a separate hardware platform. There are several ways a supervisor can be attached to observe the input and output signals of a target system. This work is targeted towards systems with a large number of input and/or output connections such as communication controllers, telephone exchanges etc. The physical connection of the supervisor to each input/output wire of a large system is practically infeasible. Two commonly used approaches will be described here, namely: (1) tapping of a data link and (2) polling of controlled hardware interface memory.

Both are shown in figure 2.4.

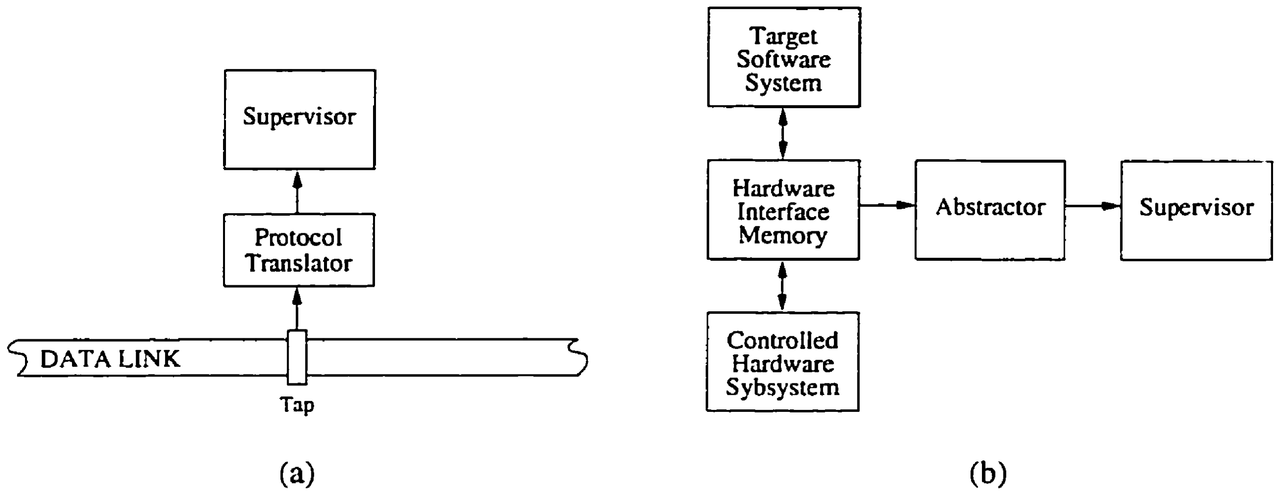


Figure 2.4: Supervisor Connectivity Patterns

2.5.1 Tapping of a Data Link

Tapping of a data link refers to snooping traffic traveling across a communication channel. Data is monitored in read-only mode. A protocol translator converts physical-layer signals to events that can be processed by the supervisor.

The difficulty with this approach is the multiple interpretations of a lack of information by the protocol translator. The absence of information is typically handled by timeouts in many protocols. The protocol translator must deal with the absence of an event (for example) in the same way as the target system. The precise time that the timeout occurs is non-deterministic due to the lack of knowledge in the supervisor about the local clock of the target system (as discussed in section 2.2).

In such cases, two behavioral alternatives need to be considered by the supervisor: (1) that the timeout has expired before the event is received and by the target system and (2) that the timeout expires after the event is received.

2.5.2 Polling of Controlled Hardware Interface Memory

A software system's input and output signals can be identified by polling the controlled hardware interface memory. An *abstractor* (figure 2.4) is used to convert bit changes into signals recognizable by the supervisor. Current hardware design-for-testability trends such as boundary-scan [42] may facilitate polling hardware interface memories.

Several issues arise when poling the hardware interface memory. Three common ones are described here. First, signals of short duration may be missed. Second, the order signals are reported may be permuted by the abstractor and finally, the scanning of some signals may be dependent on the correct target system operation. A brief overview of each of the issues follows.

Short Duration Signals

An abstractor samples the hardware interface memory at a fixed frequency, f_s. Consider a signal E, generated by the target system with duration, T_E such that T_E is less than the sampling period (i.e. $T_E < \frac{1}{f_s}$). If E is generated between sampling points, it will be missed by the abstractor.

Consider the example in figure 2.5a. Signal, E is generated between sampling points 1 and 2. The abstractor will miss reporting the occurrence of signal E. The missed event will be reported by the supervisor as an illegitimate failure of the target system.

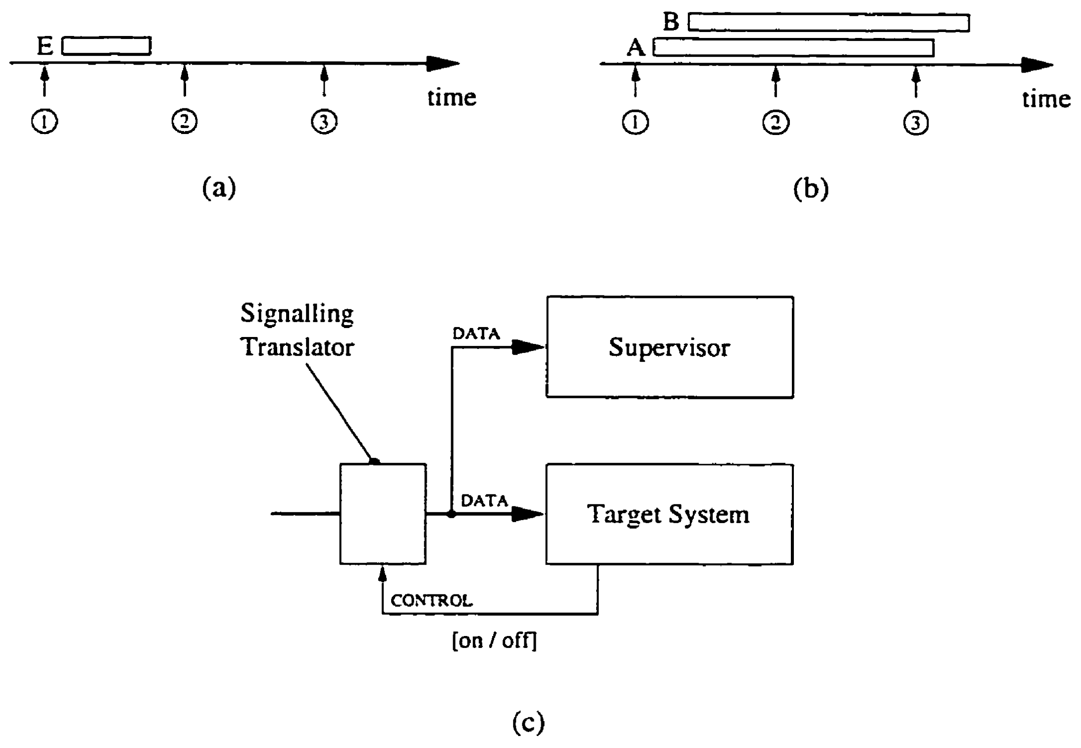


Figure 2.5: Sampling of the Hardware Interface Memory

Reversal of Signal Order

If two or more signals are generated between sampling intervals, the abstractor will not be able to report the actual signal generation order. Rather, the order reported will be based on some internal abstractor scanning order.

As an example consider the two events, A and B occurring between sampling points 1 and 2 as shown in figure 2.5b. Both signals, A and B will be detected by the abstractor at sampling point 2 and the actual order of occurrence is not resolvable by the abstractor.

For some specifications, order of signal generation is critical. If the abstractor reports signals out of order, the supervisor will report an erroneous failure of the target system.

Dependence on Correct Target System Operation

The supervisor relies on the correct operation of the target system for some signals to be reported by the abstractor. Consider the case shown in figure 2.5c. A common signaling translator is used by both the supervisor and target system. Furthermore assume that the signaling translator is turned off and on as needed by the target system software. This could be representative of a power-critical application such as a battery-operated device, or the case where the signaling translator is a shared resource, allocated/deallocated as needed.

The difficulty arises in that the requirements specification only specifies the externally observable behavior. Switching the signaling translator on and off is typically not specified at the requirements specification since it is not an externally observable event.

If a software fault exists that omits turning on the signaling translator, the events will be suppressed by the translator and neither the target system nor supervisor will receive them. This type failure is not detectable by a supervisor as the supervisor relies on the correct operation of the target system for the signal to be generated.

2.6 Continuation of Supervision After Detection of a Failure

A requirements specification typically does not specify the behavior of a target system after the occurrence of a failure. From the requirements specification perspective, a failure causes the target system to traverse a state transition that does not correspond with any transition in the requirements specification. This may lead the target system into a state that does not correspond with any state in the requirements specification.

If the supervisor remains attached to a system after a failure occurs with the supervisor state different from the target system state, the supervisor would expect one behavior and the target system would generate another. The result would be a shower of failure reports generated by the supervisor.

Most systems exhibit some fault tolerance capability. For minor failures a system may be able to recover its operation after a period of time, t_f (figure 2.6). Session oriented systems typically fall into this category. For example, if a failure is observed during a telephone call in North America, a natural reaction would be to place the telephone onhook and to re-attempt the call, effectively re-setting the state of the local phone.

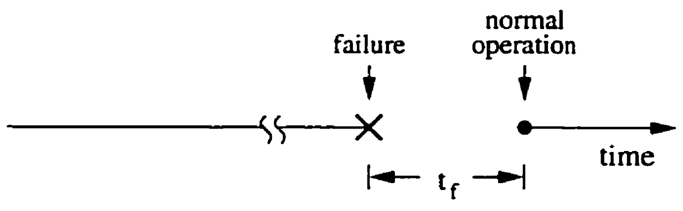


Figure 2.6: Operation of a System After Occurrence of a Failure

2.6.1 Resynchronization

The post-failure state of the target system is not known by the supervisor, but is needed to prevent generation of spurious failure reports. The post-failure state of the target system may be determined once it resumes normal operation. Once the state of the target system is known, supervision may resume.

A *resynchronization mechanism* is needed to determine the post-failure state of a target system. The mechanism accepts as input both target system inputs and outputs, just like the supervisor. It generates a state corresponding with the current state of the target system based on the requirements specification. The problem is complicated because *distinguishing signal sequences* must be determined for all CEFSMs including internal ones that do not communicate directly with the environment. This result is a very large possible search space [2, 16, 15].

In the context of supervision, resynchronization was studied in [30, 35]. The central research issue in both cases was coping with the large number of possible states that the target system could be in. Both used assumptions to limit the number of possible states, for example [30] made the assumption that the post-failure state was closest to the pre-failure state while [35] proposed resynchronization based on the pre-failure state and target system fault models.

2.7 Related Work

Previous work on monitoring software systems for failures can be subdivided into two broad categories: *intrusive* and *non-intrusive*. Intrusive approaches require modifications to the target system software while non-intrusive approaches do not. Existing work on several intrusive and non-intrusive approaches to software monitoring is described in the following section.

2.7.1 Intrusive

Software Audits

Software data errors are detected and possibly corrected by means of *audit programs* [1, 13, 41, 43] before they manifest themselves as failures. Audit programs consist of additional software which has access to the main program's data structures. An audit executes at a lower priority than the main program and periodically checks data structures for errors.

Audits principally detect three types of errors [41]: (1) *direct comparison errors*, comparison of data structures with a duplicate, (2) *comparison by association errors*, detection of failures with the aid of data structure redundancy such as a doubly-linked list and (3) *format comparison errors*, common sense checking of data such as bounds checking.

The main advantage of audits is that they are able to detect software errors before the errors manifest themselves as failures. However, audits detect only a limited set of errors. In addition, audits themselves may contain faults, potentially reducing the overall reliability of the software.

Watchdog Timers

The watchdog timer is an approach for detecting severe system failures [38]. The approach requires that the target system software be instrumented with code to generate sanity pulses within an interval of time, T. Generally, generation of sanity pulses surrounds code such as procedure calls, resource requests or loops with known worst case execution times. In the event that some portion of code does not terminate before its maximum execution time, a sanity pulse is not generated within the required time.

An external unit or watchdog timer, monitors the sanity pulses. The timer may be implemented in hardware and/or software [29]. Hardware implementations are able to report a broader range of failures than purely software approaches. If a pulse is not received within T units of time, the unit reports that a failure of the software has occurred.

The advantage of watchdog timers is that they are simple and easily implemented. The disadvantage is the limited set of failures that can be detected.

Run-Time Result-Checking

Run-time result-checking refers to a collection of approaches to check the correctness of results produced by program modules [8, 9, 12, 50]. Correctness checks are performed on the outputs of modules/programs. As an example, if a procedure is to compute a function, $y = f(x)$, a checker could make use of the inverse function to re-compute the actual inputs, $x = f^{-1}(y)$.

There are several difficulties with this approach. Development of a checking routine may be more complex than the actual routine itself. Result-checking software that executes on the same processor as the target system may degrade the overall system performance. In addition, the checking software may itself contain faults, reducing the overall reliability of the system. Sankar and Mandel [50] have developed a distributed monitoring approach where the monitor resides on a separate processor that alleviates these problems to some degree.

2.7.2 Non-Intrusive

N-Version Programming

N-version programming (NVP) refers to an approach for failure detection/fault tolerance [3]. From a single requirements specification, N separate designs and implementations are produced by N isolated teams of developers.

The N-versions of software are all executed concurrently. The outputs of all N copies are fed into a *voting algorithm* that compares outputs. If all outputs are not identical, a failure may be reported. Fault-tolerance is achieved by having the voting algorithm choose a non-failed output and use it as the actual output of the system. A majority-wins algorithm is one such common voting scheme.

The principal difficulty with NVP is its cost. N-versions of the software are required. Studies have shown that the N versions of software may contain identical faults despite being developed by isolated teams [21, 32]. Additionally, nondeterminism poses difficulty as each of the N versions may have different outputs that are all legitimate. Recent research has focused on ways to reduce implementation non-determinism [44]. However this may have the undesirable effect of increasing development costs.

External Assertion Checking

External assertion checking refers to an approach that checks certain properties of outputs generated by a specific target system. Two such systems, *Elektra* [31, 53] and HMON [17] are described here.

Elektra is an electronic railway control system. It consists of two primary components, *the logic processor* and the *safety bag*. The logic processor is the target system. The safety bag checks and possibly rejects outputs produced by the logic processor. The safety bag consists of a real-time rule-based expert system that encodes various safety rules stated by the railway authority.

HMON is a distributed real-time monitoring and debugging environment. It is able to monitor of several event types including system calls, context switches, interrupts and shared variables. *HMON* attaches itself to the target system software through shared libraries and a modified kernel. It allows the user to specify attributes about each of the events. Discrepancies between the specified event attributes and actually observed events are reported as failures.

Both approaches monitor properties of the target system. As a result, they are only able to reported a limited set of failures.

The Observer

The observer [4, 5, 14] is an approach for formal on-line validation of distributed systems. It is very similar to a software supervisor. The observer monitors the inputs and outputs of the target system and makes use of a formal model of the target system, derived from the requirements specification. Discrepancies between observed behaviors and behaviors represented by its internal model are reported as failures.

The observer was applied to the monitoring of distributed systems. The major difference between the observer and supervision is that the work reported on the observer does not address the issue of specification non-determinism.

Software Oracles

An oracle is an external source of information about a program. Common examples of oracles include proof axioms, another program or a formal specification [10, 40, 49]. Approaches to the automated development of oracles from specifications have been described.

A principal use of oracles has been in software testing. Oracles categorize test cases as either legitimate or illegitimate. As a result, they are typically only able categorize the behaviors represented by the test cases due to their limited model of the target system.

2.8 Research Focus

Category of Systems: This thesis addresses supervision of discrete, real-time, reactive systems that service humans. The case where the system specifications appear in a communicating extended finite state machine based formalism is considered. Such systems typically have a simple interface and as a consequence a simple specification.

Categories of Failures: The detection and reporting of behavioral and performance failures is addressed.

Behavioral failures are defined as spurious, incorrect or missing events that are generated and/or not-generated by the target system. Performance failures are defined as violations of the temporal requirements of a specification [11]. The category of performance failures considered are violations of worst case response time, T^r_{max}.

Definition of Correct Behavior: This thesis addresses supervision of CEFSM-based requirements specifications. For the sake of concreteness, discussion is aimed at the Specification and Description Language (SDL) [58]. SDL is standardized by the International Telecommunications Union (ITU) and used internationally within the telecommunications industry. The reader is referred to [7] for an introduction to the language.

Treatment of supervision with SDL-specifications is focused to a subset of SDL-88. The subset is sufficient for many applications such as telecommunications call processing software. An outline of addressed constructs follows.

Structural Constructs: system, block, process

Communication Constructs: signal, signal route, channel

SDL Process Constructs: decision, signal input, signal output, save, task, start, state, stop, any, none

Specification Non-Determinism: This work addresses non-determinism associated with multiple event consumption orders. Three types of SDL non-determinism that fall into this category are: non-deterministic channel delay, spontaneous transitions and non-deterministic decisions. The latter two types of non-determinism may be modeled with non-deterministic channel delay.

Supervisor Signal Processing Latency: This thesis focuses on out-of-time supervision. Events generated by the target system's environment or by the target system itself may be processed by the supervisor an arbitrary time after their generation.

Tradeoffs Between Accuracy and Computational Cost: The case where a complete set of failures is required is considered. Thus supervision with a full model of the target system is treated in this work.

Observability of Target System Inputs: This work assumes complete observability of all target system input and output events.

Continuation of Supervision After Detection of a Failure: Addressed is supervision of correct behavior from the point where the target system is initialized to the point where a failure is detected.

Chapter 3

Hierarchical Software Supervision

This chapter gives an overview of *hierarchical software supervision*, an approach to supervision aimed at dealing with specification non-determinism.

The chapter beings with some definitions that will be used throughout the remainder of the thesis. The internal organization of a hierarchical supervisor is described next followed by a discussion of each function unit within the supervisor. The chapter concludes with a description of the operation of the supervisor.

3.1 Definitions

Definition 3.1.1 (Process State) *For an SDL process, P_i, the process state is defined as a 3-tuple, $\psi =< \sigma, V, Q >$ where:*

- *σ represents the current symbolic state of P_i;*
- *V is the set of all variables and associated assignments;*

- *Q is the sequence representing the contents of P_i's input queue.*

Definition 3.1.2 (Global State) *For an SDL specification consisting of processes, $P_1, P_2, \ldots, P_n$, the global state of the specification, Σ is defined as a tuple of the all n process states, $\Sigma = < \psi_1, \psi_2, \ldots, \psi_n >$.*

Note that the definition of global state assumes that all communication channels in the specification are empty. Thus it may be considered a *quiescent* global state. This definition simplifies the discussion as the additional state space introduced by channels is omitted.

3.2 Internal Organization of a Non-Hierarchical Supervisor

The following description gives a conceptual overview to the components and operation of a software supervisor. Conceptually, a software supervisor consists of five fundamental components: the supervisor model, interpreter, expected behavior buffer, observed behavior buffer and a matcher. One possible variant of a software supervisor, where inputs are used to generate expected behaviors or an input-driven supervisor is shown in figure 3.1.

The supervisor model captures the legitimate behaviors of the target system. As discussed in section 2.8, the case where the supervisor model is specified in SDL is considered. The interpreter interprets the supervisor model. Behaviors expected to be generated by the target system (expected behaviors) are buffered in the expected behavior buffer. Correspondingly, observed behaviors are buffered in the observed behavior buffer. This alleviates the need for both behaviors to be

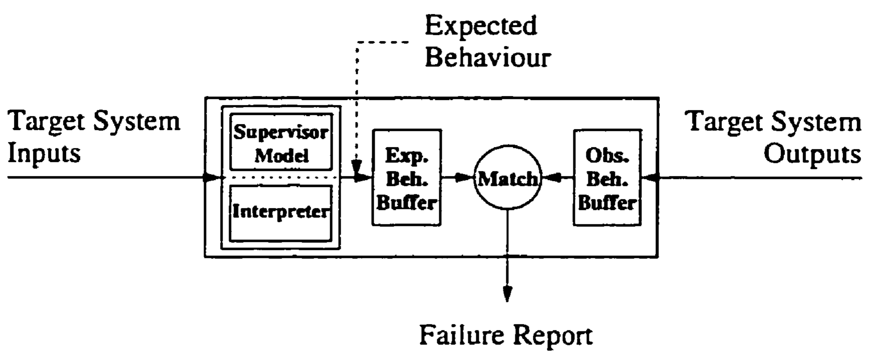


Figure 3.1: Anatomy of a Software Supervisor

generated at precisely the same time. A matcher compares the contents of the two buffers and reports a failure if a match cannot be made.

3.2.1 Approaches to Dealing with Specification Non-determinism

Specification non-determinism permits more than one legitimate expected behavior for a given observed behavior. If the behavioral alternatives are visualized as alternate EPs through the supervisor model, as outlined in section 2.2.1, the supervisor must be able to consider all alternate EPs. Two approaches have been developed.

The *belief method* [26] explores all legitimate EPs in a breadth-first manner. A separate thread of execution or *belief* is created for each encountered EP. A belief represents one global state of the supervisor model and the contents of the expected/observed behavior buffers. Beliefs are terminated as their externally observable behavior is invalidated by the actually observed target system behavior.

The belief method is a conceptually elegant approach for dealing with behavioral alternatives. However, its most serious shortcoming is its worst case time/space complexity. Consider the case where N signals are queued for consumption whose order cannot be determined. In this scenario, the worst case computational complexity of the supervisor is given by (3.1) [26].

$$\sum_{i=0}^{N} \frac{N!}{i!} \tag{3.1}$$

The *optimistic path prediction and rollback* (OPPR) approach [56] was developed to overcome the large time and space requirements of the belief method. OPPR explores legitimate EPs in a depth-first fashion, according to a heuristic derived from the target system's operational profile.

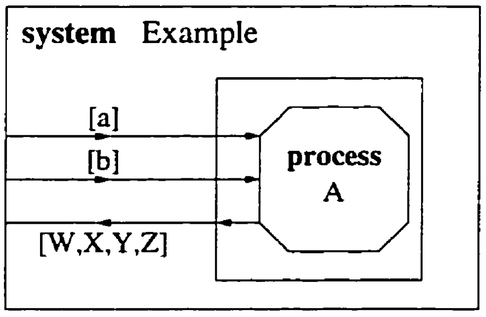
Results indicate that the average case complexity of the OPPR approach is significantly better than the belief based approach [55, 56]. However, upon occurrence of a failure, the OPPR must explore all behavioral alternatives, resulting in a worst-case complexity similar to that of the belief-based method.

3.3 Tracking Target System Operation

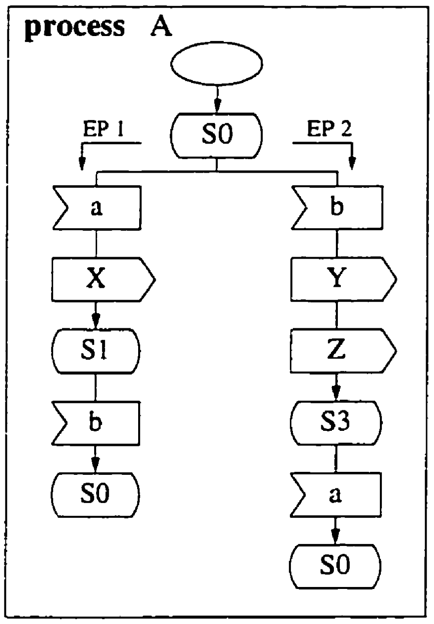
The belief method considers all EPs concurrently while OPPR considers a heuristically ordered sequence of EPs. In many cases, however the actual EP chosen by the target system may be inferred dynamically from the observable signals to and from the target system.

As an example, consider the SDL specification in figure 3.2. Assume that signals a and b are generated by the environment within a short duration, ϵ of each other[1]. Due to the non-deterministic SDL channel delay, process A could consume the signals in order: $a - b$ or $b - a$. Specification non-determinism thus permits either path 1 or path 2 to be legitimately traversed.

[1]The actual bounds for ϵ will be discussed later.



(a)



(b)

Figure 3.2: Example SDL Specification

By having the supervisor watch for key signals (either target system inputs or target system outputs), the path chosen by the target system could be inferred. For the specification in figure 3.2, a supervisor could infer that path 1/path 2 was followed if signal, X/Y was generated by the target system. The reader should note that signals X and Z would have been equally effective in detecting the two state transitions.

3.3.1 The Tracking Model

In general, both target system input and output signals may be used to track target system operation through the supervisor model. The observed signals are used to detect the occurrence of state transitions corresponding with target system behavior.

For each state transition in the requirements specification, a different signal may be used to detect that the transition is taken place. A *tracking model* is one representation of such signals.

The tracking model contains all symbolic states and state transitions of the requirements specification. The principal difference between the two models is their stimuli. Stimuli for the tracking model are chosen to detect state transitions corresponding with target system behavior. For each state transition in the tracking model, a stimulus is chosen from the set of signals consumed/generated during the corresponding state transition in the requirements specification.

A primary criterion to select stimuli for the tracking model is signal uniqueness. Uniqueness is a relative concept. In general, signal, S_1 is considered more unique than S_2 if S_1 can be consumed/generated in fewer states than S_2. The precision of state detection is improved by choosing more unique stimuli. This reduces uncertainty within the supervisor as to the actual state transition that occurred and as a consequence improves the supervisor time and/or space complexity.

For the example requirements specification in figure 3.2, a corresponding tracking model is shown in figure 3.3. The model is developed based on the choice that signals X and Y are used to detect paths 1 and 2 through the requirements specification. Note that the SDL system specification (figure 3.3a) has the output channel reversed to introduce the supervisor perspective. Signals, a, b and Z are not used to track the target system and are consumed without effect. Additional (non-SDL) constructs are used to output path information once it has been determined.

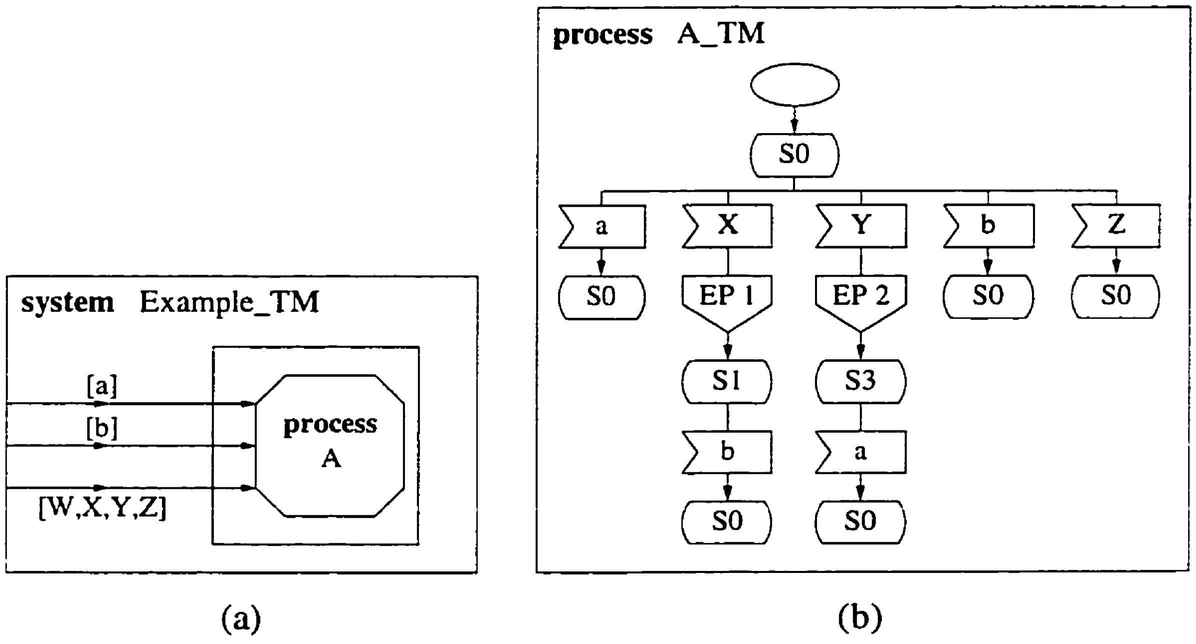


Figure 3.3: Example Tracking Model

3.4 Hierarchical Software Supervisor

Supervision may be decomposed into two smaller sub-problems: (1) tracking the evolution of the target system state through the requirements specification and (2) detailed behavior checking. Lessons learned from disciplines such as AI planning indicate that a problem can be solved more efficiently if decomposed and each part solved with a domain-specific problem solver [6, 34]. The resultant architecture is hierarchical and consists of two functional units: the path detection module (PDM) and the base supervisor (BSup) (figure 3.4).

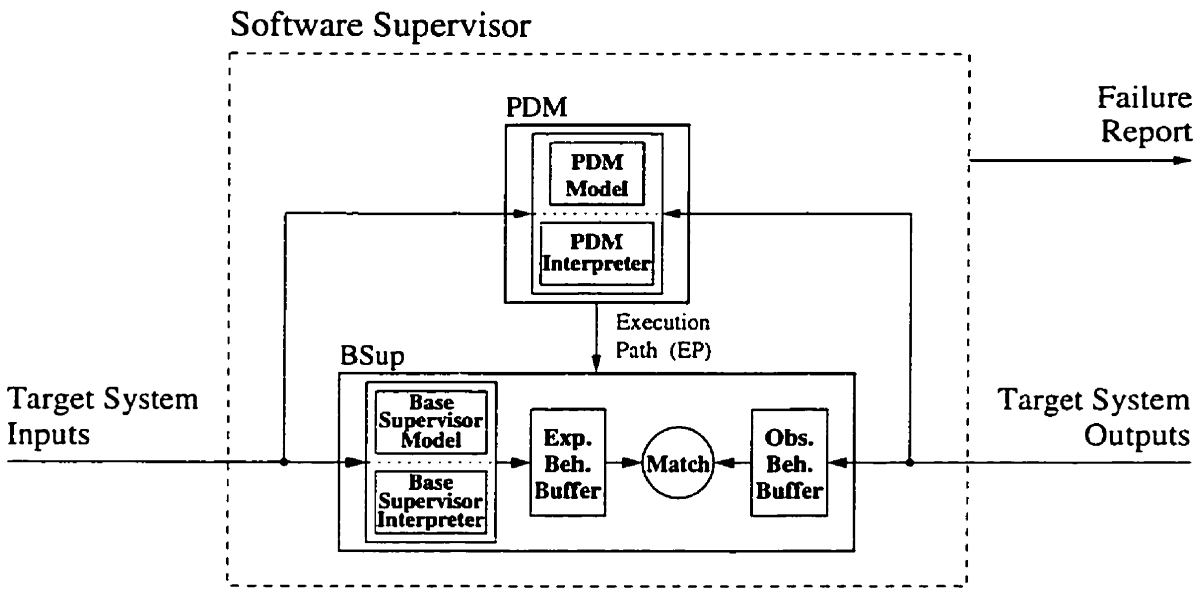


Figure 3.4: Hierarchical Software Supervisor

The PDM tracks the operation of the target system. It accepts both input and output signals of the target system and generates EP information. The PDM consists of a PDM-model, similar to the tracking model described in section 3.3.1 and an interpreter. The PDM-model is derived from the requirements specification.

The BSup is a detailed behavior checker. It accepts target system inputs, outputs and EP information from the PDM. The BSup consists of the five components described in section 3.2. The BSup-model very closely resembles the requirements specification. The interpreter interprets the BSup-model, steering execution according to EP information generated by the PDM.

3.4.1 Operation of the Hierarchical Supervisor

The hierarchical supervisor operates with one of its two functional units active at any point in time. Figure 3.5 shows the operating states of the hierarchical supervisor.

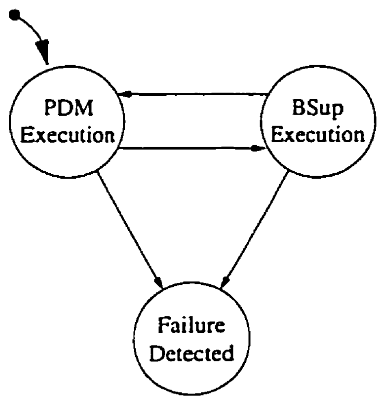


Figure 3.5: Operating States of a Hierarchical Supervisor

Execution begins at the PDM. The PDM executes until it determines the next segment of the EP followed by the target system. The PDM communicates this information to the BSup and passes control to the BSup. The BSup attempts to follow the EP through the requirements specification and generates the expected output(s) corresponding to the EP traversed. The matcher compares the expected output(s) with the actually observed output(s).

Failure Reporting

Failures may be reported by either the PDM or BSup. The PDM reports a failure if the signals generated by the target system could not have been generated along any path emanating from the current symbolic state. The BSup reports a failure in any one of three cases: (1) if the BSup cannot be steered along the path prescribed by the PDM, (2) if the expected and observed behaviors do not match and (3) if a timeout occurs while the BSup waits for path information to be generated by the PDM

Failures described are sub-divided into four commonly-occurring types, categorized by two attributes: the failure category and the hindrance of the PDM's tracking ability. The two failure categories are: (1) spuriously-generated signals and (2) missing or not-generated signals. The presence of a failure may or may not cause the PDM to report an incorrect EP. Both cases are described. The failure types are summarized in figure 3.6.

