
Fuzzy Belief-Based Supervision

by Alexandre Vorobiev

A thesis presented to the University of Waterloo
in fulfillment of the thesis requirement
for the degree of Master of Applied Science
in Electrical Engineering

Waterloo, Ontario, Canada, 1997

©1997 Alexandre Vorobiev



**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-30582-1

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

This thesis presents a new approach to automatic failure detection (supervision) of session-oriented, real-time software systems. The system being supervised is assumed to be specified in a formalism based on communicating extended finite state machines such as ITU-T SDL. The presented approach is a significant refinement of an existing belief-based supervision approach. Its novelty lies in the association of a feasibility factor with individual hypotheses about the state of the target software system. The feasibility changes over time according to the closeness of the hypothesis to the observed behaviour. The competition algorithms presented in the thesis decide which hypotheses are left and which are discarded. The approach allows for a continuous supervision, capable of resynchronization with the target system following occurrences of failures.

After a description of the approach, an experimental evaluation of the research results is presented. The target system in the evaluation was the control program of a small telephone exchange. Both the simulated exchange and the supervisor executed on a UNIX workstation.

Acknowledgments

I would like to express my deep gratitude to my supervisor, Professor R. E. Seviora, whose assertive, yet kind supervision and guidance inspired, encouraged and enhanced this research effort.

I would like also to thank my family and friends for their patience, support and trust in me.

I also thank my readers, Professor J.A.Fields and Professor P. Dasiewicz for their suggestions and comments.

This research has been funded by a joint research grant from Bell Canada and NCRC.

Contents

CHAPTER I	<i>Introduction</i>	<i>1</i>
	1.1 <i>Software Supervision</i>	<i>2</i>
	1.2 <i>Belief-Based Software Supervision.</i>	<i>3</i>
	1.3 <i>Scope of the Research.</i>	<i>4</i>
	1.4 <i>Thesis Objective</i>	<i>4</i>
	1.5 <i>Research Contributions</i>	<i>5</i>
	1.6 <i>Related Work</i>	<i>5</i>
	1.7 <i>Thesis Organization</i>	<i>8</i>
CHAPTER II	<i>Belief-Based Software Supervision</i>	<i>9</i>
	2.1 <i>Non-determinisms and Belief-Based Software Supervision</i>	<i>10</i>
	2.1.