		PDM Tracking Hindered	
		No	Yes
Categories of Failures	Spurious Signal	TYPE I	TYPE II
	Missing Signal	TYPE III	TYPE IV

Figure 3.6: Failure Types

As an example, consider a hierarchical supervisor that uses the PDM-model shown in figure 3.3 and BSup-model in figure 3.2. Both the PDM and BSup are initially in state $S0$. Examples of the four different failure categories are shown in figure 3.7.

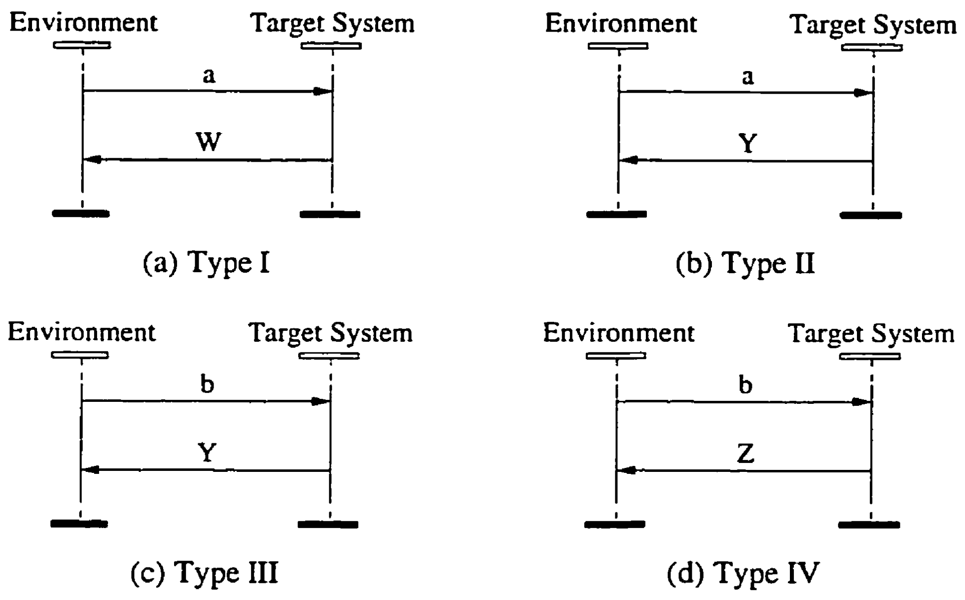


Figure 3.7: Illegitimate Behaviors

The first failure type (figure 3.7a) is an example of an illegitimate output produced by the target system. The behavior does not correspond to any path emanating from the current symbolic state. This type of failure is reported by the PDM.

The second failure type (figure 3.7b) represents an incorrect output generated that corresponds to an existing but incorrect EP (see figure 3.3). The PDM reports that path 2 was traversed by the target system. The BSup attempts to steer execution along path 2 but cannot due to the absence of signal b. The BSup reports the failure.

The third failure type (figure 3.7c) represents a missing signal that does not interfere with the PDM's ability to determine EP information. The PDM reports that path 2 was traversed. The BSup generates an expected behavior consisting of signals, Y and Z. The matcher discovers that the expected behavior does not match the observed behavior. A failure is reported by the matcher.

The final failure type (figure 3.7d) represents a missing signal that interferes with the PDM's ability to detect the EP. In this example, signal Y was not generated by the target system. The PDM cannot determine EP information since it waits for Y, however the BSup has received signals b and Z. The BSup waits T^r_{max} from the receipt of b for EP information from the PDM to account for signals b and Y. If EP information from the PDM has not arrived after this time, the BSup reports a failure.

3.4.2 Supervisor Signal Processing Latency

The PDM tracks target system behavior by waiting for key signals so that the next segment of the EP traversed by the target system can be determined. The PDM typically uses a combination of target system input and output signals. As outlined in section 2.1.1, target system outputs may have a latency of up to T^r_{max} units of time before they are generated by the target system.

The PDM cannot guarantee accurate path detection unless it lags the target system in the processing of events by at least T^r_{max} units of time. Thus out-of-time is a natural mode of operation for the hierarchical supervisor.

In some cases, the PDM may not be able to resolve the EP chosen by the target system. This is principally due to a lack of unique signals that may be generated/consumed in more than one requirements specification state transition. In such a case, the PDM must resort to an approach where several candidate EPs are considered concurrently. Two such approaches (belief method and OPPR) were described in section 3.2.1. The belief method will be used in this thesis due to its maturity over the OPPR approach.

3.4.3 Computational Cost

The hierarchical supervisor makes use of two models and two interpreters. As a first approximation, its time and space cost is twice that of a monolithic one. As will be discussed in the latter parts of this thesis, the computational cost of a hierarchical supervisor is proportional to the number of beliefs generated. Thus a point of indifference between the choice of a hierarchical supervisor and a monolithic occurs one when the hierarchical supervisor eliminates from consideration half of the beliefs generated by a monolithic one. Once more than half of the beliefs can be eliminated from consideration, a hierarchical supervisor becomes more cost effective than a monolithic one.

The time and space cost of a hierarchical supervisor depends on: (1) the amount of non-determinism in the requirements specification, (2) the implementation of non-determinism in the target system and (3) the operational profile. An analytical model of the computational cost of a hierarchical supervisor is left as future work. However, the time and space complexities of a monolithic and hierarchical supervisor are evaluated experimentally for one target system in chapter 7.

Chapter 4

The PDM Model

This chapter describes the derivation of a tracking model from the requirements specification. As mentioned previously, this model is referred to as a *PDM-model.* Recall from section 3.3.1 that the PDM-model is used by the path detection module (PDM) to track target system operation through the requirements specification.

The PDM-model derivation procedure is exemplified with the aid of a non-trivial system; a fragment of a small telephone exchange. The example was chosen to exemplify the main parts of the transformation process which are difficult to illustrate with a trivial example.

The chapter begins with a description of the telephone exchange and its requirements specification. The prominent issues arising in the derivation of a PDM-model are described next followed by the actual derivation procedure.

4.1 Example Software System

The example software system is the call processing software of a small telephone exchange. Its complete specification appears in appendix A. For discussion purposes, the SDL process interaction diagram of the exchange is duplicated in this chapter. It appears in figure 4.1.

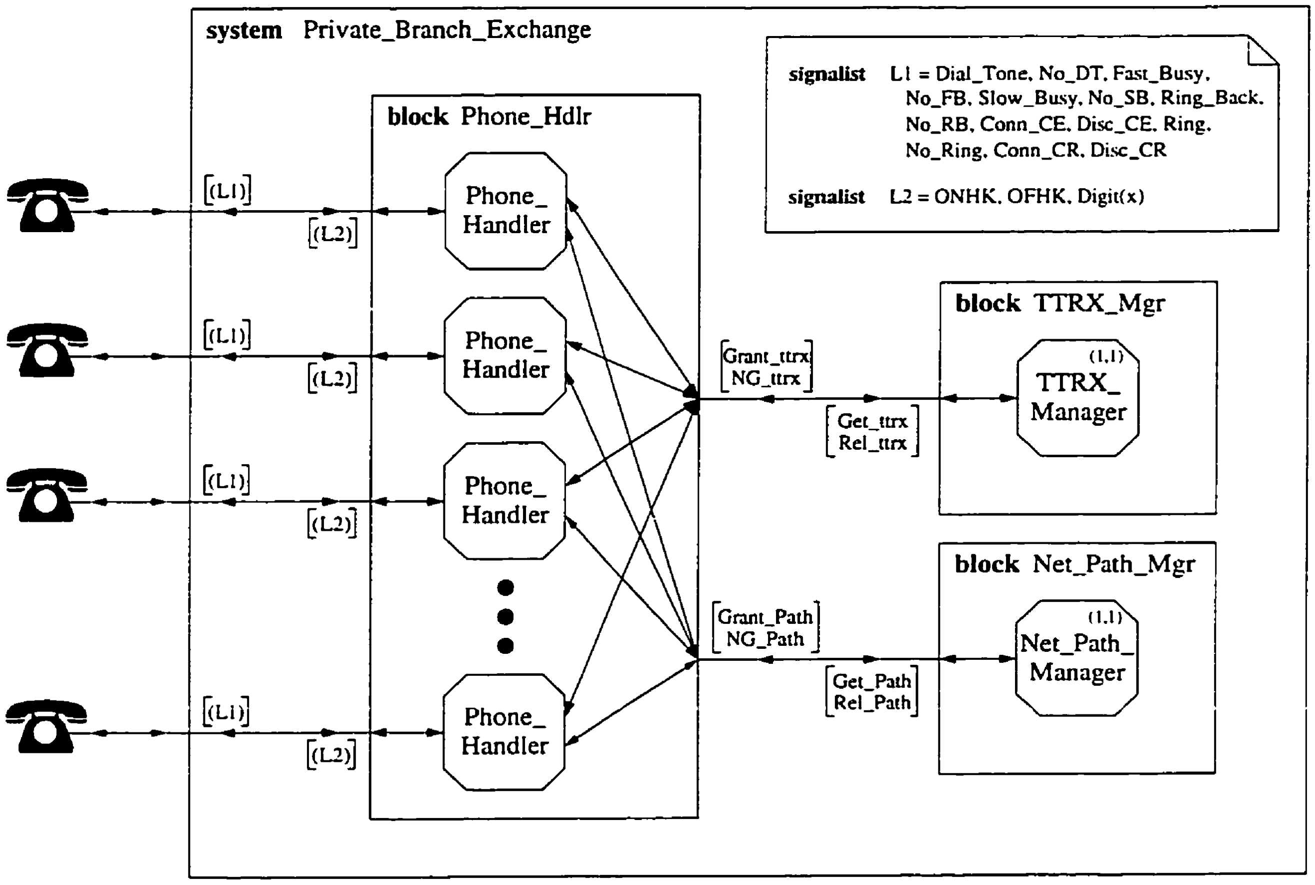
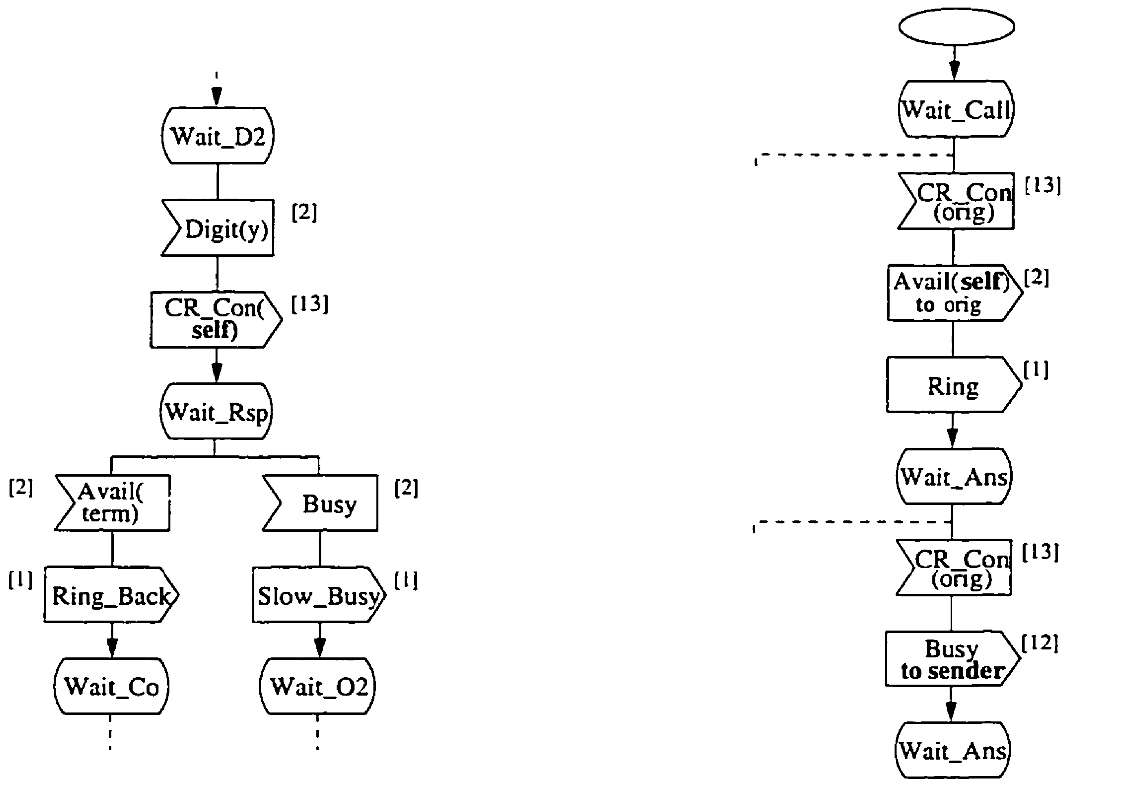

system Private_Branch_Exchange
signalist L1 = Dial_Tone, No_DT, Fast_Busy, No_FB, Slow_Busy, No_SB, Ring_Back, No_RB, Conn_CE, Disc_CE, Ring, No_Ring, Conn_CR, Disc_CR
signalist L2 = ONHK, OFHK, Digit(x)
block Phone_Hdlr
(L1)
(L2)
Phone_ Handler
block TTRX_Mgr
(1,1)
TTRX_ Manager
Grant_ttrx NG_ttrx
Get_ttrx Rel_ttrx
block Net_Path_Mgr
(1,1)
Net_Path_ Manager
Grant_Path NG_Path
Get_Path Rel_Path

Figure 4.1: Telephone Exchange SDL System Specification

The behavior seen by each telephone is defined by a *Phone_Handler* process. *Phone_Handlers* communicate to connect and terminate telephone calls. A separate, bidirectional communication path exists between each pair of *Phone_Handler* processes, represented by implicit SDL signal routes in figure 4.1.

All *Phone_Handlers* are identical. To simplify the discussion, only two fragments of the *Phone_Handler* are shown (figure 4.2). They deal with an originating party dialing the final digit of the telephone number and requesting connection with the terminating party. For brevity, identification of the destination process for signals *req_connect*, *remote_avail* and *remote_busy* is omitted as are portions of the specification dealing with exceptions such as timeouts and uncompleted dialing. The numbers in brackets ([···]) appearing in figure 4.2a will be described later.



(a) Originating Fragment (b) Terminating Fragment

Figure 4.2: Fragments of the Phone Handler Specification

4.1.1 Illustration of Nondeterministic Behavior

Consider the A and B call Z scenario. A chart illustrating the signals exchanged between *Phone_Handlers* A, B and Z and the environment is shown in figure 4.3. If both A and B dial the final digit of Z within a brief interval of each other, the indeterminate delay on the inter-process communication paths between the environment and processes A and B permits either A or B to complete the call to Z (the other will receive slow busy tone). Figure 4.3a shows the case where the delay to process B is larger and 4.3b where it is smaller than the delay from the environment to A. For this particular scenario, the specification permits two legal behavioral alternatives. Both alternatives must be considered by a supervisor.

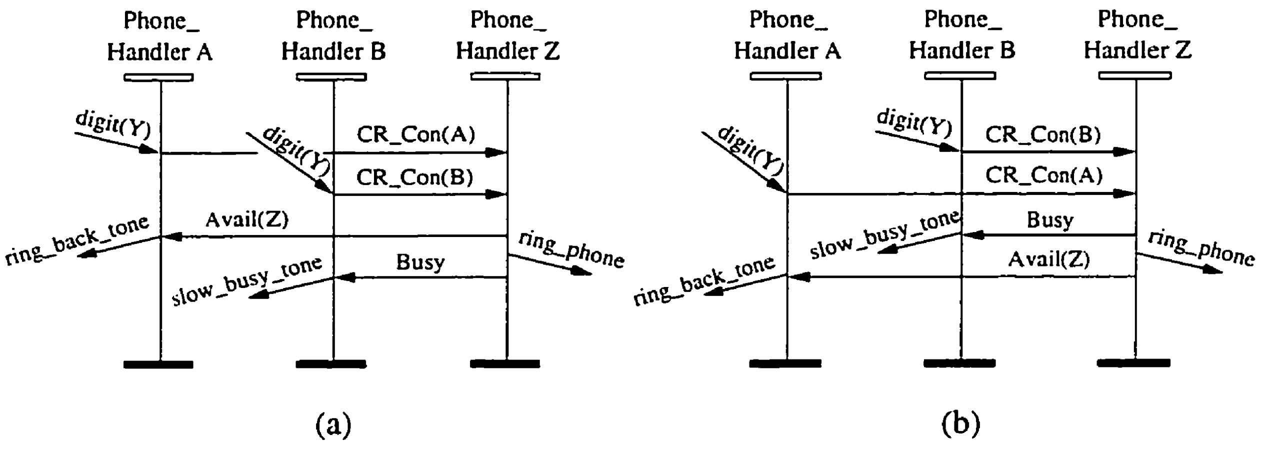


Figure 4.3: Behavioral Alternatives for the A and B Call Z Scenario

At the input port of a process, specification nondeterminism permits the two *CR_Con* signals (figure 4.4) to be consumed in either order. Provision must be made by the supervisor to consider all possible signal orderings if consumption order uncertainty exists. The consequence of considering only a subset of all possible signal permutations is that the supervisor may generate spurious failure reports. For a process with n signals in its input port, the upper bound on the number of signal permutations is $n!$. This may lead to a potentially large computational complexity if all possible signal permutations must be explored.

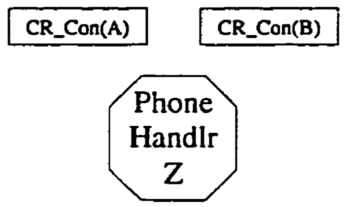


Figure 4.4: Permuteable Signals at the Input of a SDL Process

4.2 Issues in the Derivation of the PDM-Model

As mentioned in chapter 3, stimuli for the PDM-model are chosen based on their uniqueness. A metric of uniqueness is described first. A discussion of maintaining sequences of internal state transitions or causality pathways in the PDM-model is described next. The section concludes with a description of data flows in the PDM-model. Data flows appearing in the requirements specification must be maintained in the PDM-model.

4.2.1 Identification of State Transitions

As discussed in section 3.3.1, the occurrence of a state transition is detected with either target system input or output signals. The motivation for using signals other than target system inputs to detect state transitions is to reduce the number of required signal permutations and as a result the computational complexity of the supervisor.

Signals in the PDM-model, used to detect state transitions, are chosen based on their uniqueness. In the requirements specification, signals that are either consumed/generated during fewer state transitions are considered more unique than signals consumed/generated during more transitions. The precision with which state transitions can be detected improves as the uniqueness of signals used increases.

The notion of a *uniqueness metric* or u-metric is used to quantify the idea of signal uniqueness. The u-metric is defined for all signal-transition pairs in the requirement specification.

Definition 4.2.1 (Uniqueness Metric (u-metric)) *Let P be an SDL process and s an SDL signal that initiates a state transition or is generated during a state transition in P. The u-metric(s, P) is defined as:*

- *if s is an input signal, u-metric(s, P) is defined as the number of state transitions initiated by s in P*
- *if s is an output signal, u-metric(s, P) is defined as the number of state transitions in P where s is generated*

The ability to map a signal to fewer state transitions reduces the number of behavioral alternatives the supervisor must consider. The u-metric is used as a basis to select stimuli for the PDM-model by the derivation procedure to be discussed in section 4.3. Signals with lower u-metric values are preferred over signals with higher u-metric values.

Dynamic Metrics

In general, metrics for PDM-model stimulus selection may be classified as either *static* or *dynamic*. Static metrics take into consideration the specification but not the corresponding operational profile of the target system. However, dynamic metrics also take into consideration the operational profile.

This thesis describes only one static metric (u-metric). Other static or dynamic metrics may be developed. The PDM-model transformation process to be described remains the same regardless of the metric used.

Example

Consider the example in figure 4.2. The u-metrics are shown in square brackets ($[\cdots]$) beside each signal in figure 4.2. The u-metrics are computed based on the full requirements specification of the telephone exchange appearing in Appendix A.

Note that signal *CR_Con* causes state many transition (in Appendix A, the star-state notation is used to capture this) and as a result it has a high u-metric value.

4.2.2 Causality Pathways

A target system input signal may cause a sequence of n state transitions in one or more processes of the requirements specification. The n state transitions may produce zero or more externally observable outputs (target system outputs). This series of state transitions shall be referred to as a *causality pathway* (CP).

As an example, consider the specification in figure 4.2. Assume that the processes shown are in states $Wait\_D2$ and $Wait\_Call$. If signal $digit(Y)$ is consumed, it would cause state transition $Wait\_D2 \rightarrow Wait\_Rsp$ which would cause $Wait\_Call \rightarrow Wait\_Ans$ followed by $Wait\_Rsp \rightarrow Wait\_Co$. This collective set of state transitions, initiated by signal, $digit(Y)$ is referred to as a causality pathway.

Figure 4.5a illustrates this CP. A compact notation is used. Stimuli are denoted as incoming lines to the process. Generated outputs are denoted as outgoing lines. Actual state transitions are abstracted.

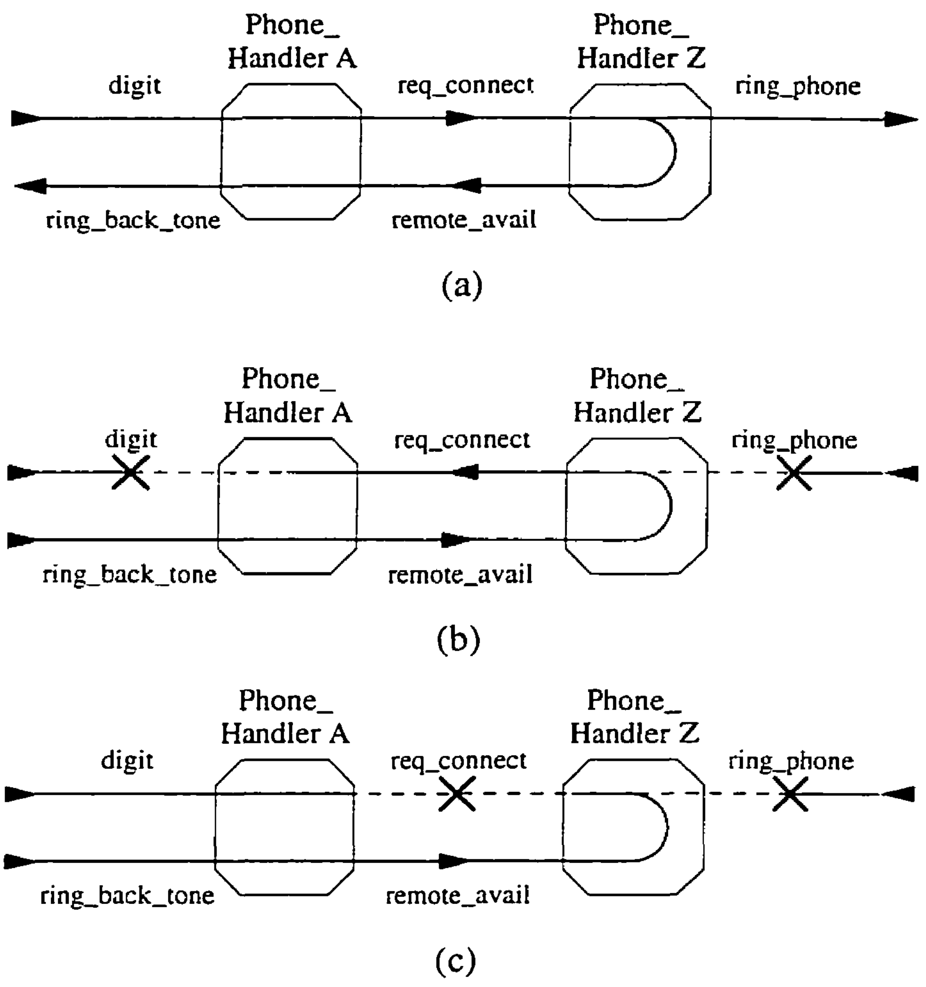


Figure 4.5: Causality Pathway and Causality Pathway Tracing

The PDM, responsible for detecting state transitions that occur, effectively traces each CP. CPs can be traced in a forward direction, backward direction or a combination of the two. A CP is traced forward by using stimuli of the requirements specification as stimuli in the PDM-model. Conversely, a CP is traced backwards by using outputs from the specification as stimuli in the PDM-model. Two issues arise when CPs are traced backwards by the PDM.

The first issue deals with a possible violation of signal sequencing in the detection of state transitions. As an example, consider the specification in figure 4.2 and the corresponding CP in figure 4.5a. If the entire CP is traced backwards while process A is in state, $Wait\_D2$, the PDM would be required to report that transition $Wait\_Rsp \rightarrow Wait\_Co$ occurred before transition $Wait\_D2 \rightarrow Wait\_Rsp$ (figure 4.5b).

The solution to this problem is to trace the CP only in the forward direction or to use a combination of forward and backward tracing. For the previous example, one possible forward/backward tracing that solves the described signal sequencing problem is shown in figure 4.5c.

The second issue deals with the consistency in the selection of stimuli for the PDM-model between individual processes. Consider two state transitions, $S0 \rightarrow S1$ and $Sa \rightarrow Sb$ occurring in two different processes such that the occurrence of $S0 \rightarrow S1$ triggers $Sa \rightarrow Sb$ (i.e. both transitions are part of a single CP) (figure 4.6a). If in the PDM-model, the identical signal is used as a stimulus for both transitions, deadlock will occur (figure 4.6b). Clearly, stimuli that are chosen in one process constrain the choice of stimuli in other processes.

4.2.3 Signal Parameters

Parameters tagged to signals constitute the data flow through the requirements specification. The state of a process is dependent on the values of data. Relevant data flows must be maintained in the PDM-model.

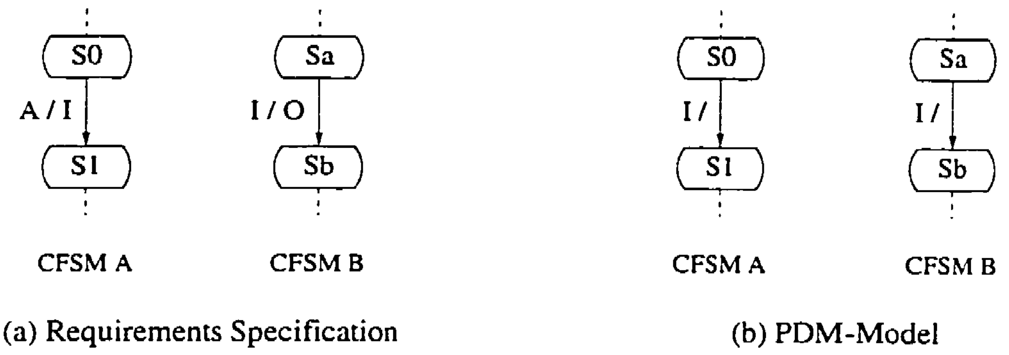


Figure 4.6: Example: PDM-Model Deadlock

The two types of parameters are addressed: implicit and explicit. Explicit parameters are specified by a specification writer. As an example the signal, *digit(Y)* in figure 4.2 uses an explicit parameter to carry a digit information.

Implicit parameters are appended to each signal by the the semantics of the specification formalism. Examples of such parameters include the sender ID of a signal, the signal type, destination ID, etc. For brevity, we restrict discussion of implicit parameters to the sender ID and signal type. Other implicit parameters may be treated in a similar manner.

In the PDM-model, all parameters used by a process must be communicated to the process. In many cases implicit parameters are not actually used and can be dropped to simplify the transformation and the resultant PDM-model.

4.3 PDM-Model Transformation Algorithm

This section presents the algorithm for transformation of the requirements specification into the PDM-model. The section begins with an overview. The transformation algorithm is presented next, followed by an example.

The presentation of the PDM-model transformation algorithm assumes that signals in the requirements specification have unique names. Formally, consider two signal send constructs, s_1 and s_2 appearing in the requirements specification. If a state transition, T does not exist such that both s_1 and s_2 *could* cause T under any given scenario, signals s_1 and s_2 must have different symbolic names. The above requirement can be enforced by simply relabeling the symbolic signal names in the requirements specification.

4.3.1 Overview

The PDM-model differs from the specification primarily in its stimuli. All states and state transitions in the original specification appear in the PDM-model.

Path information is communicated to the BSup on the occurrence of each PDM state transition. Path information consists of a sequence of stimuli that if consumed by the BSup would steer execution along the same path as determined by the PDM.

The PDM-model transformation consists of two parts: (1) stimuli selection and (2) model generation. Stimuli selection successively eliminates PDM-model stimuli (initially, all signals generated and consumed during a state transition are candidate stimuli for the PDM-model). Stimuli selection terminates when exactly one signal signal remains for each state transition. At this point model generation is invoked. Model generation constructs a communicating extended finite state machine with the chosen stimuli. The result is the PDM-model.

Stimuli selection is the most challenging part of the PDM-model transformation process. This is due to the fact that the selection of a stimulus for a particular state transition may constrain the choice of stimuli for adjacent state transitions on one or more CPs. These constraints are represented as a *constraint graph* so that as stimuli are chosen, other *inconsistent* stimuli can be removed from consideration.

The components and data flows of the transformation process are shown in figure 4.7. The stimuli selection and model generation components of the transformation are described below in further detail.

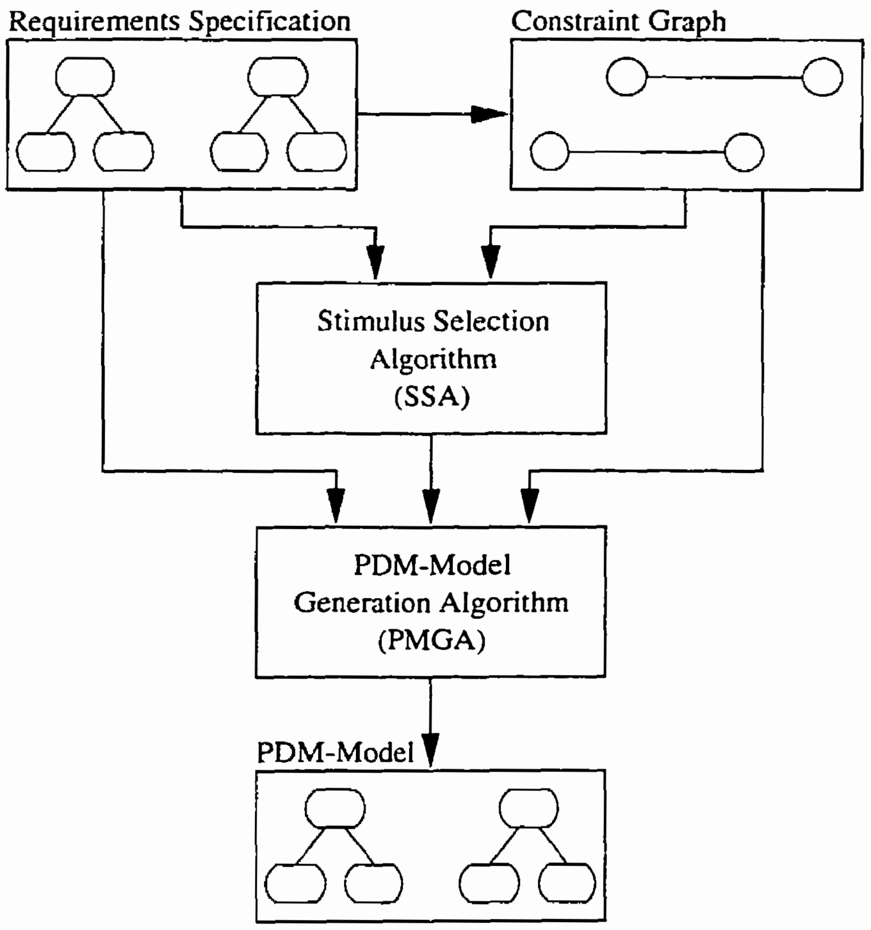

Requirements Specification
Constraint Graph
Stimulus Selection Algorithm (SSA)
PDM-Model Generation Algorithm (PMGA)
PDM-Model

Figure 4.7: PDM-Model Transformation Process

Stimuli Selection

The stimulus selection algorithm considers the requirements specification at three independent levels of abstraction.

The first level considers the data-flows through the specification. The stimuli selection algorithm ensures that data-flows remain in the PDM-model as they influence the state of processes. All processes are considered at this level.

The second level deals with the consistent selection of stimuli. As discussed, choosing a stimulus for a state transition in process X will influence the choices of stimuli in adjacent processes (processes that communicate directly with process X). Consistency of stimulus selection requires consideration of stimuli for adjacent processes.

Stimuli are actually chosen at the third level. At this level, each process is considered independently of other processes. Stimuli are chosen based on their uniqueness within the specification. A signal that causes or is generated in few state transitions will give the PDM more precise information as to which state transition occurred than would a signal that may be consumed/generated in many.

PDM-Model Generation

The PDM-model generator begins with a model that represents the requirements specification in topology. All finite state machines, states and state transitions remain the same. State transitions are unlabeled (i.e. no input or output signals appear on the transition).

The PDM-model generation consists of three steps. First the selected stimuli are added to the model. Signal output constructs are added to state transitions that are to cause internal state transitions based on the choice of stimuli. Finally, state transitions are added that consume target system input or output signals not chosen as stimuli. These signals are consumed without effect[1].

4.3.2 Constraint-Based Stimulus Consistency

To ensure consistency between the selection of stimuli for the individual state transitions of the PDM-model, the problem is projected as a finite-domain *constraint satisfaction problem* (CSP) [48]. The classic formulation of CSP problems consist of three components: (1) variables, (2) variable domains and (3) constraints between variables. A constraint satisfaction algorithm is used to ensure that all constraints are satisfied by successively restricting elements or ranges of elements from a variable's domain. The CSP is said to be *solvable* if at least one variable assignment[2] exists that satisfies all constraints.

For the model transformation problem, state transitions are mapped into variables, candidate PDM-model stimuli for a particular transition are mapped to variable domains and inequality constraints are placed between adjacent state transitions of a CP. The interpretation of the constraints is that adjacent state transitions cannot be initiated by a single signal generated or consumed during both transitions. The CSP can then be represented as a graph where nodes represent state transitions, contents of nodes represent possible PDM-model stimuli and labeled arcs represent constraints.

As an example, a constraint graph was derived for the specification fragments

[1]These transitions are equivalent in semantics to SDL implicit transitions. They are described explicitly for completeness purposes only.

[2]A variable assignment may be considered as an elimination of all domain values except one for a given variable.

in figure 4.2. The graph appears in figure 4.8. Note that due to space limitations, the graph captures only the originating fragment for phones A and B and the terminating fragment for phone Z. A complete constraint graph must capture all interactions of all processes appearing in the communication topology (figure 4.1).

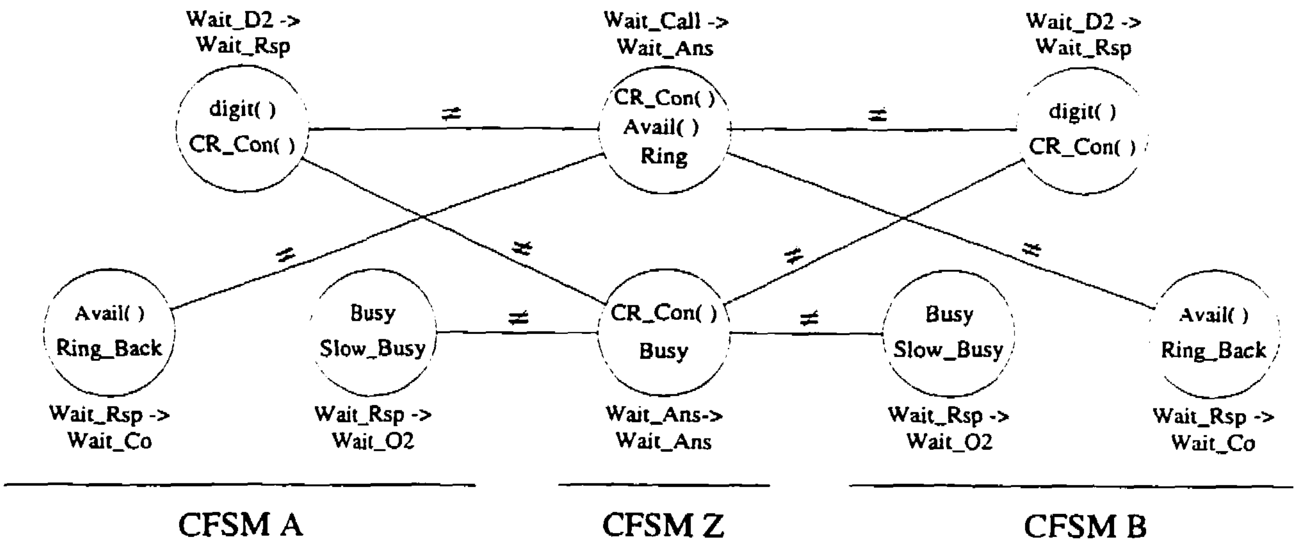


Figure 4.8: Segment of Constraint Graph

A constraint graph is said to be *consistent* if for each variable's domain value, at least one corresponding domain value exists in each variable linked by a constraint that satisfies each corresponding constraint. The elimination of domain values from variables may cause the graph to become inconsistent.

As an example, if signal $digit(Y)$ is removed from transition. $Wait\_D2 \rightarrow Wait\_Rsp$, signal CR_Con becomes the stimulus for the aforementioned transition. Thus the stimulus assignment in transition $Wait\_D2 \rightarrow Wait\_Rsp$ is no longer consistent with the assignment of CR_Con as the stimulus for either of $Wait\_Call \rightarrow Wait\_Ans$ or $Wait\_Ans \rightarrow Wait\_Ans$.

Constraint propagation is a technique to eliminate inconsistent variable domain values. A constraint propagation algorithm accepts as input an inconsistent constraint graph and returns a consistent constraint graph, provided that a consistent variable assignment exists. Constraint propagation algorithms operate by successively removing inconsistent domain values until the graph becomes consistent. The algorithm is applied each time a value is removed from a variable's domain. A survey of such algorithms can be found in [37].

From the above example, if the described graph was an input into a constraint propagation algorithm, the algorithm would eliminate signal CR_Con from the domain values of transitions $Wait\_Call \rightarrow Wait\_Ans$ and $Wait\_Ans \rightarrow Wait\_Ans$ and signal $Busy$ from the two $Wait\_Rsp \rightarrow Wait\_O2$ transitions.

4.3.3 PDM-Model Transformation Algorithm

The PDM-model transformation algorithm is presented in two parts. The first part is the stimulus selection algorithm (SSA). It is used to choose stimuli for the PDM-model. The second part, the PDM-model generation algorithm (PMGA), generates the PDM-model based on the stimuli chosen by the SSA. Recall, the transformation process was shown graphically in figure 4.7.

In the descriptions of the SSA and PMGA, the following notation will be used: T_i^{rs} will be used to refer to transition i in the requirements specification, T_i^{cg} to the corresponding node i in the constraint graph and T_i^{pm} to the corresponding transition in the PDM-model.

4.3.4 Stimulus Selection Algorithm

The SSA accepts as input the requirements specification (*Spec*) and the corresponding constraint graph, derived from the requirements specification (*Cons_Graph*). The SSA returns a stimulus for each state transition in the requirements specification.

The SSA can be subdivided into three parts. The first part checks for causality violations in the detection order of state transitions as described in section 4.2.2. The second part of the algorithm ensures that data flows remain intact in the PDM-model, as outlined in section 4.2.3. The final part of the algorithm actually selects signals that will be used to identify state transitions in the PDM-model (i.e. the stimuli for state transitions). The selection process is based on the u-metric, described in section 4.2.1.

The SSA appears in figure 4.9. A textual summary of the algorithm follows.

Causality Violations Check [lines 1–7]

As described in section 4.2.2, a violation of causality occurs if an attempt is made to determine that a transition occurs after the current one. The SSA statically detects possible causality violations by tracing the CPs through the requirements specification. If a CP is found that crosses a particular process more than once, the algorithm forces the portion of the CP which is crossed more than once to be traced forward.

As an example, the CP shown in figure 4.5a crosses process A twice. The SSA enforces that the first transition be processed in a forward direction. This effectively restricts the entire CP to be processed either entirely in a forward direction

```
Algorithm SSA(Spec, Cons_Graph)
1. for all state transitions $T_i^{rs} \in Spec$
2.     $CP$ = the set of all forward causality pathways passing through at $T_i^{rs}$
3.     if (exists a $cp \in CP$ such that cp initiates two or more state transitions
             in the process where transition $T_i^{rs}$ appears)
4.         stimulus($T_i^{cg}$) = stimulus($T_i^{rs}$)
5.         apply constraint propagation algorithm to Cons_Graph
6.     end if
7. end for
8. for all state transitions $T_i^{rs} \in Spec$
9.     if (stimulus($T_i^{rs}$) carries an explicit, used parameter
10.        stimulus($T_i^{cg}$) = stimulus($T_i^{rs}$)
11.        apply constraint propagation algorithm to Cons_Graph
12.    end if
13.    if (if implicit parameter(s) of stimulus($T_i^{rs}$) cannot be statically determined
14.        delete all signals from the domain of $T_i^{cg}$ that do not carry needed
             implicit parameters
15.    end if
16. end for
17. for all nodes, $T_i^{cg} \in Cons\_Graph$
18.    while(number_of_elements_in_domain($T_i^{cg}$) > 1) do
19.        compute collective u-metric for each element in $T_i^{cg}$
20.        delete element in $T_i^{cg}$ with largest collective u-metric
21.        apply constraint propagation algorithm to Cons_Graph
22.    end while
23. end for
24.return (stimuli)
25.end Algorithm
```

Figure 4.9: Stimulus Selection Algorithm

(figure 4.5a) or partially forward and partially backward. One example of the latter is illustrated in figure 4.5c.

Maintaining Data Flows [lines 8–16]

Data flows that appear in the requirements specification must be maintained in the PDM-model where required. The SSA checks all data flows and determines if the data is required. If so, it imposes constraints on stimuli to ensure that the data flows will appear in the PDM-model.

The SSA checks both explicit (programmer specified) and implicit parameters. As discussed, implicit parameters consist of the ID of the sender process for each signal only.

The precision of the PDM-model in detecting state transitions is reduced by imposing constraints on stimuli to maintain dataflows. In some cases, some signal parameters may be determined statically, which reduces the constraints on stimuli. As an example, the sender ID of a signal can often be determined statically from the communication structure if there is only one process that could actually generate the signal.

For parameters that are used and cannot be determined statically, the CP is constrained to be processed in a forward direction. This ensures that the direction of the CP remains identical to that in the requirements specification.

Stimuli Selection [lines 17–23]

For the remaining transitions having two or more candidate stimuli, stimuli are selected based on signal u-metric. Signals having lower u-metrics are preferred since they indicate which transition has occurred with greater certainty than a signal with a higher u-metric.

Choosing a stimulus for a particular state transition constrains choices of other stimuli along the CP as described in section 4.2.2. Stimuli are chosen to minimize the restriction on the use of signals with small u-metrics in adjacent processes. A signal, s chosen as a PDM-model stimulus eliminates other candidate stimuli from being selected. The sum of all signals u-metrics that are eliminated as a result of choosing s shall be referred to as the *collective u-metric.* Note that the collective u-metric includes the u-metric of s.

The final part of the SSA operates by repeatedly removing candidate stimuli from a particular state transition, T. The stimulus with the highest collective u-metric is removed. This means that if only one signal is left in a node that the signal becomes the stimulus for the node (state transition). A constraint propagation algorithm is applied after the removal of each stimulus to ensure consistency. This process repeats until each node in the constraint graph contains exactly one signal.

The reader should note that in the worst case, the SSA will choose stimuli for the PDM-model that are identical to those in the requirements specification. This would occur, for example, in a specification that does not generate any outputs. As a result, a consistent selection of stimuli for the PDM-model always exists. However, in some cases the stimuli selection algorithm may return an inconsistent set of stimuli. In such a case the constraint propagation algorithm could be combined with search to exhaustively consider the search space. From experience, such a scenario has not been encountered in the target system specifications considered.

4.3.5 PDM-Model Generation Algorithm

The PDM-model generation algorithm accepts as input the requirements specification, a stimulus for each state transition selected by the SSA and the unaltered constraint graph. The algorithm creates the PDM-model. The PDM-model appears at two levels of abstraction, similar to the corresponding requirements specification: (1) the system or process interaction level and (2) the process level.

At the system level, all processes appearing in the requirements specification appear in the PDM-model. The channels and signal routes connecting processes differ, principally due to the possibility of signal direction reversal based on the choice of stimuli for the PDM-model.

The communication topology is generated based on the following rules. Consider two signals, s_1 and s_2 traveling from processes P_1 to P_2 in the PDM-model. If the two signals traveled on a single channel/signal route in the requirements specification, a single channel/signal route is created between process P_1 and P_2. If the two signals traveled on different channels/signal routes, two separate channels/signal routes are created between processes $P1$ and $P2$. Note that some internal signals appearing in the requirements specification may not appear in the PDM-model and as a consequence the PDM-model may contain fewer channels and/or signal routes than the specification.

The process level PDM-model generation algorithm (Algorithm PGMA) is described in three parts. The first part creates the transitions using the stimuli prescribed by the SSA. The second part introduces constructs to communicate path information from the PDM-model to the BSup. The final part adds implicit signal consumption constructs for any signals from the environment not used as stimuli.

The PMGA is shown in figure 4.10. A textual summary of the algorithm follows.

```
Algorithm PMGA(Spec, Stimuli, Cons_Graph)
1.  create all process in PDM-Model having stimuli from Stimuli
2.  for all state transitions, $T_i^{rs} \in Spec$
3.      for all transitions, $T_j^{cg}$ having a constraint between $T_i^{rs}$
            and $T_j^{cg} \in Cons\_Graph$
4.          if (stimulus($T_j^{cg}$) $\in T_i^{rs}$)
5.              add output signal stimulus($T_j^{cg}$) to transition, $T_i^{pm}$
6.          end if
7.      end for
8.  end for
9.  for all state transitions, $T_i^{pm} \in$ PDM-Model
10.     add BSup-output construct to transition $T_i^{pm}$ to communicate signal
            stimulus($T_i^{rs}$) to BSup
11. end for
12. for all state transitions, $T_i^{rs} \in Spec$
13.     for all signals, sig $\in T_i^{rs}$
14.         if (sig $\neq$ stimulus($T_i^{pm}$)) and (sig originates from environment)
15.             if (sig appears before stimulus($T_i^{pm}$) in Spec)
16.                 add implicit transition in state before $T_i^{pm}$ occurs
                        to consume sig without effect
17.             else
18.                 add implicit transition in state after $T_i^{pm}$ occurs
                        to consume sig without effect
19.             end if
20.         end if
21.     end for
22. end for
23.end Algorithm
```

Figure 4.10: PDM-Model Generation Algorithm

Creation of PDM-Model State Transitions [lines 1–8]

This portion of the algorithm creates all state transitions with the stimuli specified by the SSA. For state transitions triggered by internally generated signals, the PMGA adds output constructs to the state transitions responsible for triggering these transitions.

Insertion of Path Information to the BSup [lines 9–11]

Constructs to communicate path information are added to each transition in the PDM-model. The path information is used to steer the BSup along the path of the PDM. Path information consists of the triggering signal name, explicit and implicit parameters. The reader should note that the path information consists of the triggering signal that would have caused the state transition in the *requirements specification*, not in the PDM-model.

Addition of Implicit Signal Consumption Constructs [lines 12–22]

All signals generated and consumed by the target system travel to the PDM. Not all signals from the environment are used as stimuli in the PDM-model. Explicit signal consumption constructs are added for all signals from the environment not used as stimuli.

Signals may be consumed without effect before or after the corresponding state transition occurs in the PDM. The main issue is to preserve the order of signal consumption specified by the requirements specification.

For explanation purposes, the signal to be consumed without effect shall be referred to as S. S is generated or consumed in the requirements specification during transition T. Assume that the chosen stimulus for transition T in the PDM-model is $Stim$. Note that $Stim \neq S$

If, during transition T in the requirement specification, signal S is consumed or generated before $Stim$, then S in the PDM-model must be consumed before transition T takes place. If during state transition T, signal S is generated after $Stim$ is consumed or generated, then in the PDM-model, $Stim$ must be consumed directly after transition, T takes place (i.e. in the terminating state of transition T).

4.3.6 PDM-Model Transformation Example

As an application example the SSA and PMGA are applied to the specification fragment illustrated in figure 4.2. The corresponding constraint graph for the specification is shown in figure 4.8. The description of each algorithm's execution is broken down into the three steps used during the description of the algorithm.

Stimulus Selection Algorithm

The outputs from intermediate stages in the execution of the SSA are illustrated in figure 4.11.

Causality Violation Check

The algorithm begins by tracing each of the CPs through the specification. In doing so, it is determined that the CP, initiated by signal *digit(y)* in process A crosses process A twice. For this reason, the stimulus for transition, $Wait\_D2 \rightarrow Wait\_Rsp$ is set to the stimulus of the requirements specification. A similar stimulus selection

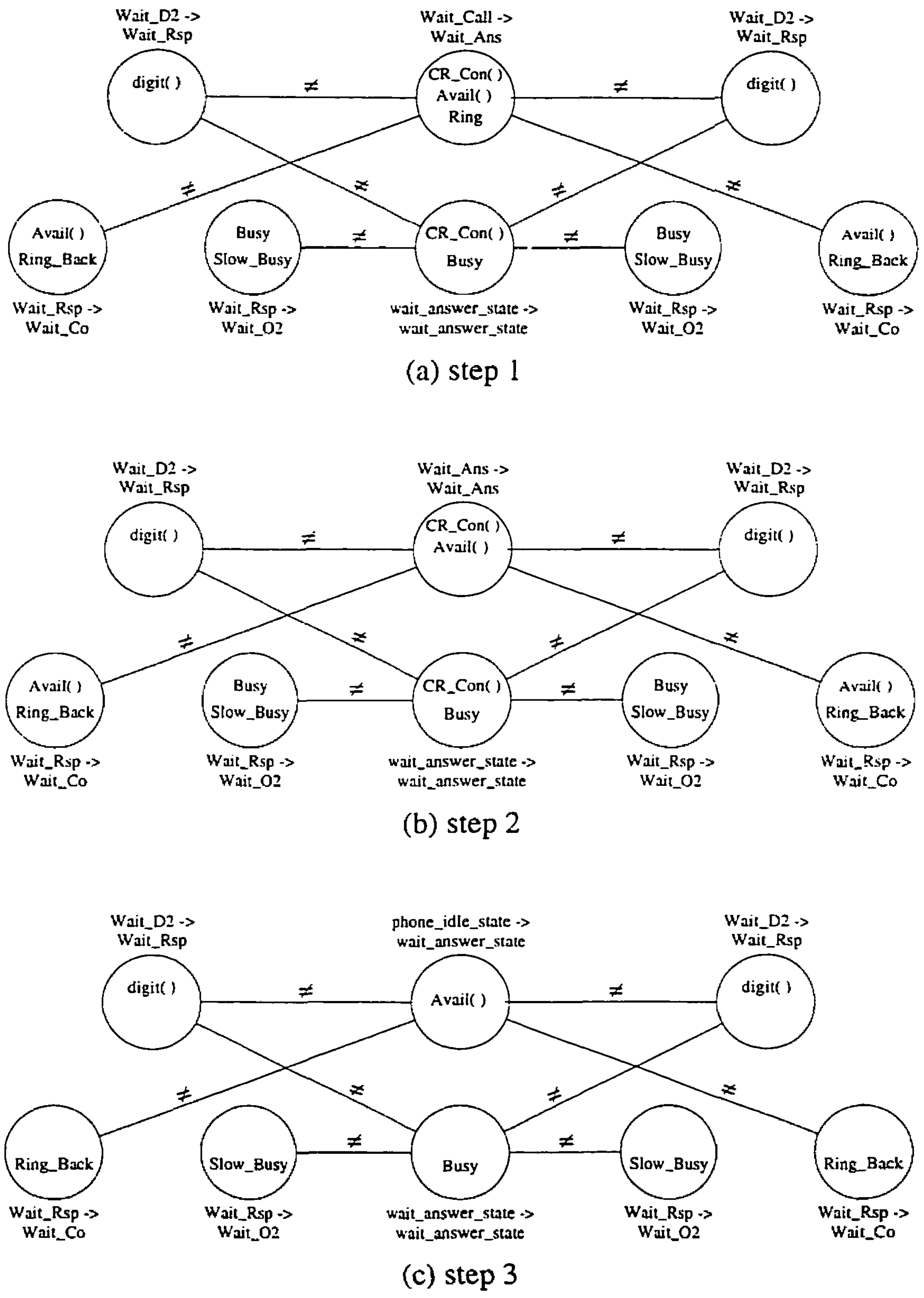


Figure 4.11: Application of the Stimulus Selection Algorithm

is made for the corresponding transition in process B. The resulting constraint graph is shown in figure 4.11a.

Maintaining Data Flows

The second step of the SSA examines the explicit and implicit parameters carried by all signals remaining in the constraint graph. The parameter (Y) carried by signal, *digit(Y)* is needed. However in the previous step, this signal was instantiated as the stimulus in the PDM-model (if it was not instantiated as the stimulus in the previous step, it would have been during this step). Signal *Ring* during state transition $Wait\_Call \rightarrow Wait\_Ans$ does not carry the sender ID of the stimulus *CR_Con* in the PDM-model. This information is needed to communicate path information to the BSup and cannot be determined statically since it depends on parameters not locally known to the process. For this reason, *Ring* is eliminated from the PDM-model as a candidate stimulus (figure 4.11b).

Stimuli Selection

During the final step, the remaining stimuli are chosen. Signal *CR_Con* is a candidate stimulus for transition $Wait\_Call \rightarrow Wait\_Ans$. This signal is eliminated as a candidate stimulus since signal *Avail* has a lower u-metric value of 2. Signal *Avail* is chosen and the constraint propagation algorithm invoked which in turn eliminates signals, *Avail* from transition $Wait\_Rsp \rightarrow Wait\_Co$.

For transition $Wait\_Ans \rightarrow Wait\_Ans$, signal *Busy* has a lower u-metric value. Thus *Cr_Con* is eliminated. The constraint propagation algorithm is invoked. The final constraint graph is shown in figure 4.11c.

PDM-Model Generation Algorithm

The intermediate stages in the execution of the PMGA are illustrated in figures 4.12 - 4.14.

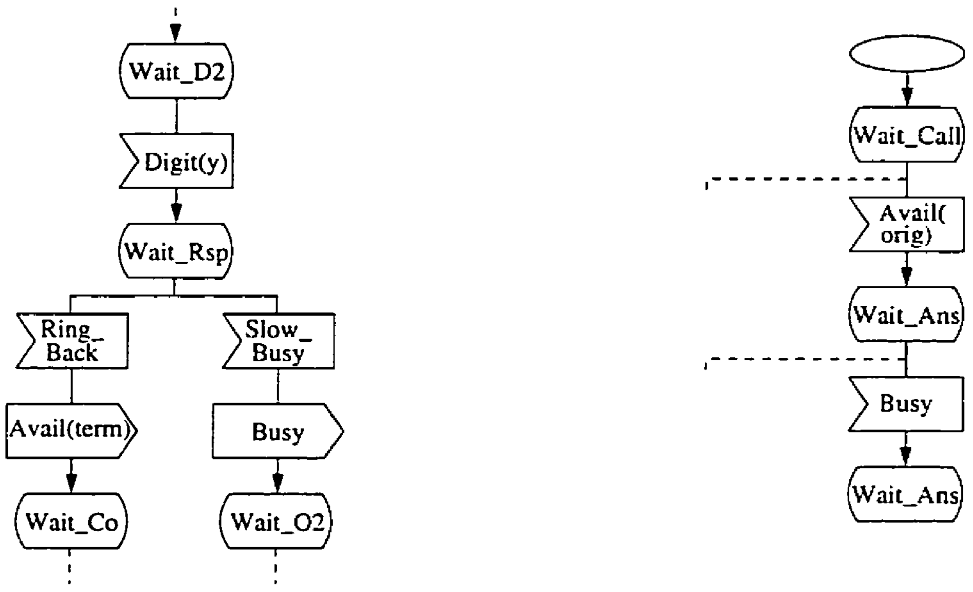


Figure 4.12: Application of the PDM-Model Generation Algorithm (1/3)

Creation of PDM-Model State Transitions

The algorithm begins by creating a PDM-model. The PDM-model contains all state transitions of the original requirements specification. The stimuli generated by the SSA are used as stimuli in the PDM-model (figure 4.12). State transitions, *Wait_Call* $\rightarrow$ *Wait_Ans* and *Wait_Ans* $\rightarrow$ *Wait_Ans* are triggered by internally generated signals. Output constructs are added to these transitions to generate these signals.

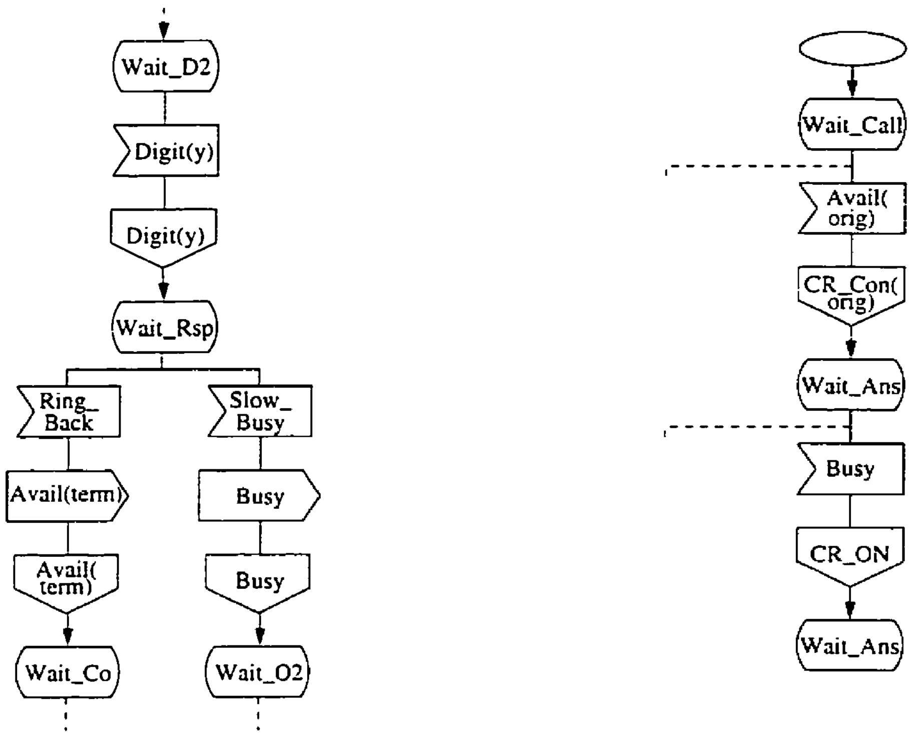


Figure 4.13: Application of the PDM-Model Generation Algorithm (2/3)

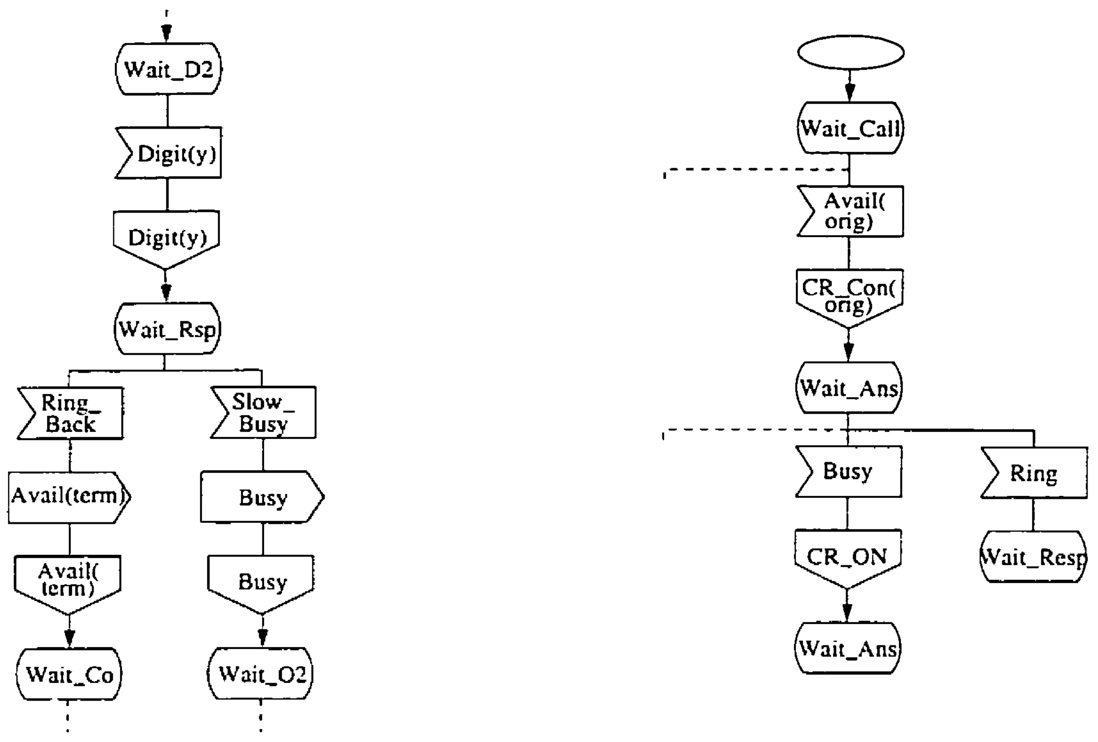


Figure 4.14: Application of the PDM-Model Generation Algorithm (3/3)

Insertion of Path Information to the BSup

The second step of the PMGA adds output constructs to communicate path information to the BSup. Path information consists of the signal that would cause the corresponding state transition in the requirements specification. Note that all explicit and implicit parameters must be defined for this signal (figure 4.13).

Addition of Implicit Signal Consumption Constructs

The final step of the algorithm adds explicit signal consumption constructs for all signals from the environment not used as stimuli. For this example, a signal consumption construct is added for signal *Ring*. In the requirements specification, it is generated after signal *Avail* and as a result it must be consumed after the transition has taken place in the PDM-model (figure 4.14).

Chapter 5

The Path Detection Module Interpreter

This chapter outlines the theory and operation of the PDM interpreter. The PDM interpreter interprets a PDM-model, which is an SDL specification. For this reason the PDM-interpreter closely resembles the SDL interpreter.

The chapter begins with an overview of the interpreter. The notion of time within the interpreter is subsequently described. The two approaches used to deal with behavioral alternatives arising from specification non-determinism: partial-order signal consumption and belief-based supervision, are described next. Finally the key algorithms of the interpreter are presented along with an analysis of their time and space complexity.

5.1 Overview

The PDM-interpreter interprets the PDM-model. The fundamental difference between the PDM-interpreter and the SDL abstract machine is their operation in the presence of non-determinism. The SDL abstract machine may select any one behavioral alternative arising from specification non-determinism. However, the PDM-interpreter must identify and follow the behavioral alternative chosen by the target system.

The most prominent SDL non-determinism is channel delay. As an example, consider the SDL process and incoming channels shown in figure 5.1. Each of the signals traveling on an SDL channel are first-in-first-out (FIFO) ordered. The contents of the channels are merged into a single input queue associated with the SDL process. Several potential total orders of signals typically exist due to the non-deterministic channel delay.

The PDM-interpreter must determine the total order chosen by the corresponding target system and sequence signal consumption accordingly. A supervisor that arbitrarily sequences signals for consumption would illegally report failures of the target system.

5.1.1 Components of the PDM

The PDM-interpreter is described in terms of its four fundamental components: (1) temporal signal tags, (2) partial-model supervision, (3) belief-based handling of non-determinism and (4) the core-interpreter. The components are described in further detail.

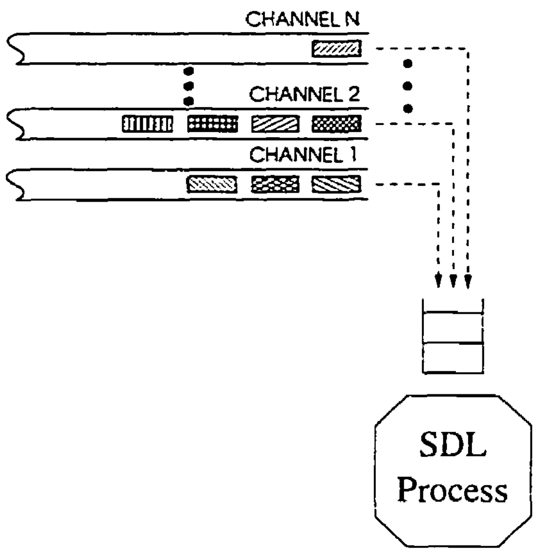


Figure 5.1: Signal Ordering

Temporal Signal Tags Each signal in the supervisor is tagged with its time of generation and/or consumption to facilitate its processing after its occurrence.

Partial-Model Supervision Used to reduce the number of behavioral alternatives needed to be considered by the PDM.

Belief Creation Algorithm Used when the PDM/partial-order signal consumption cannot resolve the behavioral alternative chosen by the target system.

Core Interpreter An out-of-time, directed SDL interpreter.

The remainder of this chapter describes, in further detail, the four components of the PDM-interpreter.

5.2 Temporal Signal Tags

Signals within the supervisor are tagged with the time of generation and/or consumption. This information facilitates their processing after their occurrence. The interpreter is responsible for generating the tags.

Signal tags are analogous to timestamps. However, uncertainty exists as to the actual signal generation/consumption time principally due to a lack of internal target system observability. As a result, signals within the supervisor are tagged with a timestamp ranging over an interval. The interval represents the time during which signals were generated and/or consumed within the target system. Such an interval is referred to as an *occurrence interval* (OI).

OIs are derived based on the time that inputs from and outputs to the environment were generated. Consider the series of state transitions in (5.1). $\Sigma_0, \cdots \Sigma_n$ represent global states of the PDM-model, i a target system input signal, o a target system output signal and $int_1, \cdots int_n$ internally generated and consumed signals.

$$\Sigma_1 \overset{i/int_1}{\longrightarrow} \Sigma_2 \overset{int_1/int_2}{\longrightarrow} \cdots \Sigma_{n-1} \overset{int_n/o}{\longrightarrow} \Sigma_n \tag{5.1}$$

OIs for the state transitions in (5.1) can be derived from the observation times of signals i and o. Assume that signal i was observed at time, t_l and o at time, t_u. t_l and t_u represent the lower and upper bounds of both signal generation and consumption. Thus an OI, $[t_l, t_u]$ represents the consumption time of signal, i, generation and consumption time of signals, $int_1 \cdots int_n$ and the generation time of signal o.

5.2.1 Interpretation of Occurrence Intervals

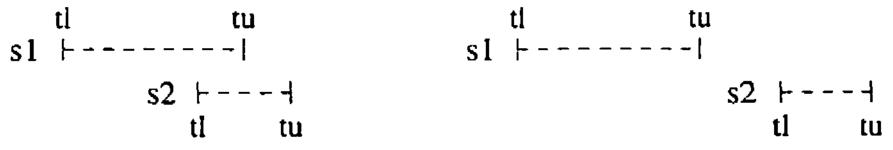
As previously stated, the actual time signals were generated and/or consumed within the target system typically cannot be determined due to a lack of observability. An OI captures the range of time over which a signal was generated and/or consumed.

Within the supervisor, OIs are used to order signals. Consider two signals, $s1$ and $s2$ both with unique OIs. An definite order of the two signals can be determined from their OIs if the OIs do not overlap. Conversely, the order of the two signals cannot be determined solely based on OIs if the OIs of the two signals overlap. The formal definition of OI overlap is defined below. The dot (.) operator is used to address the OI of a signal.

Definition 5.2.1 (Overlapping Occurrence Intervals) *The occurrence intervals of two signals, $s1$ and $s2$ overlap if:*

$$\exists t \ \textit{such that} \ [(t \geq s1.t_l) \wedge (t \leq s1.t_u)] \wedge [(t \geq s2.t_l) \wedge (t \leq s2.t_u)]$$

As an example, overlapping and non-overlapping OIs are illustrated graphically in figures 5.2a and 5.2b respectively.



(a) Overlapping OIs (b) Non-Overlapping OIs

Figure 5.2: Overlapping and Non-Overlapping Occurrence Intervals

The PDM-interpreter orders signals for consumption based on (1) their occurrence intervals and (2) their sequence on the channel/signal route which they traversed. In some cases, the interpreter may not be able to deterministically determine the exact consumption order of signals. The set of signals whose consumption order cannot be determined is referred to as the *consumable signal set*, defined below.

Definition 5.2.2 (Consumable Signal Set (CSS)) *At time t, let s represent a signal from the set of signals appearing at the heads of the incoming signal routes or channels*[1] *having the smallest occurrence interval lower bound. Let J be the set of signals other than s that appear at the heads of the incoming channels/signal routes whose occurrence intervals overlap with s. The* consumable signal set *$(K(t))$ is defined as: $K(t) = J \cup s$.*

Theorem 5.2.1 (Consumable Signals) *The signal to be subsequentally consumed must be contained in the consumable signal set.*

Proof: *Let λ be a signal such that $\lambda \notin K$. From definition 5.2.2, λ either:* (1) *does not appear as a signal at the head of an incoming signal route/channel or* (2) *the occurrence interval of λ does not overlap with s. The two cases are treated independently.*

Case 1: *λ does not appear at the head of a signal route channel. The signal at the head must be consumed before λ. Thus λ cannot be a consumable signal in the current state.*

Case 2: *There are two possible situations in which the occurrence intervals of s and λ do not overlap:* (1) *$\lambda.t_u < s.t_l$ and* (2) *$s.t_u < \lambda.t_l$. The former is not*

[1]In SDL, signal routes carry all signals to processes within a block. However, signals that travel over a channel before reaching their destination shall be referred to as ***traveling over channels***.

possible since s is defined to have the minimum t_l of all signals at the heads of the incoming channels/signal routes and ($t_l \leq t_u$). The latter case verifies that s must be consumed before λ and agrees with Theorem 5.2.1.

5.2.2 Singly Bound Occurrence Intervals

In some cases it may not be possible or desirable to obtain both upper and lower bounds of a signal's OI. For example, to obtain both upper and lower bounds on the OI for input signal, i requires that the time output o is generated be propagated backwards before the input is processed[2]. The backward propagation of event occurrence times adds a significant amount of complexity to the interpreter.

It is possible to determine an OI with only one bound, either the lower or upper. The worst-case target system response time, T^r_{max} is required in such cases. T^r_{max} may be considered an upper limit on the time at which input i will be legitimately serviced by the target system. An event that is serviced after T^r_{max} time units is considered a hard real-time failure.

An OI for the case where the lower bound is not known can be approximated as $[t_u - T^r_{max}, t_u]$. Correspondingly, the OI for the case where the upper OI bound is not known is $[t_l, t_l + T^r_{max}]$[3].

[2]OIs are used by the supervisor to order signals. A signal can not be processed by the supervisor without an OI.

[3]Note that this is an approximation of the actual OI. It is possible that in some cases the supervisor would miss reporting some failures as a result of this. In chapter 7 an empirical evaluation of the number of missed failures based on approximated OIs is presented.

5.2.3 Generation of Signal Tags

For target system input or output signals, the OI of the corresponding signal is computed based on the signal observation time and the worst case target system response time, (T^r_{max}) as described above.

OIs for internally generated signals are derived from the stimulus that caused the state transition in the PDM-model. Recall that an OI is a bound of the generation/consumption times of all signals generated during a sequence of state transitions. As an example, consider the state transition sequence in (5.1). The occurrence interval for signal i includes the time where i was consumed and o generated. Thus it must also include the generation/consumption times of signals, $int_1, int_2, \ldots int_n$. Thus all generated signals inherit the OI of the stimulus causing the state transition in the PDM-model.

5.2.4 Timers

Timers are used to implement delay and timeout facilities in CEFSM-based specifications. Conceptually, timers may be implemented with signal send and receive facilities. As an example, SDL timer set and reset constructs are shown in figure 5.3.

Timers are supervised so that delay and timeout failures can be detected by the supervisor. In an out-of-time supervisor, timers are handled with the aid of the OIs described. The semantics of SDL timers dictate that the setting of a timer (figure 5.3a) creates a signal, which shall be referred to as a timer signal, and places it in the input port of the corresponding process. Timer signals (representing an expired timer) are consumed identically to other SDL signals (figure 5.3b). The resetting of a timer (figure 5.3c) removes and discards an unconsumed timer signal from the input port. The tags of each timer signal influence when it is consumed.

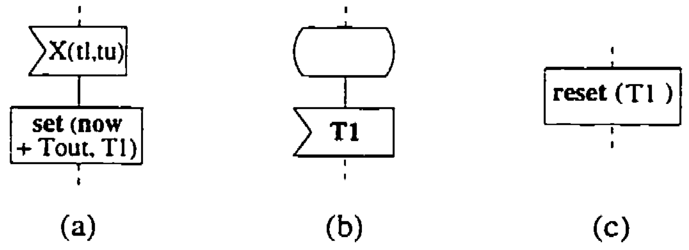


Figure 5.3: SDL Timer Set/Reset Constructs

The occurrence interval of a timer signal is a function of three parameters: (1) the OI of the stimulus that caused the state transition in which the timer was set, (2) the timeout value of the timer and (3) a parameter, Δ, which represents the tolerance of a timers in the target system. The latter of the three parameters implies that within the target system, a timeout will expire after $Tout \pm \Delta$ units of time. For the general timer set operation in figure 5.3a, the OI of the timer signal is set to $[t_l + Tout - \Delta, t_u + Tout + \Delta]$. t_l and t_u represent the OI of the signal that caused the state transition containing the timer set operation (X).

Any number of timers can be set by an SDL process. Each timer signal in the supervisor is sent to the process input port via a separate, delayed channel. Timer channels are not programmer specified but rather implicit, used to conceptualize the ordering of timer signals within the supervisor.

As an example, consider a process, $P1$ that uses two timers, $T1$ and $T2$. One configuration of channels leading to $P1$ is shown in figure 5.4. Unexpired timers are indicated by signals pending consumption. For the example shown, $T1$ is an unexpired timer, while timer $T2$ is not yet set.

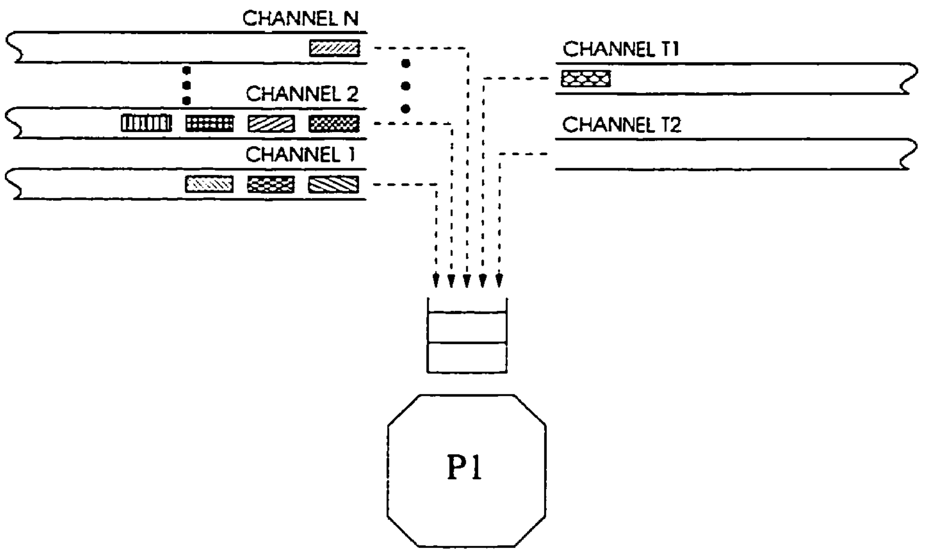


Figure 5.4: Channels Carrying Timer Signals

5.3 Partial-Order Signal Consumption

The objective of partial-order signal consumption [24, 61] is to reduce the number of behavioral alternatives that need to be considered. Recall from section 2.2.2 that behavioral alternatives arising from specification non-determinism can be partitioned into two categories, *don't know* and *don't care*. Partial-order signal consumption addresses don't care non-determinism. Its goal is to eliminate consideration of don't care non-determinism by the supervisor.

5.3.1 Application of Partial Order Signal Consumption

Three common types of SDL non-determinism are addressed in this thesis as outlined in section 2.8. The three types of non-determinism can be sub-divided into directly and indirectly specified.

Directly specified types of non-determinism addressed are spontaneous transitions and non-deterministic decisions. A specification writer must explicitly introduce one of these constructs. For this reason, directly specified types of non-determinism typically fall into the don't know category. In other words, the behavioral alternatives generated rarely lead to identical behavioral alternatives (or the non-deterministic constructs wouldn't have been introduced by the specification writer).

Non-deterministic channel delay is an indirectly specified non-determinism as SDL semantics dictate delayed channel communication must be used in certain situations, beyond the control of the specification writer. In theory, a signal traveling on an SDL channel can be delayed anywhere from $[0, \infty]$ units of time. However, in practice there is typically at least an upper bound placed on communication. From experience, many behavioral alternatives arising from channel delay fall into the don't care category as described in section 2.2.2.

Partial order supervision is thus targeted to reducing the number of don't care non-determinism arising from SDL channel delay for the context of this thesis.

5.3.2 Definitions

Behavioral alternatives arising from SDL channel delay shall be referred to as C-behavioral alternatives. C-behavioral alternatives arise as a result of multiple possible signal consumption orders. The definition of a C-behavioral alternative appears below.

Definition 5.3.1 (C-Behavioral Alternatives) *Let S_P represent the partially-ordered set of signals at the input queue of process P. Let $R_S = \{r_1, r_2, \ldots, r_N\}$ represent the set of N possible total orders of signals from set S_P based on the FIFO* *ordering of SDL channels/signal routes and individual signal OIs. Each member of R_S shall be referred to as a C-behavioral alternative.*

The notion of process behavior and process behavior equivalence is now defined. This will be used to define behavioral alternatives that lead to identical and different observable behavior.

Definition 5.3.2 (Process-Behavior) *Let P' represent a process in the PDM model, in process state ψ with m outgoing channels and signal routes. Let r represent a sequence of signals such that $r \in R_S$.*

$beh_{P'}(\psi, r)$, the behavior of P' after consumption of signal sequence r, is defined as a 2-tuple: $(\overline{\psi}, C)$ where:

- *$\overline{\psi}$ represents the process-state of P' after consumption of signal-sequence r*
- *$C = \{c_1, c_2 \dots c_m\}$ represents the set of signal sequences from the alphabet, $X' \cup \{\epsilon\}$ on the emanating channels and signal routes of P', generated as the signal-sequence r was consumed.*

Two process-behaviors are considered identical if: (1) the generated sequences of signals between the two behavioral alternatives are identical and the OIs of corresponding signals overlap and (2) the final process-states are identical.

C-behavioral alternatives may arise from don't know or don't care non-determinism. Don't know/don't care non-determinism is defined based on the equivalence of the behavior arising from the two C-behavioral alternatives.

Definition 5.3.3 (Don't Care Non-Determinism) *Let $r_1, r_2 \in R_S$ represent two C-behavioral alternatives. r_1 and r_2 are said to be generated under don't care non-determinism if $beh_P(r_1, \psi) \equiv beh_P(r_2, \psi)$ for a given process state ψ of P.*

Definition 5.3.4 (Don't Know Non-Determinism) *Let $r_1, r_2 \in R_S$ represent two C-behavioral alternatives. r_1 and r_2 are said to be generated under don't know non-determinism if $beh_P(r_1, \psi) \not\equiv beh_P(r_2, \psi)$ for a given process state ψ of P.*

From experience, many of the behavioral alternatives arise from don't care non-determinism in a SDL supervisor [39]. The following claim, stated without proof due to its obviousness, is the basis to a substantial reduction of time and/or space complexity in a software supervisor.

Claim 5.3.1 (Partial-Order Signal Consumption) *All legitimate, specified behaviors can be considered by simulating only c-behavioral alternatives generated under don't know non-determinism.*

Not all behavioral alternatives need be considered [36]. Behavioral alternatives arising from don't care non-determinism can be pruned from the search space. One approach for pruning such behavioral alternatives in a supervisor is described in the subsequent section.

5.3.3 An Implementation of Partial-Order Signal Consumption

This section describes an implementation of partial-order signal consumption. There are two principal categories of existing work on this subject, both differing in their target application. The first is with application to verification [28]. However, it is not immediately applicable to supervision since it does not address the real-time aspects of supervision. The second category is focused on automatic generation of test cases for SDL specifications [59]. However, the work does not address the non-determinism associated with SDL channel delay. In other words, it assumes SDL specifications to have a constant SDL channel delay, an assumption not valid for the context of this work.

A spectrum of algorithms to reduce consideration of don't care behavioral alternatives can be envisioned. At one end is a time-intensive algorithm that uses a generate-and-test approach to determine if behavioral alternatives were generated under don't care non-determinism. This is the approach used in belief-based supervision for example. At the other end of the spectrum is a space-intensive approach that uses a look-up table, indexed by the behavioral alternative. The table facilitates a $O(1)$ determination if two behavioral alternatives were generated under don't care non-determinism. However, the approach has an enormous space requirement. Neither of the two approaches are well suited to the problem at hand. A hybrid approach is needed.

In general, there is a tradeoff between the time and/or space complexity of the approach and the reduction in the number of don't care behavioral alternatives considered.

The partial-order approach described capitalizes on the observation that a given signal s in the input port of a process in the PDM-model is permuteable with a finite number of signals. This follows from the discussion of OIs in section 5.2. In addition, many SDL specifications have only a few states in which a signal can be consumed to result in a different behavior.

These two properties are combined into a *redundant permutation distance* (rp-distance). The rp-distance represents the minimum distance, measured in state transitions, before a transition is reached where an SDL signal can be consumed differently than in the current state. The formal definition of rp-distance follows.

Definition 5.3.5 (rp-distance) *Let P' be a process in the PDM model in process-state ψ and A' the input and output alphabet of P'. A'^* thus represents the set of all possible input signal sequences of P'. For signal $s \in A'$, the rp-distance, rp-dist(P', ψ, s) is defined as the minimum length of a signal sequence $\lambda \in A'^*$ such that:*

$$beh_{P'}(\psi, s\lambda) \neq beh_{P'}(\psi, \lambda s)$$

where $s\lambda$ denotes the concatenation of signal s with a signal sequence λ. If no such sequence exists, rp-dist$(P', \psi, s) = \infty$.

Note that rp-distance is not defined for signal-process state pairs where the signal is not consumable (i.e. where the SDL signal is saved).

The rp-distance is enumerated for each process-state and each stimulus in the PDM model. Its significance is that it can be used to reduce redundant signal permutation in the PDM. If the number of signals in a set whose order is not known (i.e. the consumable signal set) is less than the rp-distance of any signal in that set, permutation is redundant.

State/rp-distance pairs are tabulated for each process in the PDM-model. Such a table is called a *partial order distance table* (POD-table) and constitutes the static information used by the partial-order approach.

The rp-distances for all stimuli of the PDM-model fragment in figure 3.3 are shown in table 5.1.

As an example of the derivation of the table consider process state $S0$ and signal X. In the PDM-model, the closest state (in state transitions) where can be consumed with a different behavior than in state $S0$ is $S1$. The distance between $S0$ and $S1$ is one state transition. Thus the rp-distance of signal X in state $S0$ is

Process State	Signal	rp-distance
S0	a	1
S0	b	1
S0	X	1
S0	Y	1
S0	Z	∞
S1	a	2
S1	b	1
S1	X	1
S1	Y	1
S1	Z	∞
S3	a	1
S3	b	2
S3	X	1
S3	Y	1
S3	Z	∞

Table 5.1: Example: Partial Order Distance Table

one. Consider signal Z as a second example. The behavior of the PDM-model will be identical irrespective of the state in which Z is consumed. Thus, a state that results in a *different* behavior does not exist and as a result the rp-distance of Z in any state is ∞.

5.4 Belief Method

Two approaches have been described to deal with some aspects of specification non-determinism thus far. The PDM-model facilitates identification of the behavioral alternative chosen by the target system while partial order signal consumption prunes behavioral alternatives that do not lead to different external behavior. The mechanisms facilitate efficient handling of specification non-determinism. However, they are not able to resolve the behavioral alternative chosen by the target system in all circumstances. The supervisor resorts to the belief method if both of the other approaches fail (i.e. more than one unresolved behavioral alternative exists).

The belief method was discussed in section 3.2.1. It is a conceptually elegant approach for dealing with all types of non-determinism. Thus it is more general than the PDM-model and partial-order signal consumption. However, it has a much larger time and space complexity.

The PDM generates a belief for each unresolvable behavioral alternative. This occurs in two cases. The first case is where the queuing order of two or more signals cannot be determined (figure 5.1a). In this case, a belief is generated for each possible signal queueing order (e.g. for the example shown, A,B and B,A). The second case is where the PDM-model contains an ANY construct, as described in chapter 4 (figure 5.1b). In this case, a separate belief is created for each emanating path from the ANY construct.

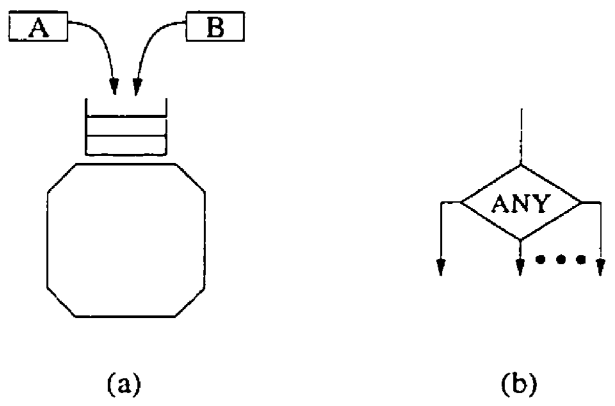


Figure 5.5: Generation of Beliefs

Beliefs are treated as separate threads of execution. They are terminated in one of two cases: (1) if the behavior represented by the belief does not match the externally observed behavior and (2) if n beliefs represent identical global states of the hierarchical supervisor (i.e. PDM and BSup), $n - 1$ of these beliefs are terminated. Note that in the latter case, the supervisor may require processing of the n beliefs for a finite period of time before it can determine that $n - 1$ of the beliefs are redundant.

5.5 Core Interpreter

The PDM interpreter closely resembles the abstract machine of SDL [58]. This section begins with an overview of the relevant portions of the SDL abstract machine. It then describes the key aspects of the PDM abstract machine.

5.5.1 SDL Abstract Machine

The semantics of SDL are formally defined by means of an abstract machine. The SDL abstract machine consists of six types of CSP [27] processes, executing concurrently and communicating synchronously. Figure 5.6 illustrates each of the process types and the communication between them. An overview of the functionality of each SDL process follows. Note that discussion focuses on the supported subset of SDL as outlined in section 2.8.

system: Responsible for creating all other process instances in the abstract machine. It also routes signals between SDL processes.

path: Handles the non-deterministic delay of channels.

timer: Keeps track of current time and handles time-outs.

sdl-process: An SDL interpreter. One instance of this process exists for each SDL process in the specification.

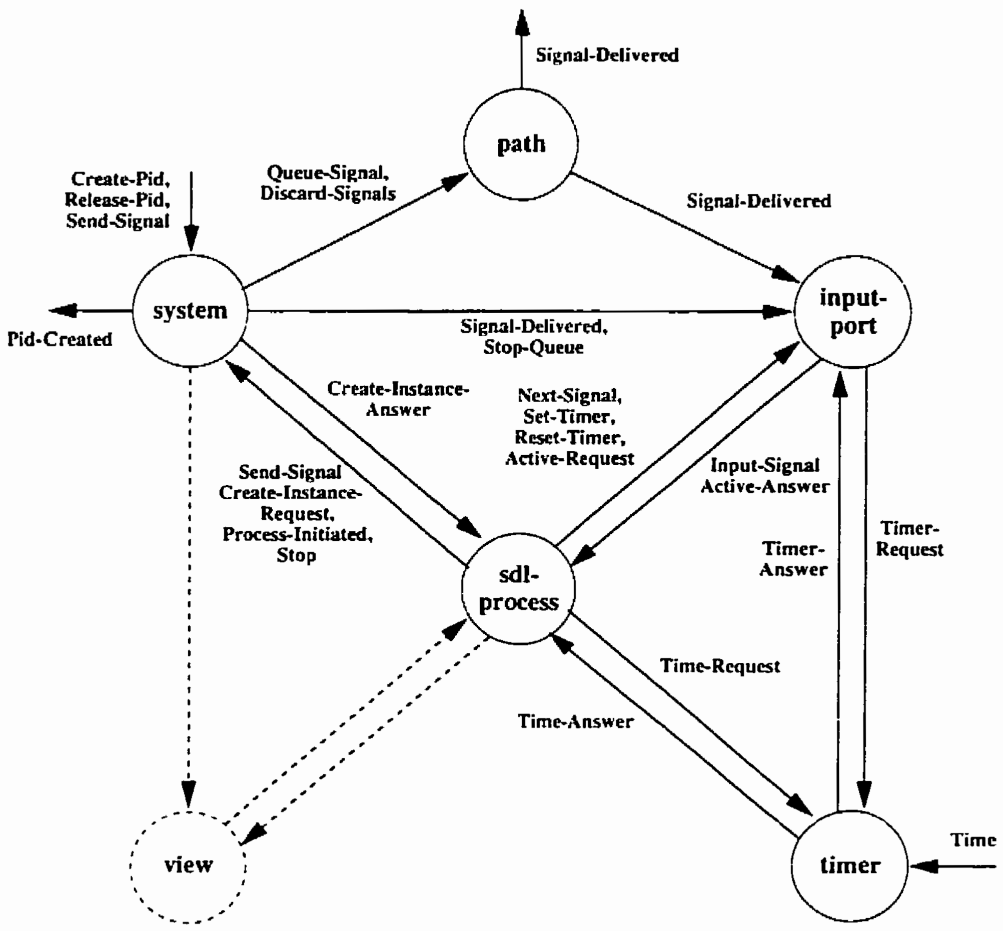


Figure 5.6: SDL Abstract Machine

input-port: Handles the queueing of signals for an SDL process. One instance of input-port exists for each sdl-process instance.

view: Keeps track of all *revealed* variables. Implements communication between sdl-processes by means of shared memory.

This process represents functionality of SDL not addressed in this work and is not discussed further.

5.5.2 PDM Abstract Machine

The PDM abstract machine is similar to the SDL abstract machine described. The difference between the two arises principally from the treatment of specification non-determinism.

The SDL abstract machine may arbitrarily choose a single behavioral alternative from the set of possible alternatives arising from specification non-determinism. The PDM abstract machine is required to identify and choose the behavioral alternative followed by the target system.

The PDM abstract machine differs in three respects from its SDL counterpart. First, as described previously, most behavioral alternatives arise from a number of possible signal permutations at the input port. As a result, the input port of the PDM abstract machine significantly differs from its SDL counterpart. Second, in some cases, the PDM will not be able to resolve the selected behavioral alternative. Beliefs were proposed as a way of dealing with this. Thus the PDM abstract machine must provide support for belief creation, management and termination. Finally, to support out-of-time processing of signals, the PDM abstract machine tags all signals with their OIs.

The process interaction diagram of the PDM abstract machine appears in figure 5.7. A textual summary of each process type in the abstract machine follows. The descriptions highlight the differences between the PDM and SDL abstract machines.

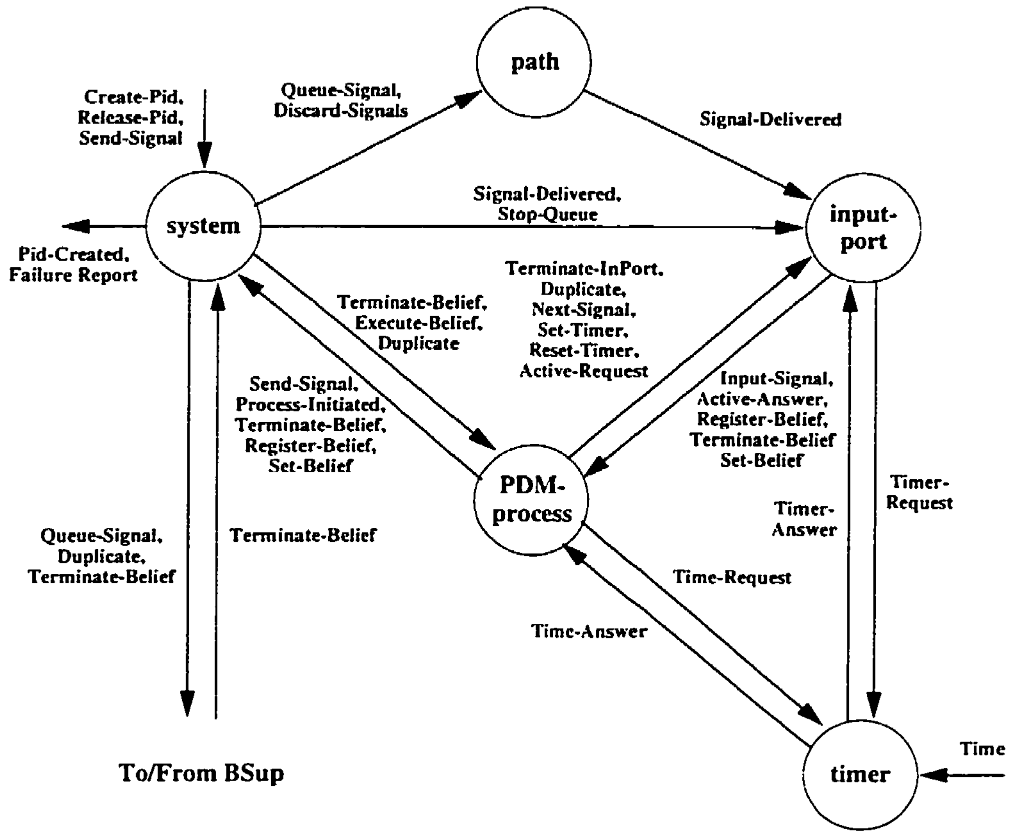

path
Create-Pid, Release-Pid, Send-Signal
Queue-Signal, Discard-Signals
Signal-Delivered
system
Signal-Delivered, Stop-Queue
input-port
Pid-Created, Failure Report
Terminate-InPort, Duplicate, Next-Signal, Set-Timer, Reset-Timer, Active-Request
Terminate-Belief, Execute-Belief, Duplicate
Send-Signal, Process-Initiated, Terminate-Belief, Register-Belief, Set-Belief
Input-Signal, Active-Answer, Register-Belief, Terminate-Belief Set-Belief
PDM-process
Timer-Request
Timer-Answer
Queue-Signal, Duplicate, Terminate-Belief
Terminate-Belief
Time-Request
Time-Answer
To/From BSup
timer
Time

Figure 5.7: Path Detection Module Abstract Machine

system: Creates, manages and terminates beliefs. Timestamps all signals generated by the environment with an occurrence interval. Handles routing of signals from the PDM to the BSup. Note that the PDM abstract machine only supports static SDL process creation.

path: Communicates signals traveling over SDL channels to their appropriate destinations. Note, unlike its SDL counterpart, the *path* process does not output signals to the environment.

timer: Keeps track of current time and handles time-outs.

PDM-process: An SDL interpreter. The only difference between the SDL-process and PDM-process is that all paths are followed by the PDM-process when executing an *ANY* construct by generating one belief for each emanating path.

input-port: Orders signals for consumption according to a corresponding order chosen by the target system when the order can be determined. Creates beliefs when the exact order cannot be determined.

Belief Creation/Termination

As outlined in section 5.4, beliefs are generated in response to unresolvable behavioral alternatives. In the hierarchical supervisor, the PDM is used to resolve the behavioral alternative chosen by the target system. As a result only the PDM creates beliefs. Beliefs are terminated when the external behavior represented by the belief doesn't match the expected behavior from the target system. Thus beliefs may be terminated either by the PDM or BSup.

Within the abstract machine, beliefs may be created by either the PDM-process or the input port as described. The control signals exchanged by the processes when creating beliefs in these two cases are shown in figures 5.8a and 5.8b respectively.

Beliefs may be terminated by either the BSup or input port. The control signals exchanged under these scenarios are shown in figures 5.9a and 5.9b respectively.

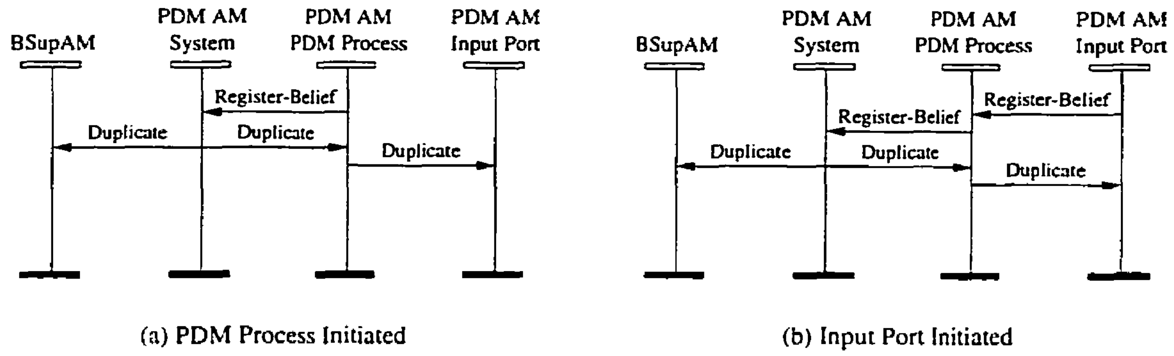


Figure 5.8: Belief Creation

Note that all SDL processes in a belief are terminated. Figures 5.9a and 5.9b show only one process being terminated as others receive identical signals.

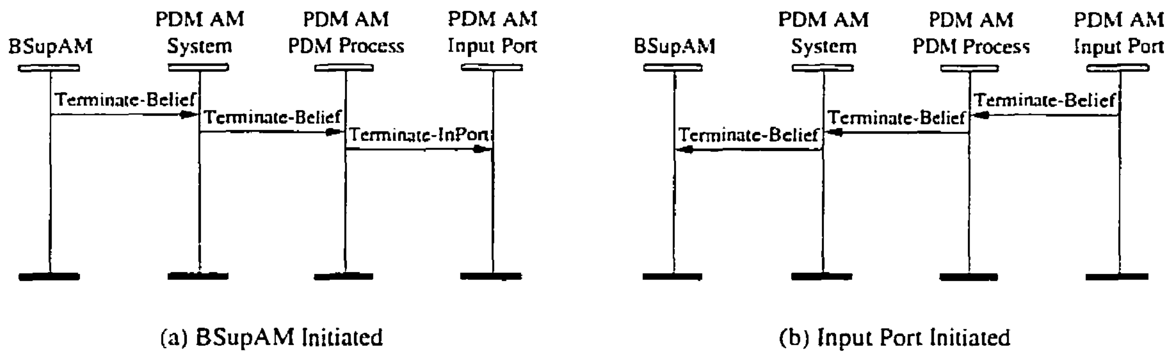


Figure 5.9: Belief Termination

The belief creation/termination facilities were originally described and formalized in [39] to which the reader is referred for further information.

Time Within the PDM

The supervisor processes target system input and output signals out-of-time as described in section 3.4.2. Signals generated at time t may be potentially processed by the PDM anywhere on the interval, $[t + T^r_{max}, \infty]$. Thus the clock of the PDM lags the clock of the target system by at least T^{r}_{max} units of time.

The actual time within the PDM or the value of the PDM's clock is defined in this section. The clock of the PDM is advanced as signals are consumed. Thus the PDM clock is derived from the timestamps of the signals in the input ports of its SDL processes. Individual processes represent executions at different points in time, depending on how the processes are scheduled. Thus the time within the supervisor is defined at two levels: (1) a process level and (2) a global level. The definitions appear below.

Definition 5.5.1 (PDM Process Time) *Let P represent a PDM SDL process, in process state σ, and S the set of signals queued in its input port. K is the consumable signal set and K' a subset of K where all signals in set K' are consumable in the current state (i.e. not in the save set) ($K' \subseteq K \subseteq S$). The process time of P (T_P) is an interval: $T_P = [T_{P_l}, T_{P_u}]$ where:*

- T_{P_l} *= minimum occurrence interval lower bound of a signal,* $s_1 \in K'$
- T_{P_h} *= maximum occurrence interval upper bound of a signal,* $s_2 \in K'$

For a process P if set K' is empty its process time is undefined.

The process time ranges over an interval due to the uncertainty of the actual generation/consumption time of signals in the target system. The process time is undefined for processes with zero signals in their consumable unsaved signal sets (CUSSs). Process times are consolidated into a PDM global time, defined below.

Definition 5.5.2 (PDM Global Time) *For a set of SDL processes, G, the global time of G ($T_{G_{PDM}}$) is an interval, $T_{G_{PDM}} = [T_{G_l}, T_{G_u}]$ where:*

- T_{G_l} = *minimum occurrence interval lower bound of a process,* $P_1 \in G$ *having a defined process time*
- T_{G_h} = *maximum occurrence interval upper bound of a process,* $P_2 \in G$ *having a defined process time*

If all processes, $P \in G$ *have undefined process times, the global time is undefined as well.*

5.5.3 PDM Input Port

The central part of the PDM abstract machine is the input port. The input port orders signals for consumption. It takes into consideration the FIFO constraints imposed by the channels and signal routes of the specification as well as the signal occurrence intervals. Thus for n signals, only a subset of the $n!$ signal orders typically need be considered.

The core of the input port is a sorting algorithm that orders signals in the consumable signal set (CSS), defined in section 5.2.1. Signals in the save set are removed from the CSS to form the consumable unsaved signal set (CUSS). Signals in the CUSS are candidate signals to be consumed in the current state.

The input port is described as two parts. The first part (*Algorithm QueueSignal()*) deals with the queueing of signals and preservation of the FIFO signal orders imposed by channels and signal routes. The second part (*Algorithm ConsumeSignal()*) outputs signals to the corresponding SDL process for consumption. The discussion begins with a description of the major type and datastructure definitions followed by the actual algorithms.

Type Definitions

signal_names symbolic signal names consumed/generated by SDL process P

OI the occurrence interval type as defined in section 5.2

signal represents an SDL signal. **signal** has the following sub-fields:

> **name** symbolic signal name of type **signal_name**
>
> **sender** pid of sender process
>
> **receiver** pid of receiver process
>
> **origin** source of the signal (i.e. PDM/BSup/environment)
>
> **OI** occurrence interval
>
> **parameters** associated signal parameters as defined in the PDM-model

PS the process states of P

time an interval ranging over a period of time

Datastructures

Datastructures principally store incoming signals to the signal routes/channels. It is assumed that an SDL process has n incoming channels and/or signal routes.

c_i a sequence of elements of type **signal**. c_i represents the sequence of signals on an incoming channel or signal route i $(1 \leq i \leq n)$. As signals are consumed they are removed from the head of c_i. Note that c_i may be empty.

C a set of sequences of type **signal**. $C = \{c_1, c_2, \ldots c_m\}$

J a sequence of signals. Contains a copy of the signals at the heads of the incoming signal channels/signal routes. J is kept sorted based on the lower bound of each signal's occurrence interval.

$PODT(\sigma)(s)$ partial order distance table, an array indexed by the current process state (σ) and signal (s), returning the partial-order distance.

T_{PDM} Global time of the PDM

$curr\_belief$ a pointer to the current belief of the process

Operations

$comm\_path\_id(s : \textbf{signal})$ accepts as input a signal s and returns c_i, the incoming communication path traversed by s where $c_i \in C$.

$x \frown y$ sequence concatenation. x and y represent sequences. The function returns the concatenation of x and y.

Queue Signal Algorithm

The queue signal algorithm is responsible for queueing signals in a datastructure that preserves the FIFO order of SDL channels/signal routes (c_i). It also updates J, a copy of the signals at the heads of the incoming channels/signal routes.

Consume Signal Algorithm

The consume signal algorithm orders signals for consumption. It implements partial-order signal consumption and creates beliefs when uncertainty exists as to the actual ordering of signals.

```
Algorithm QueueSignal(C : signal_sequence_set, J : signal_sequence, s : signal)
1. $c_i$ = comm_path_id(s)
2. if ($c_i$ == empty)
3.     insert s, into J based on the lower bound of each signal's OI
4. end if
5. append s to the tail of $c_i$
6. return(C, J)
end Algorithm
```

Figure 5.10: Input Port Queue Signal Algorithm

Due to the complexity of the algorithm, a flowchart of the algorithm appears in figure 5.11. The actual algorithm appears in figure 5.12. A textual summary of the algorithm follows.

Consume Signal Algorithm Description

lines 1-2 Construct the consumable signal set (see definition 5.2.2)

lines 3-6 Check if the global time has advanced past the process time. If so, no signal ever will arrive allowing the signals in the input port to be consumed. The current belief is terminated.

line 7 The consumable unsaved signal set (CUSS) is generated for the current process state.

lines 8-16 A check is made if the partial-order distance of each unsaved consumable signal is greater than the total number of unsaved consumable signals If so the order of signal consumption is arbitrary and *flag* is set false to indicate this.

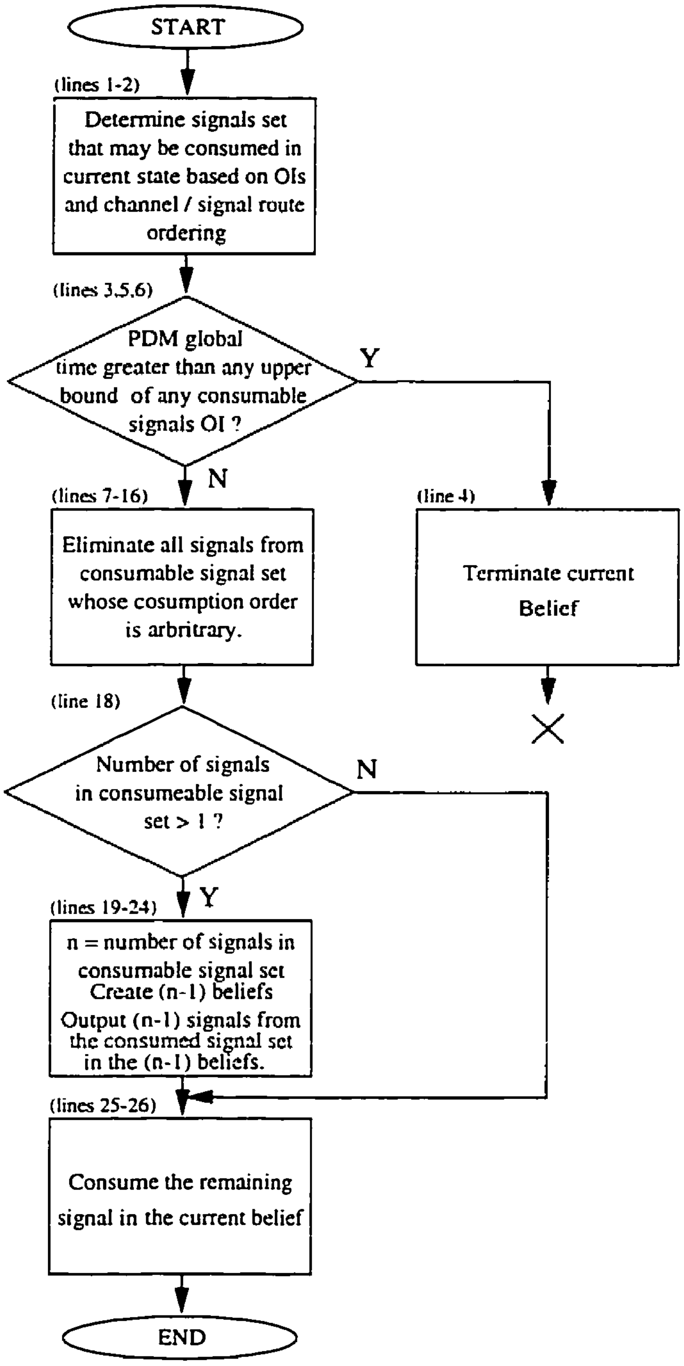

START
(lines 1-2)
Determine signals set that may be consumed in current state based on OIs and channel / signal route ordering
(lines 3,5,6)
PDM global time greater than any upper bound of any consumable signals OI ?
Y
N
(lines 7-16)
Eliminate all signals from consumable signal set whose cosumption order is arbritrary.
(line 4)
Terminate current Belief
(line 18)
Number of signals in consumeable signal set > 1 ?
N
Y
(lines 19-24)
n = number of signals in consumable signal set
Create (n-1) beliefs
Output (n-1) signals from the consumed signal set in the (n-1) beliefs.
(lines 25-26)
Consume the remaining signal in the current belief
END

Figure 5.11: Consume Signal Algorithm

```
Algorithm ConsumeSignal(C : signal_sequence_set, J : signal_sequence,
        σ : process_state, $T_{G_{PDM}}$ : time)
1.  a = head(J)
2.  construct $J_l$ and $J_h$ such that $J = J_l \frown J_h$. The OIs of all signals in $J_l$ overlap
        with a and the OIs of all signals in $J_h$ do not overlap with the OI of a
3.  if ($T_{G_{PDM}}$ does not overlap with the OI of any signal in $J_l$)
4.      output(Terminate-Belief)
5.      exit Algorithm
6.  end if
7.  let $J_l'$ represent the unsaved signals for process state, σ in $J_l$ ordered as in $J_l$
8.  let N = number of elements in $J_l'$
9.  flag = false
10. for each v of type signal_name
11.     if (a at least one signal (s) of type v exists in $J_l'$)
12.         if ($PODT(\sigma)(s) < N$)
13.             flag = true;
14.         end if
15.     end if
16. end for
17. cb = curr_belief
18. if (flag == true)
19.     for (index = 2 to N)
20.         output(Register-Belief)
21.         output(Send-Signal($J_l'(index)$))
22.         (C, J) = DeQueueSignal(C, J, $J_l'(index)$)
23.         output(Set-Belief(cb))
24.     end for
24. end if
25. output(Send-Signal($J_l'(1)$))
26. (C, J) = DeQueueSignal(C, J, $J_l'(1)$)
27. return(C, J)
end Algorithm

Algorithm DeQueueSignal(C : signal_sequence_set, J : signal_sequence, s : signal)
1.  remove signal s from J
2.  c = comm_path_id(s)
3.  delete_head(c)
4.  if (c ≠ empty)
5.      x = head(c)
6.      insert x, sorted into J based on the lower bound of each signal's OI
7.  end if
8.  return (C, J)
end Algorithm
```

Figure 5.12: PDM Input Port Signal Consumption Algorithm

lines 18-24 If *flag* is true, a separate belief is created for each possible signal consumption order.

lines 25-26 The final signal is consumed in the current belief.

DeQueue Signal Algorithm Description

lines 1-3 the signal to be consumed is removed from the internal signal list (J) and from the incoming channel/signal route

lines 4-7 if the channel/signal route carrying the signal to be consumed is not empty, the subsequent signal is added to the consumable signal list.

5.5.4 Complexity Analysis

The analysis evaluates the asymptotic time and space complexity of the major algorithms associated with the input port. A definition of the notation used in the analysis is presented first. The analysis omits discussion of portions of algorithms whose complexity is $O(1)$.

c: the maximum number of incoming channels and signal routes to a SDL process (i.e. fan-in)

t: the number of signal types consumed/generated by the SDL process (i.e. the cardinality of **signal_name**)

B: the number of beliefs generated. An analytical expression will not be presented for B as it is highly application-specific. However, B is a function of the specification, the PDM-model and the operational profile.

N: maximum number of signals queued within the PDM. N is principally a function of the load on the target system and the SDL specification.

Queue Signal Algorithm

The essential function of the queue signal algorithm is to insert signals into a sorted list (J). The list contains at most one signal from each channel/signal route (c). The algorithm is called for each signal queued (N) and is executed once per belief (B). As a result, the worst case running time complexity of the algorithm is given by 5.2.

$$T_{PDM\_IP\_QueueSig}(B, N, c) = O(B \cdot N \cdot \log c) \tag{5.2}$$

DeQueue Signal Algorithm

The running-time complexity of all lines in the DeQueue Signal algorithm are $O(1)$ except for line 6 which performs an insert into a sorted list. Thus the algorithm's running-time complexity is given by 5.3.

$$T_{PDM\_IP\_DeQueueSig}(B, N, c) = O(B \cdot N \cdot \log c) \tag{5.3}$$

Consume Signal Algorithm

line 2 a linear search and copy examines each element in J. The worst case size of J is c elements resulting in a running time complexity of $O(c)$

lines 3,7 linear search of all elements in J_l. The maximum size of J_l is identical to J, resulting in a running-time complexity of $O(c)$.

lines 10-16 a search of J_l is performed t times. Thus the running time complexity of the outer loop is $O(t \cdot \log c)$

lines 19-26 For each belief generated, a call is made to the DeQueue algorithm. A maximum one belief per element in J_l is created. Thus the running time complexity becomes $O(c \cdot logc)$

The running time complexity of the consume signal algorithm is dominated by lines 10-16 and 19-26. The algorithm is repeated for each signal consumed (N) and for each belief generated (B). Thus, the overall running-time complexity is given by 5.4.

$$T_{PDM\_IP\_ConsSig}(B, N, t, c) = O((B \cdot N \cdot (t + c) \cdot \log c) \tag{5.4}$$

5.5.5 Computational Complexity of the Input Port

The computational complexity of the input port is dominated by the consume signal algorithm. This makes intuitive sense since this is the most sophisticated of all algorithms. Thus the complexity of the input port is given by 5.5.

$$T_{PDM\_IP}(B, N, t, c) = O((B \cdot N \cdot (t + c) \cdot \log c) \tag{5.5}$$

5.5.6 Scheduling Process Execution within the PDM

SDL processes in an SDL specification execute concurrently [62]. For a given specification, many execution interleavings exist. Two or more execution interleavings may result in different observable behavior. The supervisor, as described, considers the execution of SDL processes sequentially.

Scheduling of SDL processes within the supervisor must therefore take into consideration that alternate scheduling orders may result in different externally observable behaviors. Behavioral alternatives arising from different process execution orders may be classified and dealt with similarly to behavioral alternatives arising from signal consumption orders. Don't know alternatives result in different externally observable behavior while don't care alternatives do not. Beliefs need be created to consider don't know process scheduling orders.

From experience, the majority of SDL process scheduling alternatives do not result in different externally observable behavior. The intuitive explanation behind this is that the consideration of alternate scheduling orders adds to the complexity of the specification. This makes the specification more difficult to understand and impedes its central purpose: unambiguity and understandability to all parties involved with the software development effort, from the customer to software developers and testers.

The implication of incorrectly scheduling the execution of processes within the PDM is that the hierarchical supervisor will generate false failure reports. An analysis was done on the class of systems described. It was determined that the scheduling order can be approximated by scheduling processes based on their process times. A total scheduling order can be imposed if the process times do not overlap. Processes with overlapping process times are ordered heuristically.

Processes that are about to consume signals from the environment are executed before processes that are about to consume internally generated signals, such that all internally generated signals are generated before processing begins. The scheduling algorithm appears in the subsequent section.

PDM Process Scheduling Algorithm

Scheduling of processes within the PDM is done by the system process in the PDM abstract machine. The scheduling algorithm maintains a list of SDL processes ready to execute (i.e. SDL processes having at least one unsaved signal in their consumable signal set). The first process P on the scheduling list is removed and executed provided that the upper bound of P's process time (t_u) is greater than the current time (i.e. $t_u < T_c$) (see section 3.4.2).

The scheduling algorithm accepts the parameters defined below as input. It returns an updated scheduling list, L. The algorithm appears in figure 5.13.

P : process to be scheduled

L : ordered process scheduling list

```
Algorithm ScheduleProcess(P : process, L : Scheduling List)
1. let X represent a 3-tuple: $< P, P.t_l, P.t_u >$
2. insert X into L sorted in ascending order based on $t_l$
3. re-sort L such that 3-tuples with overlapping OIs having signals from their environment
       in their input queue appear before processes having internal signals in their input que
4. return(L)
end Algorithm
```

Figure 5.13: PDM Process Scheduling Algorithm

Running-Time Complexity

Let N_P represent the number of SDL processes in the PDM-model, N the number of queued signals within the PDM and B the number of maximum beliefs generated.

The maximum size of L is N_P. A sorted insertion into L has a running-time complexity of $O(\log N_P)$. The scheduling algorithm is executed after each signal consumption. Scheduling is re-computed for each belief independently. Thus the overall running-time complexity of the algorithm is given by 5.6.

$$T_{PDM\_Sched}(B, N, N_P) = O(B \cdot N \cdot \log N_P) \tag{5.6}$$

5.6 Time and Space Complexity of the PDM

The complexity analyzed based on the dominant algorithms described (i.e. *Consume Signal* and *Schedule Process*). The time and space complexities are presented individually.

5.6.1 Running-Time Complexity

For each signal within the PDM, internalally or externally generated, the *Consume Signal* and *Schedule Processes* are invoked. The algorithms are invoked once per belief. Based on equations 5.5 and 5.6, the overall running time complexity of the PDM is given by 5.7.

$$T_{PDM}(B, N, t, c, N_P) = O(B \cdot N \cdot ((t + c) \cdot \log c + \log N_P)) \tag{5.7}$$

5.6.2 Space Complexity

The space complexity of the supervisor is largely dependent upon the number of beliefs generated (B). Each belief makes a duplicate of each signal(N), the state of each process P_s and the scheduling list N_P. Thus for a specification of size S, the space complexity of the PDM is given by 5.8.

$$R_{PDM}(B, N, N_P, S) = O(B \cdot (N + P_s \cdot N_P + N_P) + S) \tag{5.8}$$

Chapter 6

The Base Supervisor

This chapter describes the base supervisor (BSup). Like the PDM, the base supervisor consists of a BSup-model, obtained from the requirements specification by transformation, and a BSup-interpreter.

The section begins with a discussion of the BSup-model transformation process. A high-level overview of the BSup interpreter is presented next, followed by a discussion of time within the BSup. The BSup interpreter is then described in detail. Major algorithms and their time/space complexities are presented.

6.1 The Base Supervisor Model

As described in chapters 4 and 5, the objective of the PDM is to steer the execution of the BSup. Unlike the PDM, the BSup makes use of an almost-unaltered requirements specification.

The PDM steers BSup execution by specifying the signal consumption order that would lead the BSup along the determined path. In two cases, the path chosen by the BSup does not depend on the signal consumed. Execution of the SDL *ANY* or *NONE* constructs directs a SDL specification along a non-deterministically chosen path.

Two transformations are used to allow the PDM to steer the BSup in both of these cases. The transformations appear in figure 6.1. The *ANY* construct is replaced by a state transition for each emanating path (figure 6.1a). Signals causing the state transitions (*ANY_P1*, *ANY_P2* ··· *ANY_Pn*) are generated solely by the PDM and are not matched with a signal generated within the BSup. Similarly, spontaneous transitions are replaced with signal transitions initiated by the PDM (figure 6.1b).

Directives that are not matched shall be referred to as *non-matchable directives*.

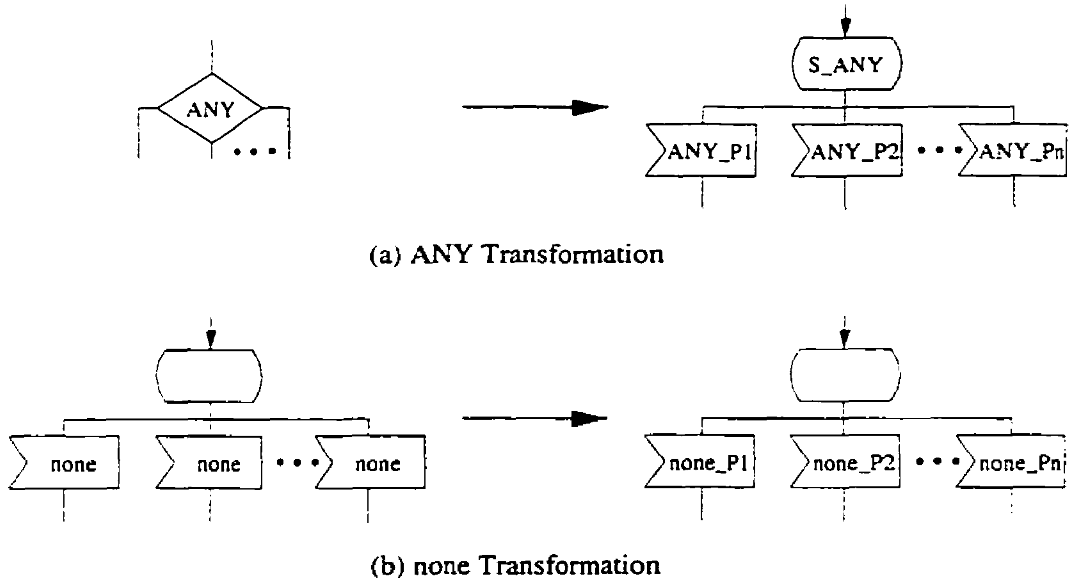


Figure 6.1: Base Supervisor Model Transformations

6.2 Base Supervisor Interpreter Overview

The BSup abstract machine is similar to the SDL abstract machine. However, differences arise from the difference in purpose of the BSup abstract machine, that is detailed behavior checking. The major differences between the SDL and BSup abstract machines are outlined below.

Time: The BSup abstract machine is an out-of-time SDL interpreter. All signals are tagged with OIs as in the PDM. OIs are used to order signals for consumption.

Belief Processing: The BSup includes facilities for belief generation, management and termination. Beliefs are created under the direction of the PDM, however they may be terminated from within the BSup (as well as within the PDM).

Comparator: With reference to figure 3.4, the comparator implements the expected/observed behavior buffers and the matcher functionality. Its function is to compare expected and observed signals generated by the BSup and target system respectively and terminate the currently executing belief if a match cannot be made.

Path-Direction: Signals to be consumed by a BSup process must match with path-directives generated by the PDM. The BSup interpreter includes facilities for matching path directives from the PDM with signals in the BSup queued for consumption.

The remainder of this chapter describes the BSup abstract machine. The discussion begins with a description of time in the supervisor. A process-level description of the BSup abstract machine is presented next. The major algorithms of the abstract machine are subsequentally presented along with their associated time and space complexities.

6.3 Time within the Base Supervisor

The BSup makes use of signal occurrence intervals, similar to the PDM. Each signal is tagged with an OI, ranging over an interval, that represents when the signal was generated and/or consumed. OIs within the BSup are derived in identically as in the PDM (discussed in section 5.2). SDL timers are implemented with the aid of OIs and operate as described in section 5.2.4.

The notions of process and global time are defined for the BSup as done for the PDM. The BSup-specific versions of the definitions follow.

Definition 6.3.1 (BSup Process Time) *Let P represent a BSup SDL process in process state σ, and S the set of signals queued in the input port of P. K is the consumable signal set and K' a subset of K where all signals in set K' are consumable in the current state (i.e. not in the SDL save set) ($K' \subseteq K \subseteq S$). The process time of P (T_P) is an interval: $T_P = [T_{P_l}, T_{P_u}]$ where:*

- T_{P_l} *= minimum occurrence interval lower bound of a signal, $s_1 \in K'$*
- T_{P_h} *= maximum occurrence interval upper bound of a signal, $s_2 \in K'$*

For a process P if set K' is empty its process time is undefined.

Unlike the PDM, a BSup input port must process signals generated within the BSup (or from the environment) in addition to signals generated by the PDM. Note that process time is not influenced by the path-directives generated by the PDM.

Definition 6.3.2 (BSup Global Time) *For a set of SDL processes, G, the global time of G ($T_{G_{BSup}}$) is defined over a range $T_G = [T_{G_l}, T_{G_u}]$ where:*

- *T_{G_l} = minimum occurrence interval lower bound of a process, $P_1 \in G$ having a defined process time*
- *T_{G_h} = maximum occurrence interval upper bound of a process, $P_2 \in G$ having a defined process time*

If all processes, $P \in G$ have undefined process times, the global time is undefined as well.

6.4 Behavior Supervisor Interpreter

The BSup interpreter is specified as an abstract machine, based on the SDL abstract machine (figure 5.6). The BSup interpreter, as described, does not implement the functionality associated with the *view* process. The *comparator* process is introduced to match expected and observed signals (figure 3.4). The BSup abstract machine process interaction diagram is shown in figure 6.2. A brief description of the functionality of each process follows.

system: Creates manages and terminates beliefs. Tags all signals generated by the environment with an OI. Note that the BSup abstract machine only supports static SDL process creation.

path: Stamps all signals with the traversed channel ID. Does not delay signals like its SDL abstract machine counterpart.

timer: Keeps track of current time and handles time-outs.

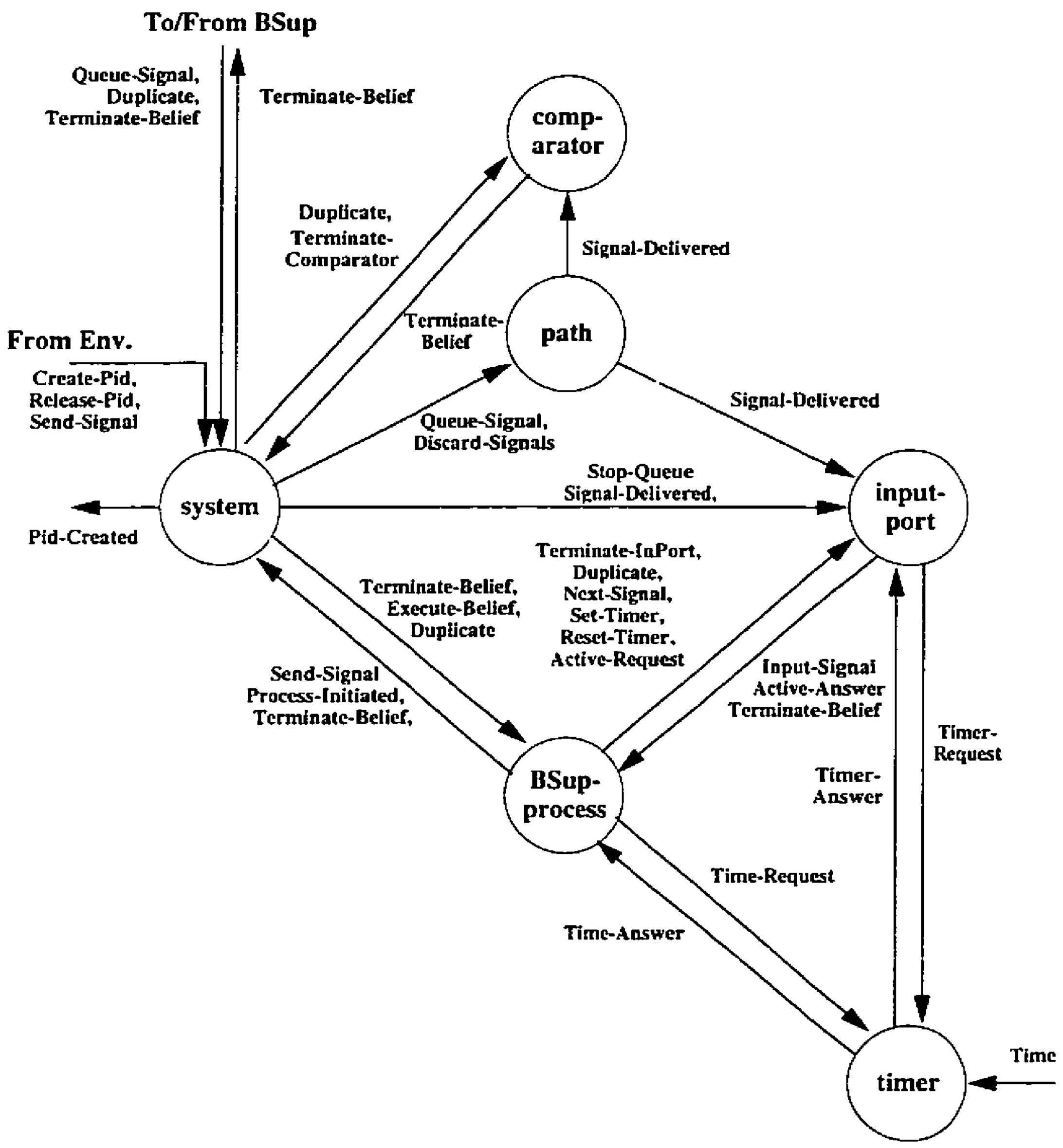

To/From BSup
Queue-Signal, Duplicate, Terminate-Belief
Terminate-Belief
comp-arator
Duplicate, Terminate-Comparator
Signal-Delivered
Terminate-Belief
path
From Env.
Create-Pid, Release-Pid, Send-Signal
Signal-Delivered
Queue-Signal, Discard-Signals
Stop-Queue Signal-Delivered,
system
input-port
Pid-Created
Terminate-InPort, Duplicate, Next-Signal, Set-Timer, Reset-Timer, Active-Request
Terminate-Belief, Execute-Belief, Duplicate
Send-Signal Process-Initiated, Terminate-Belief,
Input-Signal Active-Answer Terminate-Belief
Timer-Request
Timer-Answer
BSup-process
Time-Request
Time-Answer
timer
Time

Figure 6.2: Base Supervisor Abstract Machine

BSup-process: An SDL interpreter. Identical to the SDL abstract machine with one exception: no support is provided for execution of *ANY* or *none* constructs (see section 6.1).

input-port: Maintains two groups of signals: (1) signals generated by the environment and internally within the BSup and (2) signals generated by the PDM. Orders signals for consumption according to order prescribed by the PDM.

comparator: Queues signals for matching. Terminates the current belief if a match between expected and observed signals cannot be made.

6.4.1 Belief Creation/Termination

As indicated previously, belief creation is initiated only by the PDM. A belief created by the PDM requires the BSup to create a matching belief. Both the PDM and BSup process the same belief at all times. Beliefs may be terminated by either the PDM or BSup. If, for example, a belief is terminated by the BSup, the corresponding belief must be terminated within the PDM and vice-versa.

Within the BSup, beliefs are terminated by either the input-port or comparator. The input-port terminates beliefs in one of two cases. First, if the path prescribed by the PDM cannot be followed due to missing signals (i.e. a path directive cannot be matched with any signal in the BSup). Second, if spurious signals have been generated by the environment that do not correspond with the path prescribed by the PDM (i.e. a signal in the BSup cannot be matched with any path directive). The comparator terminates beliefs if a match cannot be made between the contents of the expected/observed behavior queues.

The belief generation/termination protocol used in the BSup is identical to that used in the PDM. As an example, the reader is referred to figures 5.8 and 5.9.

The following sections describe the novel aspects of the BSup interpreter. The discussion begins with the comparator followed by the BSup input port. The discussion concludes with a commentary on BSup process scheduling.

6.5 Comparator

The functions of the comparator are: (1) to store signals in a pair of observed / expected behavior queues and (2) to compare the contents of the two queues. If a match of the contents of the two queues can be made, the contents are annihilated. If a match cannot be made, the current belief is terminated.

The comparator is presented as two algorithms. The *QueueSignal* algorithm determines the source of signals and queues them in either the expected or observed behavior queue. The *ProcessContents* algorithm matches signal contents of the two queues. The two algorithms appear in figures 6.3 and 6.4. A description of the major datastructures used by the algorithms follows.

OBQ: Observed behavior queue. A queue of elements of type **signal**.

EBQ: Expected behavior queue. A queue of elements of type **signal**.

T_{BSup}: global time of the BSup.

6.5.1 Queue Signal Algorithm

The *QueueSignal* algorithm accepts as input the *EBQ*, *OBQ* and a signal to be queued. It returns the new *EBQ* and *OBQ* after the signal has been queued. The algorithm appears in figure 6.3.

```
Algorithm QueueSignal(EBQ : signal_queue, OBQ : signal_queue, s : signal)
1. if (source(s) == environment)
2.     append s to OBQ
3. else
4.     append s to EBQ
5. end if
6. return (EBQ, OBQ)
end Algorithm
```

Figure 6.3: Comparator Queue Signal Algorithm

6.5.2 Process Contents Algorithm

The process contents algorithm compares the contents of the expected/observed behavior queues. It accepts the *EBQ*, *OBQ* and the global time of the BSup as input. It returns the new *EBQ* and *OBQ*. The algorithm appears in figure 6.4. A textual summary of the algorithm follows.

line 1 the algorithm attempts to match the entire contents of the EBQ/OBQ

lines 4-6 matching signals in the EBQ/OBQ are annihilated

lines 7-9 if a match cannot be made the current belief is terminated

lines 12-16 if the EBQ is not empty and the global time of the BSup has advanced past the OI of the signal at the head of the EBQ, the current signal will never be matched. The current belief is terminated.

lines 17-23 if the OBQ is not empty and the global time of the BSup has advanced past the OI of the signal at the head of the OBQ, the current signal will never be matched. The current belief is terminated.

```
Algorithm ProcessContents(EBQ : signal_queue, OBQ : signal_queue, $T_{G_{BSup}}$ : time)
1.  while( (EBQ $\neq$ empty) and (OBQ $\neq$ empty) )
2.      x = head(EBQ)
3.      y = head(OBQ)
4.      if ( (x.name = y.name) and (x.OI overlaps with y.OI) )
5.          delete( head(EBQ) )
6.          delete( head(OBQ) )
7.      else
8.          output (Terminate-Belief)
9.          exit Algorithm
10.     end if
11. end while
12. if ( EBQ $\neq$ empty )
13.     if ( head(EBQ).OI.$t_u$ < $T_{G_{BSup}}.t_l$ )
14.         output (Terminate-Belief)
15.         exit Algorithm
16.     end if
17. end if
18. if ( OBQ $\neq$ empty )
19.     if ( head(OBQ).OI.$t_u$ < $T_{G_{BSup}}.t_l$ )
20.         output (Terminate-Belief)
21.         exit Algorithm
22.     end if
23. end if
24. return(EBQ, OBQ)
end Algorithm
```

Figure 6.4: Comparator Signal Matching Algorithm

6.5.3 Complexity Analysis

The asymptotic running-time complexity is presented for the algorithms comprising the comparator. The notation used is be consistent with that introduced in chapter 5.

Queue Signal Algorithm

The queue signal algorithm simply appends a signal to the appropriate queue. Its running time complexity is $O(1)$. It is invoked once for each signal to be queued. Signals are re-queued individually in each belief. For a worst case of N signals queued, and a maximum of B beliefs, the running-time complexity of the algorithm is given by 6.1.

$$T_{BSup\_Comp\_QueueSig}(B, N) = O(B \cdot N) \tag{6.1}$$

Process Contents Algorithm

The *ProcessContents* algorithm compares the contents of the two queues. For a worst-case queue length of N, its running time complexity is $O(N)$. Queues are duplicated in each belief and thus the running-time complexity of the algorithm is given by 6.2.

$$T_{BSup\_Comp\_ProcCont}(B, N) = O(B \cdot N) \tag{6.2}$$

6.6 Input Port

The input port is specified as a collection of three algorithms: *QueueSignal*, *Annihilate* and *ConsumeSignal*. The *QueueSignal* algorithm queues both BSup signals and PDM path directives in the input port. *Annihilate* deletes a matching BSup signal and PDM path directive from the input port and *ConsumeSignal* performs the matching between path directives and BSup signals, keeping track of the PDM/BSup global times.

The majority of type and datastructure definitions used by the algorithms are consistent with those used to specify the PDM input port and appear in section 5.5.3. In addition, the BSup input port must queue path directives from the PDM and thus it requires an appropriate datastructure, defined below.

PDMQ a queue of path directives of type **signal** from the PDM

6.6.1 Queue Signal Algorithm

The *QueueSignal* algorithm accepts as input the set of signal sequences corresponding to the incoming channels/signal routes (C), the $PDMQ$, a sorted list of signals at the heads of the incoming channels/signal routes (J) and the signal to be queued (s). It returns the updated datastructures C, $PDMQ$ and J.

6.6.2 Consume Signal Algorithm

The BSup *ConsumeSignal* algorithm has a similar function to the PDM *ConsumeSignal* algorithm described in section 5.5.3. Unlike its PDM counterpart, the BSup

```
Algorithm QueueSignal(C : signal_sequence_set, PDMQ : signal_queue,
        J : signal_sequence, s : signal)
1.  if ( s.origin = PDM )
2.      insert s at the tail of PDMQ
3.  else
4.      $c_i$ = comm_path_id(s)
5.      if ($c_i$ = empty)
6.          insert s, sorted into J based on the lower bound of each signal's OI
7.      end if
8.      append s to the tail of $c_i$
9.  end if
10. return(C, PDMQ, J)
end Algorithm
```

Figure 6.5: Input Port Queue Signal Algorithm

algorithm matches signals in the input port with the directives from the PDM. The effect is that execution is steered along the PDM-specified path.

The algorithm accepts as input queued signals on the incoming channels/signal routes (C), the sequence of signals at the heads of the channels/signal routes (J), the $PDMQ$, the current process state of the associated BSup-process (σ) and the global times of both the PDM and BSup (T_{PDM} and T_{BSup}). It returns the updated datastructures C, J and $PDMQ$ and outputs a signal to the corresponding process for consumption.

The algorithm appears in figure 6.6. A textual summary of the algorithm follows.

Consume Signal Algorithm

line 1 the algorithm attempts to match all unsaved path directives from the PDM with signals within the BSup

```
Algorithm ConsumeSignal(C : signal_sequence_set, J : signal_sequence,
        PDMQ : signal_queue, $\sigma$ : process_state, $T_{PDM}$ : time, $T_{BSup}$ : time)
1.  while( (PDMQ $\neq$ empty) and (J $\neq$ empty)
2.      x = head(PDMQ)
3.      if ( x is a non-matching path directive )
4.          output( Send-Signal(x) )
5.      else if ( fields of x match with fields of a signal, s $\in$ J
            and OIs of x and s overlap)
6.          output( Send-Signal(x) )
7.          (C, J, PDMQ) = Annihilate(C, J, PDMQ)
8.      else
9.          output(Terminate-Belief)
10.         exit Algorithm
11.     end if
12.     if ( PDMQ $\neq$ empty )
13.         x = head(PDMQ)
14.         if ( $x.OI.t_u < T_{G_{BSup}}.t_l$ )
15.         output(Terminate-Belief)
16.         exit Algorithm
17.     end if
18.     if ( J $\neq$ empty )
19.         x = head(J)
20.         if ( $x.OI.t_u < T_{G_{PDM}}.t_l$ )
21.         output(Terminate-Belief)
22.         exit Algorithm
23.     end if
24. end while
25. return (C, J, PDMQ)
end Algorithm

Algorithm Annihilate(C : signal_sequence_set, J : signal_sequence,
        PDMQ : signal_queue)
1.  x = head(PDMQ)
2.  delete_head(PDMQ)
3.  remove signal x having identical fields as x and
        overlapping OIs from J
4.  c = comm_path_id(x)
5.  delete_head(c)
6.  if (c $\neq$ empty)
7.      x = head(c)
8.      insert x, sorted into J based on the lower bound of each signal's OI
9.  end if
10. return (C, J, PDMQ)
end Algorithm
```

Figure 6.6: BSup Input Port Signal Consumption Algorithm

line 2 x represents the path directive for the current path from the PDM. Note that that path directives are identical to signals but are generated by the PDM rather than internally within the BSup

line 3-4 if the current path directive is a *non-matchable directive* (see section 6.1) then it is consumed directly

lines 5-7 if the path directive matches with a signal in J', the signal is consumed. The matching path directive and signal are deleted.

lines 8-11 if the path does not match with any signal in the BSup, the current belief is terminated.

lines 12-17 if a path directive from the PDM exists but no signals exist to be matched and the BSup global time has advanced past the OI of the path directive a signal will never be generated to match the directive. Thus the current belief is terminated.

lines 18-24 if signals exist but no path directive has been generated and the PDM time has advanced past the smallest OI of the signals, a matching path directive will never be generated. The current belief is terminated.

Annihilate Algorithm

lines 1-2 the path directive corresponding to the followed path is deleted

lines 3-5 the consumed signal is deleted

lines 6-9 the consumable signal set is updated

6.6.3 Complexity Analysis

Queue Signal Algorithm

The complexity of the *QueueSignal* algorithm is dominated by line 6 which does an insertion into a sorted list (J). As outlined in section 5.5.4, the maximum size of J is the worst-case fan-in of the corresponding SDL process (c). The algorithm is repeated for each signal queued (N) and each active belief (B). Thus the worst-case running-time complexity of the algorithm is given by 6.3.

$$T_{BSup\_IP\_QueueSig}(B, N, c) = O(B \cdot N \cdot \log c) \tag{6.3}$$

Annihilate Algorithm

The complexity analysis of the *Annihilate* algorithm is dominated by line 8 which does an insertion into a sorted list (J). Thus the running-time complexity of this algorithm is identical to the complexity of *QueueSignal* algorithm and is given by 6.4.

$$T_{BSup\_IP\_Annih}(B, N, c) = O(B \cdot N \cdot \log c) \tag{6.4}$$

Consume Signal Algorithm

Due to the size of the *ConsumeSignal* algorithm, the running-time complexity is presented on a line-by-line basis. Lines with $O(1)$ running-time complexity are omitted from the analysis.

line 1 the outer loop iterates once per signal-pair in the input port, its complexity is $O(N)$

line 3 a binary search of J' whose worst case size is c. The resultant complexity is $O(\log c)$.

line 5 from above, the *Annihilate* algorithm has a running time complexity of $O(\log c)$.

The resultant complexity of the consume signal algorithm per belief is $O(N \cdot \log c)$. The algorithm is re-executed for each belief. Thus the overall complexity is given by 6.5.

$$T_{BSup\_IP\_ConsSig}(B, N, c) = O(B \cdot N \cdot \log c) \tag{6.5}$$

6.7 Scheduling Process Execution within the BSup

Recall from the discussion in section 5.5.6 that the PDM determines the scheduling order corresponding to target system execution. Given that the PDM-model and BSup-model both contain identical SDL processes, the BSup must execute SDL processes in the BSup model according to the scheduling order prescribed by the PDM.

Scheduling order is prescribed by the PDM indirectly. Path directives are generated. For a path directive to be generated a corresponding process must be executed within the PDM. If processes within the BSup are executed in the same order as path-directives are generated, the BSup will follow the scheduling order prescribed by the PDM.

As in the PDM, scheduling is done by the *system* process in the BSup abstract machine. Processes are executed in the order that path-directives are received from the PDM. Recall that the system routes signals (and path directives) between SDL processes. A SDL process within the BSup is queued for execution as path-directives from the PDM are observed.

6.7.1 Complexity Analysis

The complexity of the BSup scheduling algorithm is a function of the number of processes scheduled. In the worst case, each process is scheduled for each signal consumed. A scheduling/de-scheduling operation consists of an addition/deletion from a scheduling queue. Both are $O(1)$ operations. Thus the running-time complexity of the scheduling algorithm for N signals and B beliefs is given by 6.6.

$$T_{BSup\_Sched}(B, N) = O(B \cdot N) \tag{6.6}$$

6.8 Time and Space Complexity of the BSup

The complexity analyzed based on the dominant algorithms described (i.e. *Consume Signal* and *Process Contents*). The time and space complexities are presented individually.

6.8.1 Time Complexity

The running-time complexity of the PDM is dominated by the input port. Thus for B beliefs, a maximum of N signals queued within the BSup and a worst-case fan-in of c, from 6.5, the running-time complexity of the BSup is given by 6.7.

$$T_{BSup}(B, N, c) = O(B \cdot N \cdot \log c) \tag{6.7}$$

6.8.2 Space Complexity

The space complexity of the supervisor is largely dependent upon the number of beliefs generated (B). Each belief makes a duplicate of each signal (N), the state of each process P_s and the scheduling list of worst-case size N_P. Thus for a specification of size S, the space complexity of the BSup is given by 6.8.

$$R_{BSup}(B, N, N_P, S) = O(B \cdot (N + P_s \cdot N_P + N_P) + S) \tag{6.8}$$

6.9 Time and Space Complexity of the Hierarchical Supervisor

6.9.1 Time Complexity

From equations 5.7 and 6.7, it is clear that the running-time complexity of the hierarchical supervisor is dominated by the PDM. Conceptually this makes sense since the PDM must identify the chosen behavioral alternative, a much more complex task than merely detailed behavior checking. The running-time complexity of the hierarchical supervisor is thus given by 6.9.

$$T_{HS}(B, N, t, c, N_P) = O(B \cdot N \cdot ((t + c) \cdot \log c + \log N_P)) \tag{6.9}$$

6.9.2 Space Complexity

The space complexity of the PDM and BSup is largely dominated by the number of beliefs generated as indicated by equations 5.8 and 6.8. The asymptotic space complexity of the PDM and BSup is identical. Thus the overall space complexity of the hierarchical supervisor is given by 6.10.

$$R_{HS}(B, N, N_P, S) = O(B \cdot (N + P_s \cdot N_P + N_P) + S) \tag{6.10}$$

Chapter 7

Evaluation

This chapter is organized into three parts. The first part provides an overview of the structure and operation of a demonstration supervisor. The second part describes the testbed (including target system) that was used to evaluate the supervisor. The third part describes the experiments conducted to evaluate the supervisor.

7.1 Demonstration System

A demonstration supervisor was developed based on the supervisor abstract machines outlined in chapters 5 and 6. The top-level design of the supervisor, in the *object model notation* [20] appears in figure 7.1.

The following two sub-sections describe the static function of each top-level class appearing in figure 7.1 and the dynamic communication between classes under common operational scenarios.

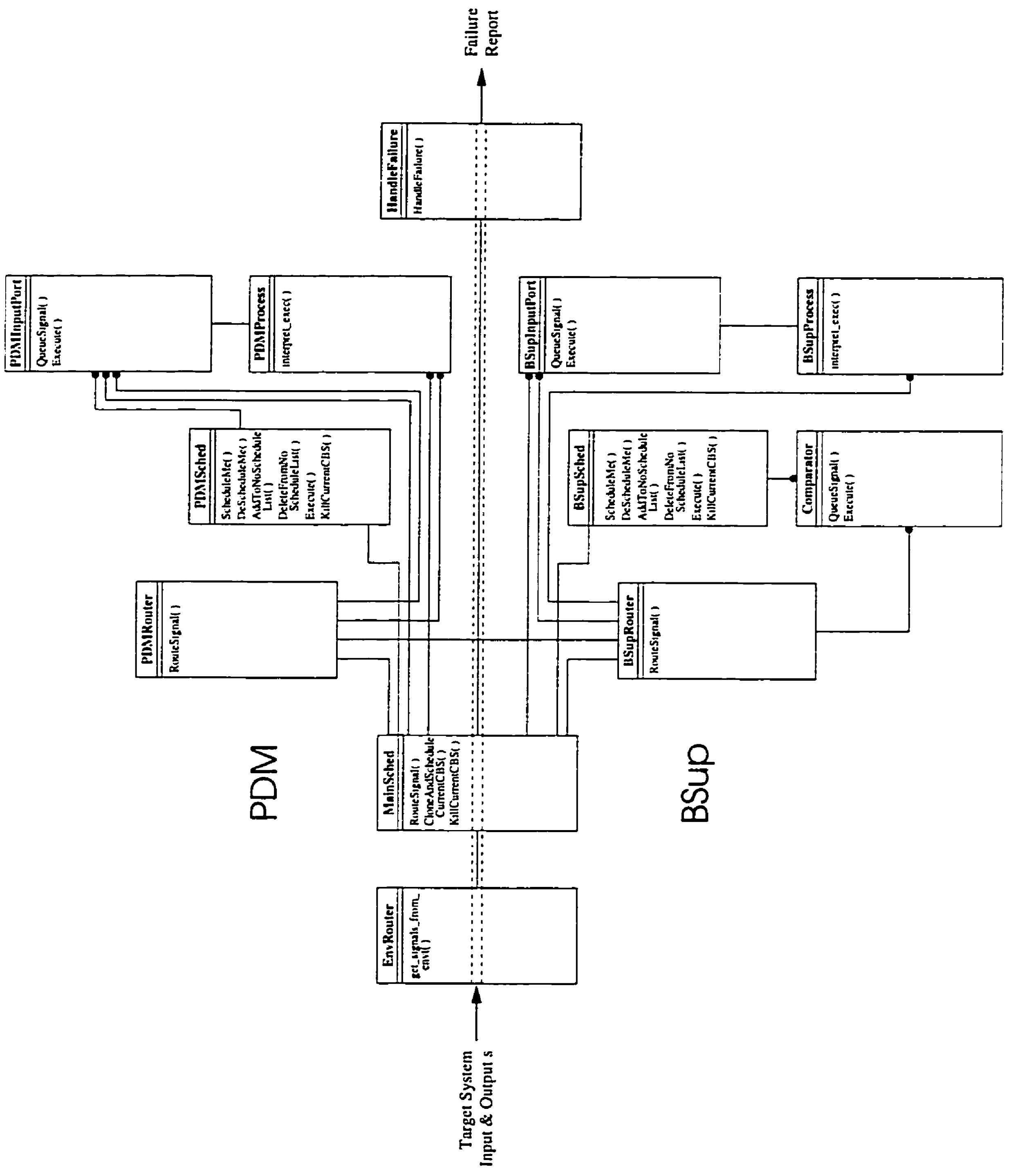


Figure 7.1: High-Level Supervisor Design

7.1.1 Class Description

The principal difference between the specification of the PDM/BSup abstract machines (appearing in figures 5.7 and 6.2 respectively) and the design of the supervisor is that the system process is refined into several classes. The PDM system process is refined into objects: *EnvRouter*, *MainSched*, *PDMRouter* and *PDMSched*. Similarly, the BSup system process is refined into objects: *EnvRouter*, *MainSched*, *BSupRouter* and *BSupSched*. Note that the *EnvRouter* and *MainSched* are shared between the PDM and BSup.

The PDM/BSup path process functionalities are implemented by the *PDMRouter* and *BSupRouter* classes respectively. The PDM/BSup input port and SDL processes are implemented by the *PDMInputPort*, *PDMProcess*, *BSupInputPort* and *BSupProcess* respectively. The BSup comparator process is implemented by the *Comparator* class. A *HandleFailure* class, shared by both the PDM and BSup, implements failure reporting once a failure has been detected. A detailed description of the function of each class follows.

EnvRouter: Collects target system input and output signals. Tags all signals with OIs.

MainSched: Specific functions of this class include:

- creates/terminates and manages the list of beliefs
- compacts beliefs representing identical global states (i.e. beliefs created under don't care non-determinism)
- schedules for execution the PDM, BSup, and EnvRouter

PDMRouter/BSupRouter: Routes signals between processes within the PDM / BSup. Uses the system specification of the PDM/BSup models as the communication topology. The *PDMRouter* includes additional functionality for routing signals from the PDM to the BSup.

PDMSched/BSupSched: Schedule individual *PDMProcess/BSupProcess* objects for execution. Scheduling is implemented as described in chapters 5 and 6. For scheduling purposes, comparators are treated as SDL processes. Thus the *BSupSched* class includes additional functionality to schedule the execution of *Comparator* objects.

PDMInputPort/BSupInputPort: Queue and order signals for consumption by the corresponding process. The *PDMInputPort* uses a partial-order distance table to reduce redundant signal permutation. It creates beliefs when a unique total order of signals cannot be determined. The *BSupInputPort* queues signals into two groups: (1) signals generated by the environment and signals generated internally within the BSup and (2) path directives generated by the PDM.

PDMProcess/BSupProcess: Implement an SDL interpreter for each process. The *PDMProcess* and *BSupProcess* are almost identical in functionality except that the *BSupProcess* does not include support for *ANY* and *none* constructs as described in section 6.1. The *PDMProcess* generates a separate belief for each emanating path from an *ANY* construct and for multiple *none* constructs emanating from a single state.

Comparator: Each process implements one expected and one observed behavior queue per channel. Each *Comparator* process is responsible for queuing signals in the appropriate queue, comparing the contents of queues, annihilating identical queue contents and signaling for the current belief to be terminated if a match between contents cannot be made.

Component	Commented Lines of Source	Non-Commented Lines of Source
PDM	15,200	7,000
BSup	16,400	7,800
Common	6,400	3,600
Total	38,000	18,400

Table 7.1: Hierarchical Supervisor - Lines of Source

HandleFailure: Reports a failure of the target system, terminates operation of the hierarchical supervisor.

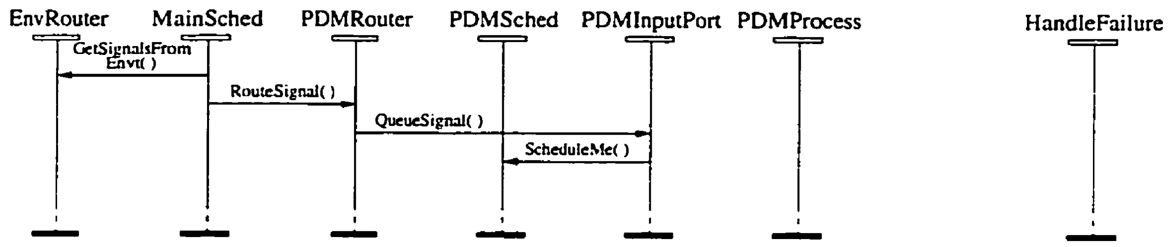
The supervisor was implemented in C++. It consists of approximately 110 different classes, 1000 methods and 38,000 commented lines of source. The line counts of the PDM, BSup and common components of the supervisor appear in table 7.1.

7.1.2 Supervisor Operation

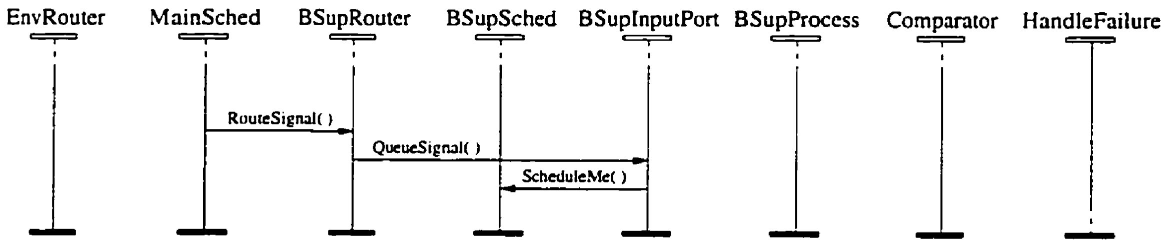
The operation of the hierarchical supervisor is described in several sections. Each of the sections describes one particular aspect of functionality within the supervisor. A textual overview of the functionality is presented, followed by an example of the methods invoked by each class under one particular scenario.

Signal Routing

Observed signals (i.e. inputs and outputs of the target system) are tagged with OIs by the *EnvRouter* and transmitted to *MainSched*. For each belief in existence, *MainSched* duplicates each signal and routes it to the belief. For the currently executing belief, signals are routed to their appropriate destinations by the *PDM-Router* and *BSupRouter*. As an example, the flow of control during routing of a target system input signal in the PDM and BSup is shown in figures 7.2a and 7.2b.



(a) PDM Signal Routing



(b) BSup Signal Routing

Figure 7.2: Signal Routing within the Hierarchical Supervisor

Scheduling SDL Processes and Comparators

The two objectives of scheduling within the PDM are: (1) to order execution of SDL processes and comparators such that expected signals match with the observed signals[1] and (2) to reduce the computational complexity of the supervisor by listing objects ready-to-run and thus eliminating the need to exhaustively search all objects to determine if they are ready-to-run. Processes are scheduled to execute when a

[1]Assuming that the target system is operating as specified.

signal is queued in their input port. Comparators are scheduled to execute when both their expected and observed input queues are non-empty.

In response to the second motivation for scheduling, there are three types of scheduling within the hierarchical supervisor: (1) ready-to-run and (2) not-ready-to-run and (3) not scheduleable. SDL processes with unsaved signals in their input port, or comparators with signals in both their expected/observed behavior queue, are classified as ready-to-run. SDL processes where every signal in the consumable signal set is in the save set and comparators with either the observed or expected behavior queue empty and the other non-empty are classified as not-ready-to-run. If the PDM/BSup global time advances past the OI of the signal with the smallest OI lower bound in the input port/comparator, the process/comparator will never be ready to run and as a result the currently executing belief is terminated. SDL processes with no signals in their input ports are classified as not scheduleable since they cannot execute.

An example of ready-to-run scheduling is shown in figure 7.2a. After a signal is queued in the *PDMInputPort*, the *PDMInputPort* schedules itself to execute. A *PDMInputPort* is re-scheduled only if the scheduling parameters of the process change. Note that the *PDMProcess* is not scheduled however, it is executed by the *PDMInputPort*. Thus it executes after the *PDMInputPort* executes. *BSupInput-Port/BSupProcess* and *Comparator* ready-to-run scheduling operates similarly.

An example of not-ready-to-run scheduling is shown in figure 7.3. A signal is queued in a comparator with an empty observed and expected behavior queue. The flow of control within the supervisor for not-ready-to-run scheduling of a *PDMProcess* and a *BSupProcess* process is similar.

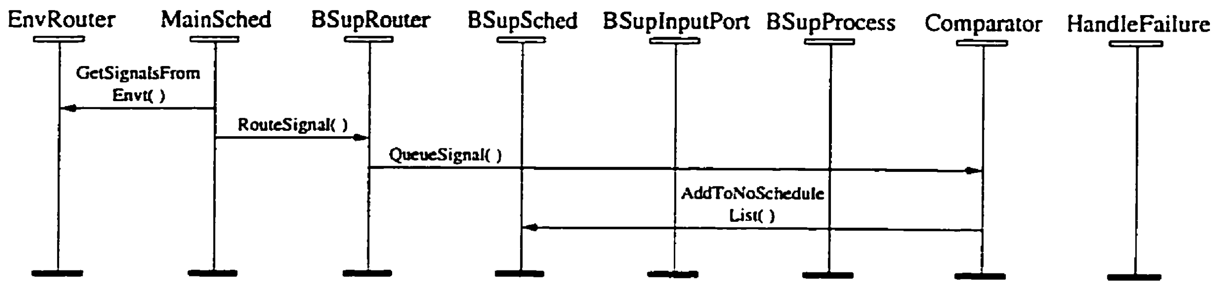


Figure 7.3: Scheduling a not-ready-to-run Comparator

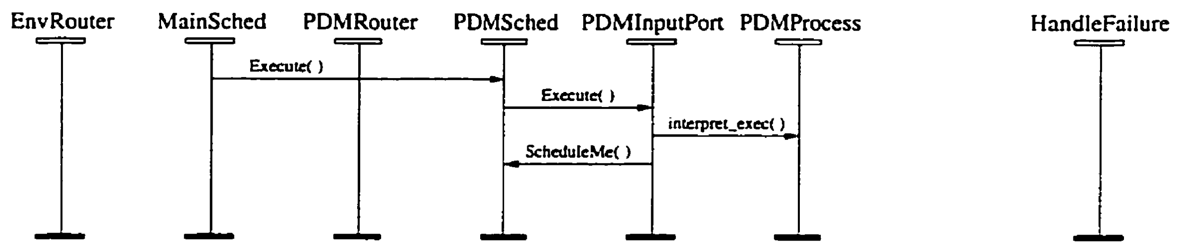
PDMProcess, BSupProcess and Comparator Execution

Execution of a *PDMProcess*, *BSupProcess* or *Comparator* is initiated by the *MainSched*. Initially, all ready-to-run processes in the PDM are executed followed by ready-to-run processes/comparators in the BSup. Both processes and comparators after executing must re-schedule themselves based on the remaining signals in their input ports or expected/observed queues. After execution, processes/comparators may be in a ready-to-run, not-ready-to-run or not scheduleable state (if no signals remain in their input port/queues). Objects in either the ready-to-run or not-ready-to-run state must be scheduled as described in the previous section.

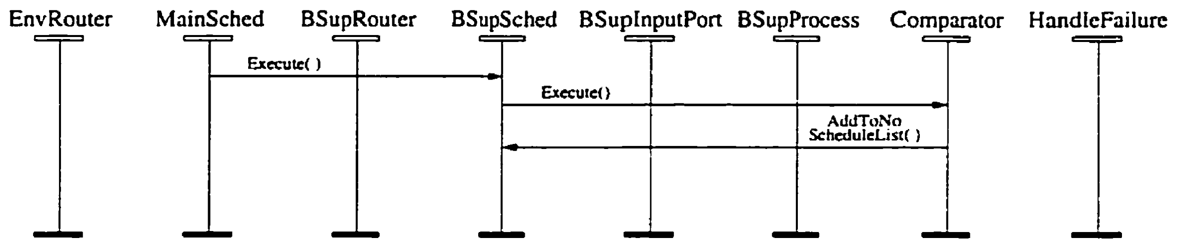
As an example, figure 7.4a shows the flow of control in the hierarchical supervisor when executing a PDM process that becomes ready-to-run after execution. Figure 7.4b illustrates the case where a comparator is executed and after execution only the observed behavior queue (for example) is non-empty (i.e. the comparator is not-ready-to-run).

Belief Creation

As discussed, beliefs are created only by the PDM. Within the PDM, beliefs may be created by either the *PDMProcess* upon the execution of an *ANY* construct, by the



(a) PDM Process Execution



(b) BSup Comparator Execution

Figure 7.4: Execution of a *PDMProcess* and *Comparator*

PDMInputPort if ambiguity exists with regards to the actual signal consumption order or the *PDMSched* if uncertainty exists as to the actual process execution order.

As an example, the flow of control during the creation of a belief by a *PDMProcess* executing an *ANY* construct is shown in figure 7.5a. The other two cases are handled in a similar fashion. Note that the thread of control remains with the current belief. The new belief is subsequentally scheduled by *MainSched*.

Belief Termination

The currently executing belief may be terminated within either the PDM or BSup. Within the PDM/BSup, beliefs may be terminated by the *PDMSched/BSupSched* if the PDM/BSup time advances past the OI of any signal in a *PDMInputPort* /

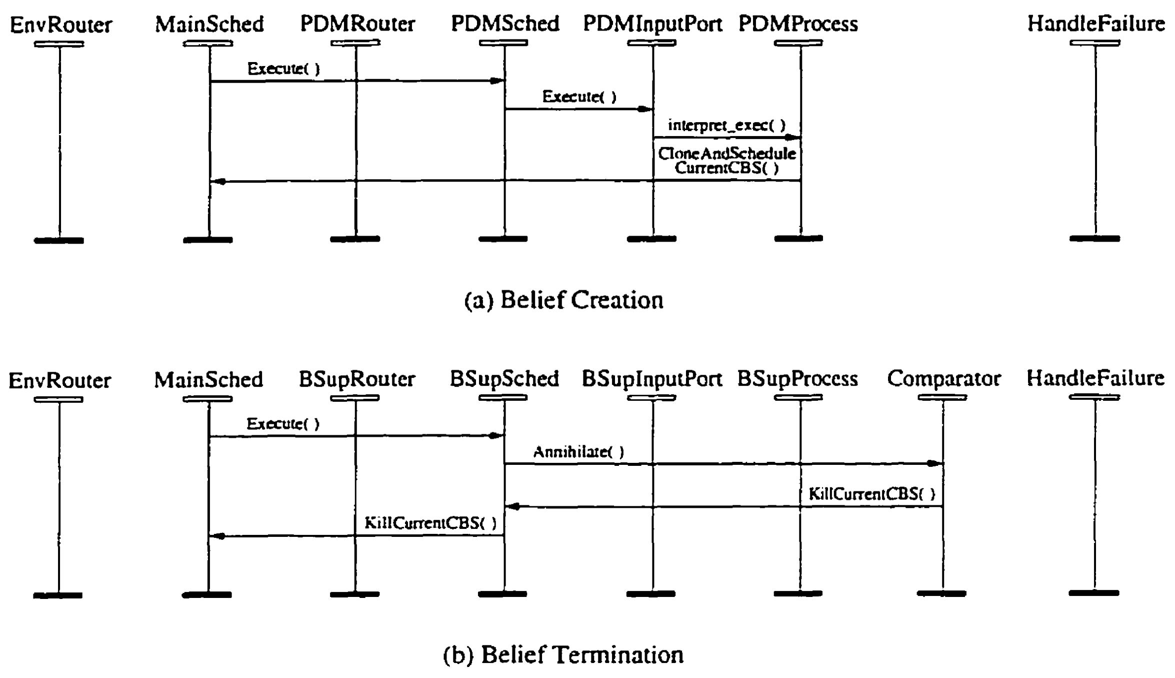


Figure 7.5: Belief Creation/Termination

BSupInputPort that is not-ready-to-run[2]. Additionally, a belief may be terminated by the *Comparator* in one of two cases: (1) if only one of the expected or observed queue is non-empty and the BSup time advances past the OI of the signal at the head of the non-empty queue or (2) if a match cannot be made between the heads of the expected/observed signal queues.

As an example, the case where the contents of the observed/expected behavior queues do not match is illustrated in figure 7.5b.

[2]Recall that PDM and BSup times are based on process times of processes that are ready-to-run.

Redundant Belief Compaction

Beliefs are generated in response to uncertainty as to the behavioral alternative chosen by the target system. In some cases, typically resulting from the tradeoffs made in partial-order signal consumption (described in section 5.3), $n > 1$ beliefs may be generated that represent identical observable behavior. The *redundant belief compaction mechanism* (RBCM) is used to terminate $n - 1$ of these beliefs.

Recall that a belief represents the global state of both the PDM and BSup. Essentially the RBCM compares the global states represented by two beliefs and if identical, terminates one of the two beliefs. To reduce the computational cost of the RBCM, two-level hashing is used during the comparison. Two hash functions, $h_1(), h_2()$ were developed such that for two beliefs, A and B, if the hash values of either functions are different then the two beliefs represent different global states (i.e. if ${h_1}_A() \neq {h_1}_B()$ or ${h_2}_A() \neq {h_2}_B()$ then $A \neq B$).

The first level hash simply takes into consideration the symbolic state of each process and the number of signals in each input port/comparator queue. The second level hash takes into consideration symbolic signal names and OIs of signals. If both the first and second level hash functions are equal for the two beliefs, the global state of the two beliefs is exhaustively compared before one of the two beliefs is terminated.

From empirical measurements, the first-level hash is able to identify approximately 70% of different beliefs and the second level 100% for a sample size of several hundred beliefs.

The RBCM is invoked by the *MainSched* in one of two cases. First, if the number of beliefs, exceeds a threshold (Δ_B) and second if the age of a belief, exceeds a threshold (Δ_T).

Failure Reporting

Failures are reported by the hierarchical supervisor after all beliefs are terminated. *MainSched* manages and schedules beliefs for execution. If the belief scheduling list becomes empty, *MainSched* signals *HandleFailure* to report a failure of the target system. The flow of control within the supervisor is illustrated in figure 7.6.

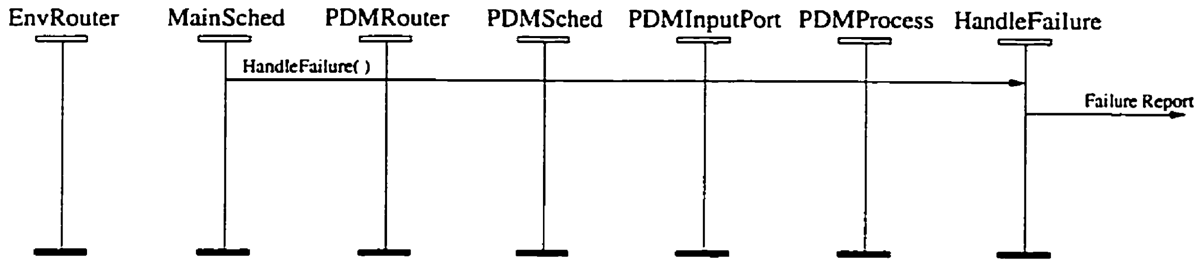


Figure 7.6: Generating a Failure Report

7.2 Evaluation Testbed

The control program of a small telephone exchange was used as a target system based on which the hierarchical supervisor was evaluated. The exchange serviced 60 telephones.

The exchange hardware was simulated and exchange software executed on a UNIX workstation. A random telephone traffic generator served as a generator of inputs. Several tools were used to analyze the traffic data generated. The various components of the test bed and their interconnections are shown in figure 7.7. A detailed description of each component follows.

Telephone Traffic Generator: The telephone traffic generator simulated typical, random *plain old telephone service* (POTS) usage patterns. Several pa-

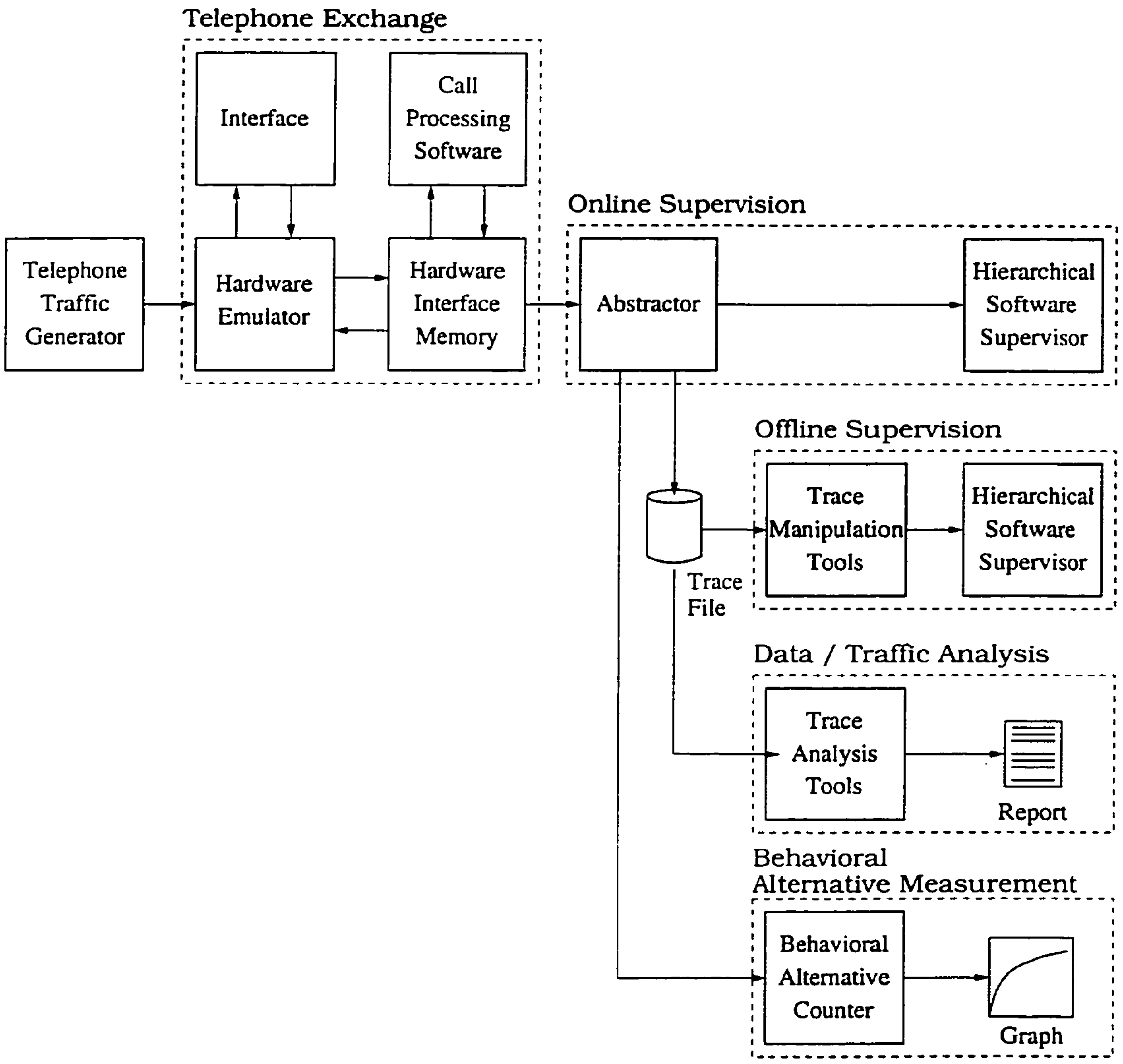


Figure 7.7: Evaluation Testbed

rameters such as the origination rate and connect time were programmable allowing modeling of various load profiles. It executed as a single UNIX process. The generator used in this work is described further in [47].

Hardware Emulator: This unit emulated the exchange hardware. It supported up to 60 telephones. The emulator executed as a single UNIX process.

Hardware Interface Memory: The hardware interface memory represented the memory map of the exchange hardware. It was implemented as a contiguous block of UNIX shared memory.

Call Processing Software: Provides functionality for all telephones serviced by the exchange and manages shared hardware resources. The SDL requirements specification and an overview of the call processing software can be found in Appendix A.

Interface: Served two purposes: first, it provided a visual display of the state of each telephone and second, allowed manual telephone calls to be placed. Note that use of the user interface in the latter case excludes use of the telephone traffic generator.

Abstractor: Translated bit sequences appearing in the hardware interface memory into signals as appearing in the SDL requirements specification.

Hierarchical Software Supervisor: Consists of the PDM-model, BSup-model and interpreters as described in section 7.1.

Trace Manipulation Tools: Permitting seeding random failures at random points in the trace file.

Trace Analysis Tools: A collection of utilities for analysis of telephone traffic statistics. Parameters such as the number of originations, number of calls routed to slow busy, number of calls routed to fast busy etc. are generated from the contents of a trace file.

Behavioral Alternative Counter: A tool used to measure the total number of behavioral alternatives (i.e. don't know and don't care) that arise under a given requirements specification and traffic load over time.

The components of the testbed are written in five programming languages as some languages are more suitable for certain applications than others. The majority of the testbed is written in C and C++, the Interface which is largely graphical is written in Tcl/Tk, the Trace Manipulation/Analysis Tools are written in Perl and csh. The entire testbed consists of approximately 70,000 lines of commented source.

7.3 Evaluation

The hierarchical supervisor presented in this thesis was evaluated along two lines: (1) its failure detection capability and (2) its time/space complexity.

The section begins with an experimental evaluation of the supervisor's ability to detect failures and to simultaneously limit generation of false-failure reports. An analysis of the size of the problem space (i.e. the total number of behavioral alternatives) is presented next. This is followed by experimental evaluations of the supervisor time and space complexity. The section concludes with a commentary on the scalability of the hierarchical supervision to industrial systems based on the evaluations presented.

7.3.1 Failure Detection Capability

The failure detection capability of hierarchical supervision was evaluated with the aid of the target system described. The supervisor was used to monitor the exchange for extended periods of time. The failure detection capability of the supervisor was evaluated based on two attributes: (1) the supervisor's spurious failure reporting and (2) the supervisor's failure detection capability. Both sets of evaluations are presented in the following two sub-sections.

Spurious Failure Reporting

Spurious failure reporting refers to the number of unwarranted failure reports generated by the supervisor. It was evaluated by having the supervisor monitor the operation of the target system for several thousands of call originations. Typical reliability requirements for North American telephone switching systems are that up two calls out of ten thousand may be mishandled. These requirements were used as a guideline in setting the interval during which the supervisor was executed.

The supervisor was used to supervise several traces consisting of over twenty thousand call originations ranging in origination rates from 2-6 calls/phone/hour. The loads were chosen to range from heavy residential to heavy commercial traffic levels. The target system call processing software was a third-generation debugged version. The supervisor detected several failures in the output of the exchange. Detected failures were subsequentally traced back to either (1) faults in the PDM/BSup model or (2) residual faults in the target system control program. The faults in the PDM/BSup models were introduced during the transformation of the models from the requirements specification and resulted from human error. The results are summarized in table 7.2.

Fault Category	Number of Fault Types	Number of Instances
Supervisor Model	1	1
Implementation	2	6

Table 7.2: Supervisor Failure Detection Capability

One supervisor model fault type was detected by the supervisor. The supervisor was able to detect and subsequentally report the discrepancy. The supervisor model fault was due to resources being incorrectly deallocated. When a call was placed and the terminating party was busy, resources were not deallocated upon the originating party going onhook. The supervisor reported a failure after all resources within the supervisor model were depleted (i.e. after the effect of resources not being deallocated became externally visible).

Two types of residual target system faults manifested themselves as externally observable failures. The first related to the scanning of digits dialed by the user. When waiting for the first digit, the control program disconnected and re-connected the touch-tone receiver hardware as part of the process of removing dial-tone. The connection/disconnection of the touch-tone receiver is not an externally observable event. However, it was interpreted by the supervisor (or more precisely by the abstractor) as two separate digits dialed. The second failure type was due to a difference between a specified and implemented timeout duration. The supervisor reported the external signal generated after the timeout as a failure, because it was not expecting it at that time.

All failures detected by the supervisor were either traced back to faults in the supervisor model or residual faults in the target software system. Based on the experiment conducted, no unwarranted failure reports were generated by the hierarchical supervisor.

Failure Detection Capability

The evaluation of the supervisor failure detection capability is difficult due to the large sizes of the trace files. Manual verification that a given trace represents a behavior corresponding to the specification is almost impossible.

For this reason, the failure detection capability of the supervisor was evaluated by seeding known failures into a trace representing the execution of the exchange. The failure model consisted of altering the signals emitted during state transitions [54]. Three types of failures were seeded: (1) signal removal, (2) signal insertion and (3) signal replacement. Note that the final failure type is a combination of the first two.

Two different types of evaluations were carried out: exhaustive and random. They differed principally in how failures were seeded. Exhaustive evaluation is better suited for use with small trace files due to its computational cost. Random evaluation is better suited for use with large trace files. Evaluations of the supervisor based on these two types of evaluations are described below.

Exhaustive evaluation refers to seeding all three types of failures at each line of the trace file. Ten small trace files representing loads from 2-20 calls/phone/hour were used. Each line of the trace was seeded with all three failure types, representing a total of approximately 30 failures per line. The traces contained a total of approximately 10 calls or 200 lines each. Thus a total of $10 \cdot 30 \cdot 200 = 60,000$ failures were seeded in separate traces. The supervisor was executed on each individual trace. The presence of all seeded failures were reported by the supervisor.

Random evaluation refers to seeding randomly chosen failures at random locations in a given trace file. Fifteen trace files representing approximately 20,000 calls at loads ranging between 2-20 calls/phone/hour were seeded with the three failure types described. A total of approximately 60,000 failures were seeded into separate traces. The supervisor was executed on each trace individually. The presence of all seeded failures were reported by the supervisor.

7.3.2 Number of Legitimate Behavioral Alternatives

The size of the supervisor problem space was estimated by measuring the number of legal behavioral alternatives (BAs). Internally, BAs arise from specification non-determinism under a particular input scenario. Within the supervisor, they are represented as beliefs. A subset of the BAs generated by the supervisor actually lead to different externally observable behavior.

The number of generated BAs is a function of the requirements specification and the target system load. The small telephone exchange was run under several different traffic loads. The maximum number of BAs generated by the supervisor (i.e. beliefs) for each load is plotted in figure 7.8.

Further analysis on the number of generated BAs was done to determine the proportion of don't care and don't know BAs. BAs were grouped into n sets: $s_1, s_2, \cdots, s_n$. All BAs in set s_i result in identical observable behavior (i.e. don't care BAs). While any two BAs in sets s_i and s_j $i \neq j$ represent different observable behavior (i.e. don't know BAs). Thus the total number of don't know BAs is n and total number of don't care BAs is equal to $(\#s_1 + \#s_2 + \cdots + \#s_n - n)$ where $\#$ represents a set cardinality operator. For the experiment described, the results are plotted in figure 7.8.

As expected, the total number of behavioral alternatives is very large. This is

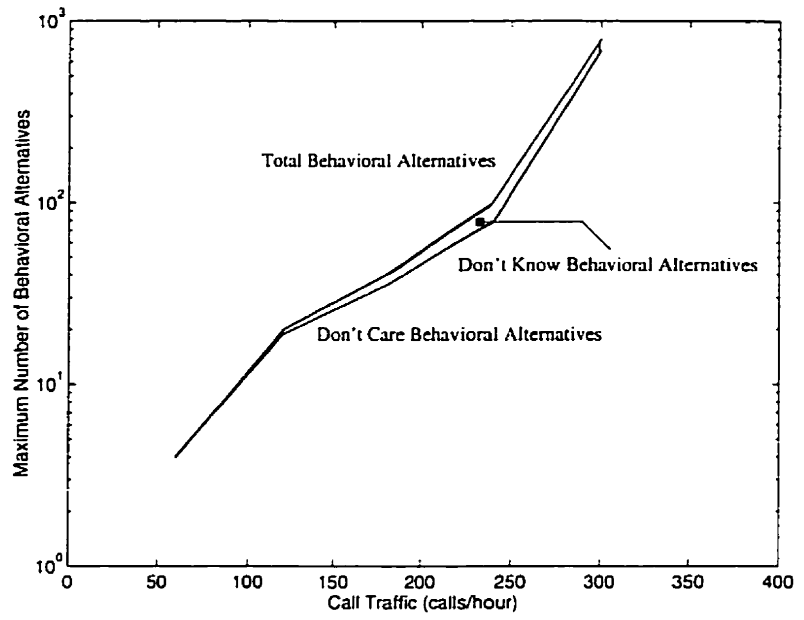


Figure 7.8: Measured Number of Beliefs

due to the worst case factorial number of possible signal interleavings at the input ports of SDL processes, each leading to a legitimate BA. Few of these BAs actually lead to different observable behavior, making the motivation for pruning such BAs from consideration strong.

7.3.3 Number of Behavioral Alternatives Generated

The number of behavioral alternatives generated is a key parameter in both the time and space complexity of the hierarchical supervisor. The supervisor was executed on the load described in section 7.3.2. The number of behavioral alternatives generated is plotted in figure 7.9.

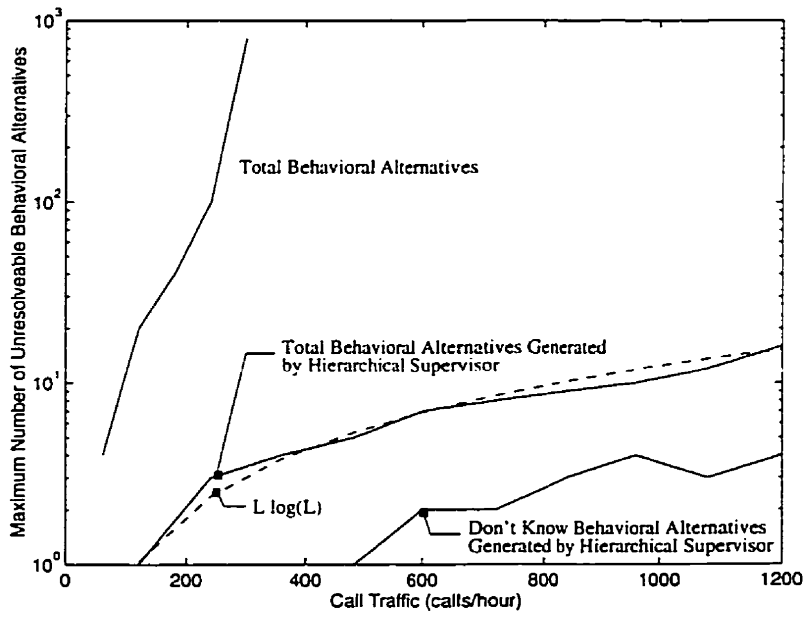


Figure 7.9: Number of Behavioral Alternatives Generated

As shown, the hierarchical supervisor significantly reduces the number of behavioral alternatives considered. The majority of the BAs generated are don't care BAs due to the tradeoffs made with partial-order signal consumption. As the load on the exchange is increased, the number of don't know BAs increases. This is principally due to resource starvation; the PDM is not able to determine which of the two resources in the target system have been depleted based on the signals observed (i.e. which path was followed). For the example system described, a PDM would be able to accurately track the don't know BAs for an exchange with properly provisioned resources.

The actual number of beliefs generated is highly application specific. It depends on the requirements specification, the algorithm used to derive the PDM-model, the load and the detailed implementation of algorithms in the PDM interpreter. Empirical curve fitting revealed that for an exchange subject to a traffic load L, the number of beliefs generated by the hierarchical supervisor is of order $O(L \cdot \log L)$ as shown in figure 7.9. This is a substantial reduction from the factorial-number of total legitimate BAs.

7.3.4 Running-Time Complexity

This section presents empirical validation of the running time complexity of the hierarchical supervisor as given by equation 6.9. For the experiment described, the worst case fan-in of each process (c), the number of signal types (t) and the worst-case number of SDL processes N_P are defined by the specification topology and are treated as constants since the evaluation deals only with one target system. Thus equation 6.9 reduces to $T_{HS'}(B, N) = O(B \cdot N)$.

From the empirical analysis presented in section 7.3.3, B can be estimated as $B = O(L \cdot \log L)$ where L represents the load on the exchange. N, representing the number of signals in the supervisor, increases linearly with the load on the exchange. Thus N can be approximated as $N = O(L)$. The resultant running-time complexity of our example is thus given by 7.1.

$$T_{HS'} = O(B \cdot N) \approx O(L^2 \cdot \log L) \tag{7.1}$$

The hierarchical supervisor was used to monitor the operation of the target system at several different operational loads. As the load increased, the number of beliefs generated increased (as described in section 7.3.3), increasing the CPU time required by the supervisor per telephone call.

The supervisor CPU time per call was measured by running several hundred calls and averaging the total supervisor running time by the number of originations. The number of originations was made large to reduce the effect of the supervisor initialization on the total running time. Supervisor running time was measured using the UNIX **getrusage** system call.

The CPU time per call is plotted in figure 7.10 for various operational loads. Results were obtained on a machine having a SPECint95 and SPECfp95 of 1.0. A $O(L^2 \cdot \log L)$ curve is plotted as a reference. As shown, good correspondence between the predicted and measured running-time complexity was observed.

7.3.5 Space Complexity

The predicted space complexity of the supervisor (given by equation 6.10) was compared with the measured space complexity of the supervisor developed. For the particular experiment described, only one specification was considered. Thus the specification size S and the number of processes in the specification N_P is constant.

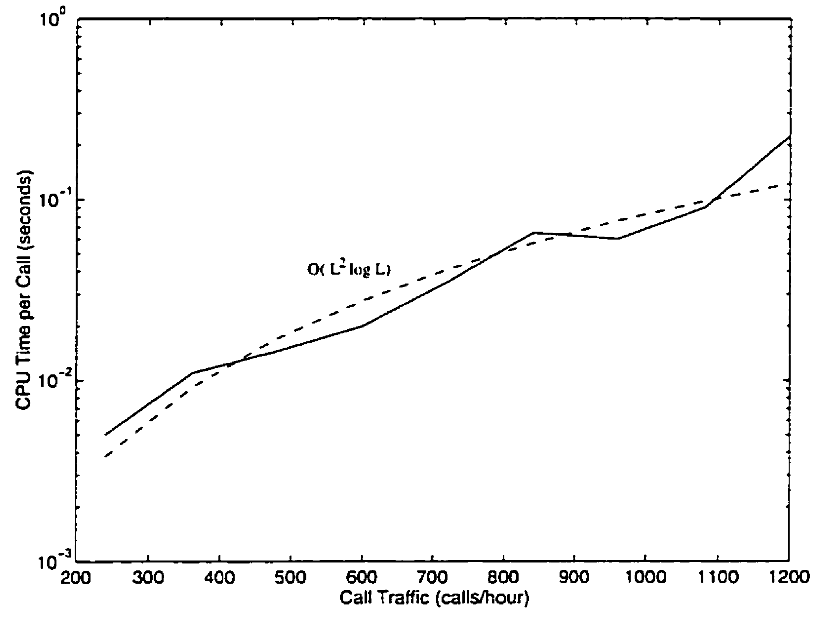


Figure 7.10: CPU Time Per Call

The number of signals within the supervisor, N is approximated as $N = O(L)$ (as outlined in section 7.3.4). Thus the resultant space complexity of our example is given by 7.2.

$$R_{HS'} = O(B \cdot N) \approx O(L^2 \cdot \log L) \tag{7.2}$$

The exchange was executed over several different traffic loads. The maximum memory usage of the supervisor was determined with the aid of the UNIX **top** command[3]. The measured supervisor memory usage is plotted in figure 7.11 for various loads. The constant size of the supervisor executable was subtracted from the results plotted.

7.3.6 Scalability

This section attempts to extrapolate the time complexity results presented to larger systems. The results are meant only to serve as a general indicator of the scalability of the approach. An actual system would introduce factors not taken into consideration in the presentation below such as a larger requirements specification.

Most telephone exchanges have modular organization to facilitate module reuse and to allow ease of expandability. For example, the line interface module (LIM) that interfaces subscribers telephones with the central exchange controller services approximately 1000 lines in both the Northern Telecom DMS-100 [57] as well as the Lucent 5ESS [19]. It would be difficult to observe the inputs and outputs of the

[3]The supervisor contains an internal memory manager. Memory, when deallocated is returned to the supervisor memory pool rather than the operating system memory pool. Thus the maximum memory usage of the supervisor results just before the supervisor completes its execution.

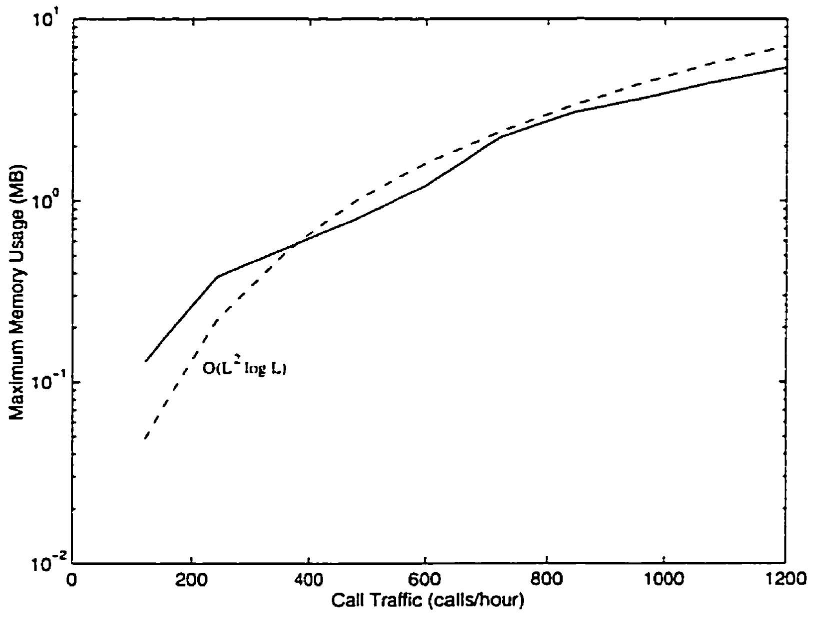


Figure 7.11: Maximum Memory Usage

entire exchange. However, a supervisor would be suitable for monitoring a single module such as the LIM.

The CPU requirements of the supervisor were estimated based on maximum business traffic (i.e. 6 calls/phone/hour). The supervisor is assumed to monitor a LIM servicing 1000 telephones at the standard business origination rate of 6 calls/hour/phone. Extrapolating from figure 7.10, the supervisor running on a machine having a SPECint95 and SPECfp95 of 1.0 requires approximately 3 cpu seconds/call at this load. The LIM is required to process 6×1000 calls/hour or 1.67 calls/second. Thus a CPU having a SPECint95 and SPECfp95 greater than $3 \times 1.67 = 5$ (e.g. any modern Intel Pentium processor) would be sufficient for this application.

Chapter 8

Conclusions

This thesis addressed the automatic detection of software failures or software supervision. The software supervisor is a unit that monitors the inputs and outputs of a given target software system. It makes use of the target system's requirements specification as a definition of correct behavior. Discrepancies between specified and observed behaviors are reported as failures by the supervisor.

The complexity and sophistication of modern software systems makes automatic detection of failures an industrially important area of research. Three potential applications of supervision include on-line detection of failures, evaluation of testcase results during software development and the collection of accurate failure data to identify problem areas and improve the reliability of software.

This thesis focuses on the supervision of real-time reactive software systems. This class of systems represents some of the largest and most complex software ever developed. The case where the requirements specification of the target system external behavior appears in a finite state machine based formalism is considered.

Software supervision is a highly complex activity. Several open research issues related to supervision exist. The principal issue addressed by this work is the computationally efficient handling of specification non-determinism. Non-determinism is an important component of a specification formalism. It permits the specification writer to avoid stating unpertinent aspects of behavior. This leaves freedom to the software designer to choose the least costly or otherwise desirable alternative. However, the supervisor must be able to consider all legitimate behavioral alternatives such that unwarranted failure reports are not generated. A potentially large number of alternatives exist even for moderate size systems and exhaustive consideration of all alternatives is prohibitive. Hierarchical supervision addresses this issue.

8.1 Hierarchical Supervision

A novel approach to supervision, called hierarchical supervision, was proposed. The objective of the approach is the efficient handling of specification non-determinism.

Hierarchical supervision improves the efficiency of non-determinism handling by a divide-and-conquer approach. Supervision is split into two sub-problems: (1) determination of the path through the specification chosen by the target system and (2) detailed behavior checking. The corresponding architecture of the hierarchical supervisor has two layers. The *path detection module* (PDM) determines the path through the specification chosen by the target system while the *base supervisor* (BSup) checks that the followed path was actually the legitimate one.

The functionality underlying the BSup has been studied extensively and is relatively well understood. However, the PDM has not been addressed previously. The major focus of the thesis is on the PDM.

The PDM relies on signals, generated by the target system, that uniquely identify the followed path through the target system. The precision of tracking the target system improves with the availability of signals that uniquely identify the path followed.

Hierarchical supervision is best suited for target systems where the average uniqueness of signals used by the PDM to track the target system (i.e. PDM-model stimuli)[1] is greater than the average uniqueness of the requirements specification stimuli. The chosen behavioral alternative is identified by the PDM based on a subset of the signals directed to/from the target system. Unique signals may be mapped to fewer state transitions than less unique ones. As a result, fewer behavioral alternatives need be considered by the supervisor. The average number of behavioral alternatives explored by a hierarchical supervisor decreases as the average uniqueness of signals chosen to track the target system increases.

8.2 Major Research Contributions

This thesis presented five major research contributions to cost-effective automatic detection of software failures in the presence of specification non-determinism: (1) the notion of splitting supervision into two sub-problems: path determination and detailed behavior checking, (2) improvement of the accuracy of path determination by the use of both target system input and output signals, (3) exploration of the tradeoffs in having the supervisor lag the target system, (4) development of a method for pruning alternatives arising from specification non-determinism not leading to different observable behaviors and (5) development of a base supervisor,

[1]An underlying assumption is that a suitable metric of uniqueness exists. The reader is referred to section 4.2.1 for a definition of one such metric.

a directed simulator for detailed behavior checking. A further description of each contribution follows.

Splitting supervision into two sub-problems separates the two fundamental functionalities of the supervisor. It may be considered a divide-and-conquer approach to reducing the computational cost of supervision. The two components of the hierarchical supervisor which implement these functionalities differ substantially in their purpose, design and implementation. The result is that each component implements a more specialized function than a monolithic supervisor allowing for improved efficiency and a conceptually simpler implementation.

Hierarchical supervision makes use of both target system input and output signals to determine the path chosen by the target system. Thus the occurrence of a state transition in the requirements specification may be determined to have taken place by either an input or output signal. This improves the use of the information provided about the path traversed by the target system. From the perspective of the supervisor, it reduces the number of behavioral alternatives that need to be considered. However, it complicates the derivation process of the PDM-model which must ensure that sequences of state transitions in the PDM-model follow a similar causal path as in the requirements specification.

A supervisor that has the capability to lag the target system by a sufficiently long period Δ (or an out-of-time supervisor) needs only consider *what happened* rather than *what may happen*. The advantage of the approach is that only a subset of the behaviors need to be considered by such a supervisor. In addition, signals generated by the target system may be stored, allowing the supervisor to process peak target system activity over a longer period of time. The tradeoff with out-of-time supervision is the increased space requirement required to store signals generated by the target system during the interval Δ in addition to the latency of failure reporting by a worst-case period, Δ.

In many requirements specifications and operational scenarios, a number of behavioral alternatives arise from specification non-determinism that do not lead to different externally observable behaviors. Partial-order techniques were proposed to prune such alternatives from consideration. The approach makes use of static information compiled from the requirements specification. Static information is used to dynamically discard alternatives arising from specification non-determinism not leading to unique externally observable behavior. A spectrum of such algorithms can be envisioned, each suited for different applications. However, in general, a tradeoff exists between the time/space resource requirements of the approach and its capability to prune behavioral alternatives.

At the core, a software supervisor must have a simulator to generate expected behaviors of the target system. Expected behaviors are compared with observed behaviors to determine and failures reported if a match cannot be made. A typical simulator chooses a behavioral alternatives in the presence of non-determinism. However, the proposed simulator (i.e. the BSup) is directed by the PDM along the behavioral alternative chosen by the target system.

8.3 Future Work

The fundamental contributions described may have further applications than those described in this thesis. Future work is sub-divided into three categories: (1) further reductions in computational complexity arising from specification non-determinism, (2) continuation of supervision after detection of a failure and (3) alternate applications of the described work. A discussion of each follows.

This thesis described a hierarchical approach to software supervision consisting of two layers. Experience gained in domains such as artificial intelligence planning indicate that N-layer problem solving is a principal means of dealing with computational complexity [33].

The two-layer approach to supervision could be extended into an N layer approach by abstracting paths and successively resolving paths at lower layers in the supervisor. Several state transitions could be abstracted into a single, aggregate state transition. Upon determination that the aggregate state transition has taken place, the supervisor effectively knows the destination state. Subsequently lower layers could then resolve the actual path from the previous composite state to the current composite state. It is believed that such an approach would yield further reductions in computational complexity provided that sufficiently unique signals exist to track the target system. The tradeoff with the approach is the increased delay in failure reporting.

Supervision requires that the state of the target system and the specification state of the supervisor be in-sync. However, few assumptions can be made about the the post-failure specification state of a target system. For supervision to continue, an approach to determining the state of the target system after the occurrence of a failure is needed.

The notion of path detection is similar to the notion of state detection. Path detection attempts to identify the state transition that took place from the emanating state. State detection requires that the state transition that took place and consequentially the final state be identified without a notion of the current state. Thus the research contributions developed for path detection such as tracking the execution path by means of unique signals and delaying the reporting of the execution path may be applied to state detection. One obvious difficulty is the enormous potential state space. Research results indicate a tradeoff between the amount of time spent determining the state and the computational complexity of the approach [35, 52]. A unit to determine the post failure state will probably have to lag the target system by an interval greater than the PDM.

The work on path detection as described may have several other applications other than supervision. For example, a PDM with a properly instrumented model may be used as a quality of service (QoS) monitor. For systems that have large amounts of internal state, simple assertion checking is typically not suitable. The out-of-time orientation of the described PDM is naturally suitable for monitoring QoS. Other applications include resource utilization monitoring and specification coverage metering for applications such as software testing.

Bibliography

[1] R.P. Almquist, J.R. Hagerman, R.J. Hass, R.W. Peterson, and S. L. Stevens. Software protection in No. 1 ESS. In *International Switching Symposium Record*, pages 565–569. IEEE, 1972.

[2] Rajeev Alur, Costas Courcoubetis, and Mihalis Yannakakis. Distinguishing tests for non-deterministic and probabalistic machines. In *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing*, pages 363–372, 1995.

[3] Algirdas Avižienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[4] J.M. Ayache, P. Azema, and M. Diaz. Observer, a concept for on line detection for control errors in concurrent systems. In *9th International Symposium on Fault-Tolerant Computing*, pages 79–86. IEEE, 1979.

[5] J.M. Ayache, J.P. Courtiat, and M. Diaz. Self-checking software in distributed systems. In *3rd International Conference on Distributed Computer Systems*, pages 163–170. IEEE, 1982.

[6] F. Bacchus and Q. Yang. The expected value of hierarchical problem-solving. In *Proceedings of the annual National Conference on Artificial Intelligence.* American Association for Artificial Intelligence (AAAI), 1992.

[7] F. Belina, D. Hogrefe, and A. Sarma. *SDL with Applications from Protocol Specification.* Prentice Hall International, 1991.

[8] M. Blum and H. Wasserman. Program result-checking: A theory of testing meets a test of theory. In *35th Annual Symposium on Foundations of Computer Science (FOCS '95)*, pages 382–392, 1994.

[9] M. Blum and H. Wasserman. Reflections on the pentium division bug. *IEEE Transactions on Computers*, 45(4):385–393, April 1996.

[10] D.B. Brown, R.F. Roggio, J.H. Cross, and C.L. McCreary. An automated oracle for software testing. *IEEE Transactions on Reliability*, 41(2):272–279, June 1992.

[11] Alan Burns and Andy Wellings. *Real-Time Systems and their Programming Languages.* Addison-Wesley, 1990.

[12] S.E. Chodrow, F. Jahanian, and M. Donner. Run-time monitoring of real-time systems. In *Proceedings of the Real-Time Systems Symposium*, pages 74–83. IEEE, 1991.

[13] John R. Connet, Edward J. Pasternak, and Brude D Wagner. Software defenses in real-time control systems. In *Fault-Tolerant Computing*, pages 94–99, June 1972.

[14] M. Diaz, G. Juanole, and J.P. Courtiat. Observer–a concept for formal on-line validation of distributed systems. *IEEE Transactions on Software Engineering*, 20(12):900–913, December 1994.

[15] D.Lee, A.Netravali, K.Sabnani, and B.Sugla. Passive testing and applications to network management. to appear.

[16] D.Lee and M.Yannakakis. Principles and methods of testing finite state machines - a survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[17] P.S. Dodd and C.V. Ravishankar. Monitoring and debugging distributed real-time programs. *Software – Practice and Experience*, 22(10):863–877, October 1992.

[18] P. Edwards, editor. *The Encyclopedia of Philosophy*, volume 2. Collier - Macmillan Limited, London, 1967. pages 359–378.

[19] H.G. Holland et. al. The 5ESS-2000 switch: Exceeding customer expectations. *AT&T Technical Journal*, 73(6):28–38, November/December 1994.

[20] J. Rumbaugh et. al. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[21] Peter G. Bishop et al. PODS – A Project on Diverse Software. *IEEE Transactions on Software Engineering*, SE-12(9):929–940, September 1986.

[22] International Organization for Standardization. *A Formal Description Technique based on Temporal Ordering of Observational Behavior.* ISO/IEC 8807, Geneva, 1989.

[23] International Organization for Standardization. *A Formal Description Technique based on Extended State Transition Model.* ISO/IEC 9074, Geneva, 1990.

[24] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State Explosion Problem.* PhD thesis, Universite de Liege, Facultè des Sciences Appliquèes, October 1994.

[25] D.B. Hay. A belief method for detecting operational failures in soft real time systems. Master's thesis, University of Waterloo, Department of Electrical and Computer Engineering, Waterloo, Ontario, Canada N2L 3G1, 1991.

[26] D.B. Hay and R.E. Seviora. A real-time validator. In *Proceedings of the Third IEE International Conference on Software Engineering for Real-Time Systems*, pages 199–204. IEE, 1991. IEE Publication Number 344.

[27] C.A.R Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[28] G. Holzmann and D. Peled. An improvement in formal verification. In *Proceedings of the Seventh International Conference on Formal Description Techniques (FORTE'94)*, Berne, Switzerland, October 1994.

[29] Yennun Huang and Chandra Kintala. Software implemented fault tolerance: Technologies and experience. In *The Twenty-Third Annual International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 2–9, Los Alamitos, California, June 1993. IEEE Computer Society Press.

[30] Radu Iorgulescu. Resynchronization in supervision of soft real-time systems. Master's thesis, University of Waterloo, Department of Electrical and Computer Engineering, Waterloo, Ontario, Canada N2L 3G1, 1991.

[31] Peter Klein. The safety-bag expert system in the electronic railway interlocking system Elektra. In *EXPERTSYS-90*, pages 177–182, 1990.

[32] J.C. Knight and N.G. Leveson. An experimental evaluation of the assumption of independence in multi version programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[33] Craig A. Knoblock. *Automatically Generating Abstractions for Problem Solving.* PhD thesis, Carnegie Mellon, School of Computer Science, Pittsburgh, PA 15213, May 1991. CMU-CS-91-120.

[34] Craig A. Knoblock, Josh D. Tenenberg, and Qiang Yang. Characterizing abstraction hierarchies for planning. In *Proceedings of the Annual National Conference on Artificial Intelligence*, pages 692–697. American Association for Artificial Intelligence (AAAI), 1991.

[35] Radmila Kovacevic. A resynchronization scheme for belief-based real-time software supervision. Master's thesis, University of Waterloo, Department of Electrical and Computer Engineering, Waterloo, Ontario, Canada N2L 3G1, 1991.

[36] Robert Kowalski. *Logic for Problem Solving.* Artificial Intelligence Series. North Holland, New York, 1979.

[37] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, Spring 1992.

[38] P.A. Lee and T. Anderson. *Fault Tolerance – Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems Series.* Springer-Verlag, Wein, New York, 2nd edition, 1990.

[39] J.J. Li. *A Real-Time Software Supervision Approach for Automatic Failure Detection.* PhD thesis, University of Waterloo, Department of Electrical and Computer Engineering, Waterloo, Ontario, Canada, N2L 3G1, 1996.

[40] Luqi, H. Yang, and X. Zhang. Constructing an automated texting oracle: An effort to produce reliable software. In *Computer Software and Applications Conference*, pages 228–233. IEEE, 1994.

[41] M.N. Meyers, W.A., Routt, and K.W. Yoder. No. 4 ESS: Maintenance Software. *The Bell System Technical Journal*, 56(7):1139–1167, September 1977.

[42] Kenneth P. Parker. *The Boundary-Scan Handbook.* Kluwer Academic Publishers, Norwell, Massachusetts 02061, 1992.

[43] Brian Penney. The DMS-100: A switch that takes care of itself. *Telesis*, Four:41–43, 1980.

[44] S. Poledna. *Fault-Tolerant Real-Time Systems – The problem of Replica Determinism.* Kluwer Academic Publishers, 1996.

[45] Danny Prairie. Supervising a class of software systems with two weakly interacting functionalities. Master's thesis, University of Waterloo, Department of Electrical and Computer Engineering, Waterloo, Ontario, Canada N2L 3G1, 1996.

[46] Roger S. Pressman. *Software Engineering – A Practitioner's Approach.* McGraw-Hill Inc., third edition, 1992.

[47] Scott Reid. Reduced model supervision: Quantifying trade-offs in failure detection and computational complexity. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1, 1996.

[48] Elaine Rich and Kevin Knight. *Artificial Intelligence.* McGraw-Hill Inc., 1991.

[49] D.J. Richardson, S.L. Aha, and T.O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118, 1992.

[50] S. Sankar and M. Mandal. Concurrent runtime monitoring of formally specified programs. *IEEE Computer*, pages 32–41, March 1993.

[51] T. Savor and R.E. Seviora. User-oriented supervision of SDL-specified software. In *Fourth Bellcore/KPN/Purdue Workshop on Issues in Software Reliability*, October 1995.

[52] T. Savor and R.E. Seviora. An approach to automatic detection of software failures in real-time systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, 1997. To appear.

[53] E. Schoitsch, E. Dittrich, S. Grasegger, D. Kropfitsch, A. Erb, P. Fritz, and H. Kopp. The Elektra testbed: Architecture of a real-time test environment for high safety and reliability requirements. In *Proceedings of the 1990 IFAC/IFIP Symposium on Safety of Computer Control Systems (SAFECOMP '90)*, pages 59–65, 1990.

[54] D.P. Sidhu and T.K. Leung. Formal methods for protocol testing: A detailed study. *IEEE Transactions on Software Engineering*, 15(4):413–426, April 1989.

[55] D. Simser and R.E. Seviora. Supervision of real-time systems using optimistic path prediction and rollbacks. In *Proceedings of the Seventh International Symposium on Software Reliability Engineering (ISSRE '96)*, pages 340–349. IEEE, 1996.

[56] David Simser. Supervision of real-time systems using optimistic path prediction and rollbacks. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1, 1996.

[57] R. Swan. DMS-100 family evolution. *BNR Telesis*, 3(2):2–9, 1983.

[58] International Telegraph and Telephone Consultative Committee. *Functional Specification and Description Language, Recommendations Z.100-Z.104, (Blue Book).* International Telecommunication Union, Geneva, 1989.

[59] D. Toggweiler, J. Grabowski, and D. Hogrefe. Partial order simulation of sdl specifications. In R. Braek and A. Sarma, editors, *Proceedings of the 7th SDL Forum.* Elservier, 1995.

[60] Department of Electrical University of Waterloo and Computer Engineering. ECE 455 Software engineering course project, PBX hardware description. Version 1.2.

[61] Pierre Wolper and Patrice Godefroid. Partial-order methods for temporal verification. In Eike Best, editor, *4th International Conference on Concurrency Theory (CONCUR'93)*, pages 233 - 246, Hildesheim, Germany, August 1993. Apringer-Verlag.

[62] ITU-T Recommendation Z.100. *Specification and Description Language, SDL.* International Telecommunication Union, Geneva, 1992.

Appendix A

Target System Specification

This appendix contains a full specification for a private branch telephone exchange (PBX). The PBX consists of 60 telephones as shown in figure A.1. Each telephone is allocated a two digit telephone number. To simplify the system, inter-PBX calls are not allowed. A description of the PBX hardware can be found in [60].

The specifications that follow are for the control program of the PBX and are given in SDL/GR. Figure A.2 illustrates the system specification of the PBX as consisting of two types of processes, the *Phone_Handler* and *Resource_Manager*. The finite state machine specification of the *Phone_Handler* appears in figures A.3 and A.4. The *Net_Path_Manager* appears in figure A.5 and the *TTRX_Manager* in figure A.6.

The *Phone_Handler* is responsible for the behavior associated with both originating and terminating telephones. The *Resource_Manager* controls access to shared resources which are required for the duration of a call. Each of the 60 telephones in the PBX is allocated an individual *Phone_Handler* process.

Tables A.1 and A.2 supplement the SDL specifications by giving a textual de-

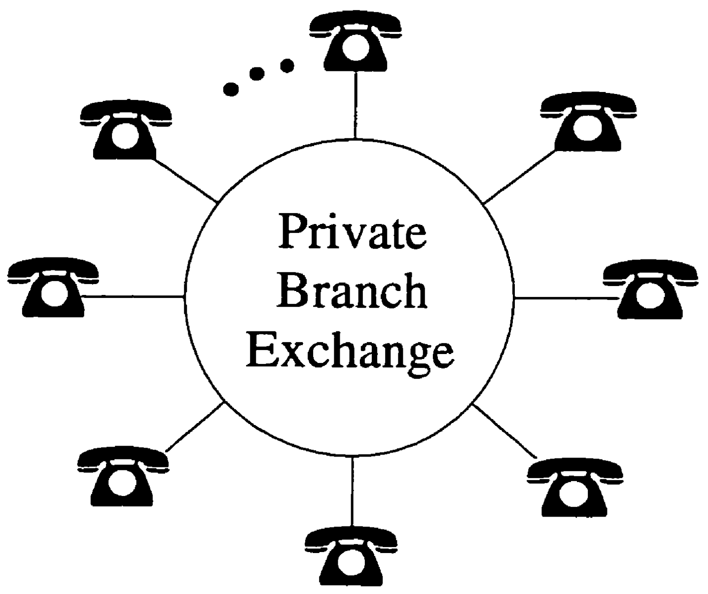


Figure A.1: Private Branch Exchange (PBX)

scription of all signals used. A brief, textual summary of each type of process follows.

A.1 Phone_Handler

The *Phone_Handler* process (figures A.3 and A.4) specifies all observable signal sequences of an originating telephone. *Phone_Handler* is responsible for obtaining all resources required for the duration of a telephone call in addition to establishing a voice path between the originator and terminator.

Signal, *CR_Con(x)* is assumed to be routed to the *Phone_Handler* process which has been assigned to the telephone whose directory number matches the dialed number. This is omitted from the SDL specification to limit specification size and complexity. Pairs of *Phone_Handler* processes corresponding to the originator and terminator of a telephone conversation communicate via *implicit* SDL signal routes. Conceptually, a bi-directional signal route exists between each pair of *Phone_Handler* process.

A.2 TTRX_Manager

The *TTRX_Manager* process (figure A.5) arbitrates allocation of touch tone receivers (TTRXs) which are required during dialing. Resources are requested and released by signals *Get_ttrx* and *Rel_ttrx* respectively. Similarly, resources are granted and indicated as not being available by signals *Grant_ttrx* and *NG_ttrx* respectively.

A.3 Network Path Manager (Net_Path_Manager)

The *Net_Path_Manager* process (figure A.6) arbitrates allocation of network paths though the exchange which are required for the duration of the call. Resources are requested and released by signals *Get_path* and *Rel_path* respectively. Similarly, resources are granted and indicated as not being available by signals *Grant_path* and *NG_path* respectively.

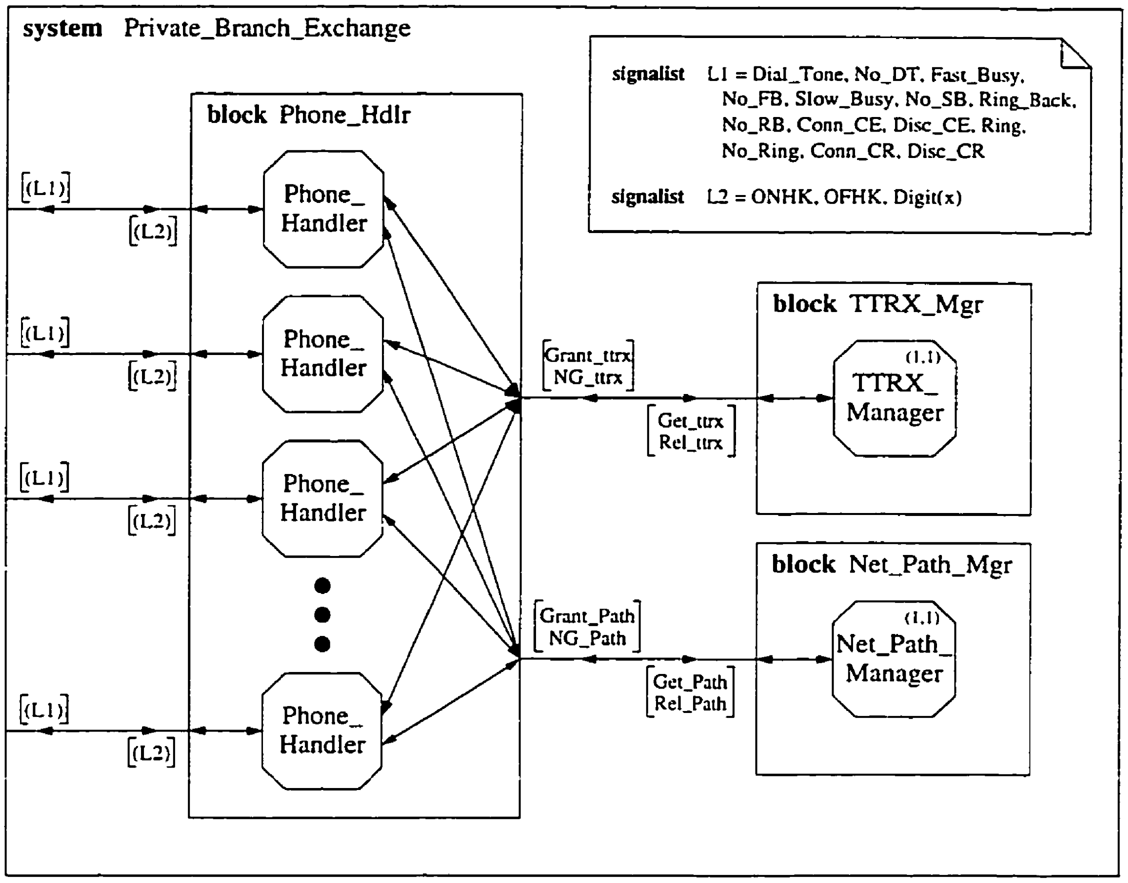


Figure A.2: System Specification of PBX

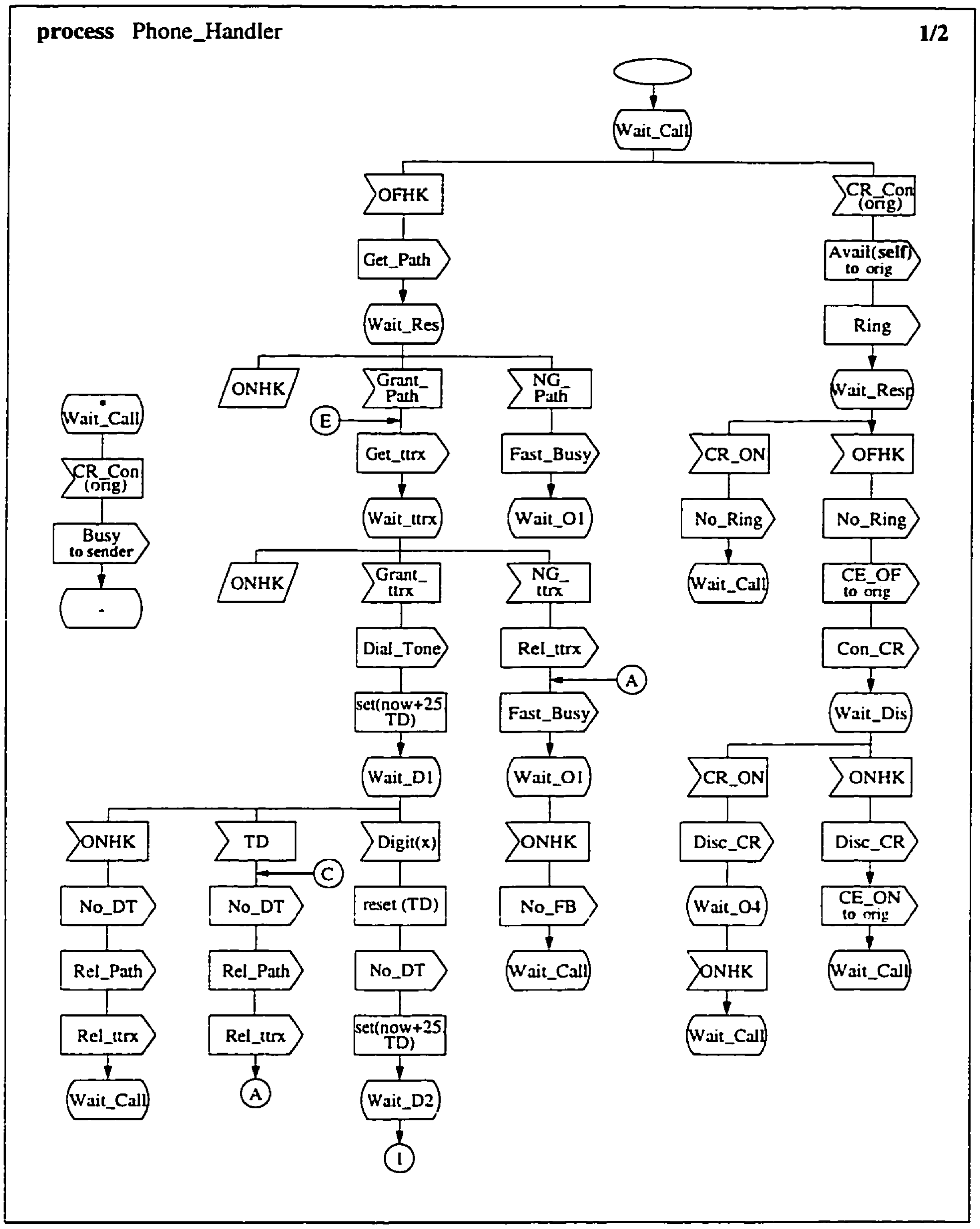


Figure A.3: SDL Specification of Phone_Handler Process (1/2)

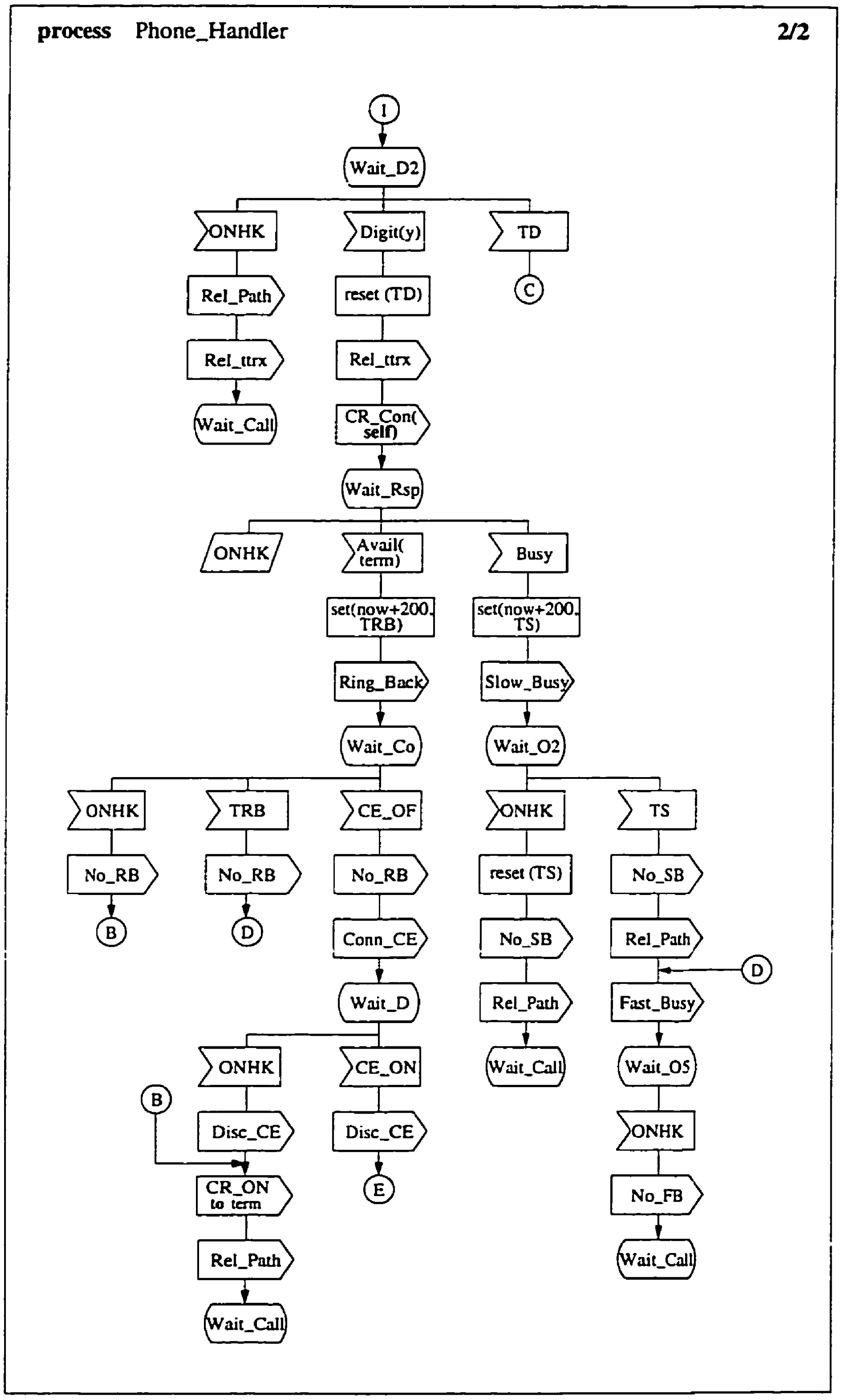


Figure A.4: SDL Specification of Phone_Handler Process (2/2)

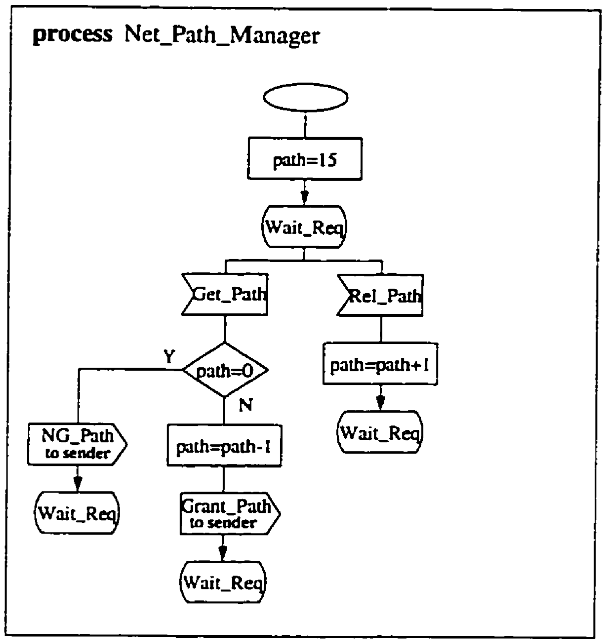


Figure A.5: SDL Specification of Net_Path_Manager Process

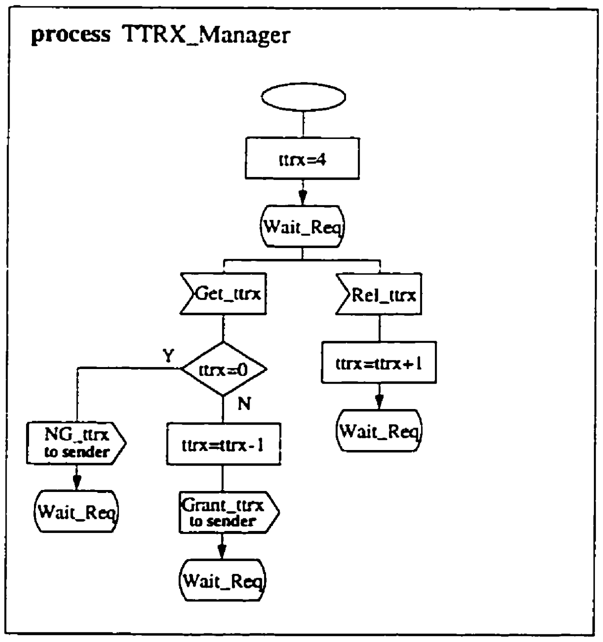


Figure A.6: SDL Specification of TTRX_Manager Process

Signal	Source	Destination	Description
Digit(x)	environment	Phone_Handler	User dialed digit number x
OFHK	environment	Phone_Handler	User has taken originating telephone offhook
ONHK	environment	Phone_Handler	User has placed originating telephone onhook
TD	timer	Phone_Handler	Digit dialing timer
TS	timer	Phone_Handler	Slow Busy tone timer
TRB	timer	Phone_Handler	Ring Back tone timer
Dial_Tone	Phone_Handler	environment	PBX provides user with dial tone
No_DT	Phone_Handler	environment	PBX removes dial tone
Fast_Busy	Phone_Handler	environment	PBX provides user with fast busy tone
No_FB	Phone_Handler	environment	PBX removes fast busy tone
Slow_Busy	Phone_Handler	environment	PBX provides user with slow busy tone
No_SB	Phone_Handler	environment	PBX removes slow busy tone
Ring_Back	Phone_Handler	environment	PBX provides user with ring-back tone
No_RB	Phone_Handler	environment	PBX removes ring-back tone
Conn_CE	Phone_Handler	environment	Assign voice connection from caller to callee
Disc_CE	Phone_Handler	environment	Deallocate voice connection from caller to callee
Disc_CE	Phone_Handler	environment	Deallocate voice connection from caller to callee

Table A.1: SDL Requirements Dictionary (1/2)

Signal	Source	Destination	Description
Get_ttrx	Phone_Handler	TTRX_Manager	Acquire touch tone receiver (if available)
Rel_ttrx	Phone_Handler	TTRX_Manager	Release acquired touch tone receiver
Grant_ttrx	TTRX_Manager	Phone_Handler	Grant an available touch tone receiver
NG_ttrx	TTRX_Manager	Phone_Handler	Requested touch tone receiver is not available
Get_Path	Phone_Handler	Net_Path_Manager	Acquire network path (if available)
Rel_Path	Phone_Handler	Net_Path_Manager	Release acquired network path
Grant_Path	Net_Path_Manager	Phone_Handler	Grant an available network path
NG_Path	Net_Path_Manager	Phone_Handler	Requested network path is not available
OFHK	environment	Phone_Handler	User has taken terminating telephone offhook
ONHK	environment	Phone_Handler	User has placed terminating telephone onhook
Ring	Phone_Handler	environment	Onhook callee telephone is started ringing
No_Ring	Phone_Handler	environment	Terminating telephone is stopped from ringing
Conn_CR	Phone_Handler	environment	Assign voice connection from callee to caller
Disc_CR	Phone_Handler	environment	Deallocate voice connection from caller to callee
CE_ON	Phone_Handler	Phone_Handler	Terminating party has gone onhook
CE_OF	Phone_Handler	Phone_Handler	Terminating party has gone offhook
Busy	Phone_Handler	Phone_Handler	Requested terminating party telephone is offhook
Avail(x)	Phone_Handler	Phone_Handler	Terminating party's telephone is onhook and its handling process PID is x
CR_ON	Phone_Handler	Phone_Handler	Originator has gone onhook
CR_Con(x)	Phone_Handler	Phone_Handler	Call request to terminator from originator with PID x

Table A.2: SDL Requirements Dictionary (2/2)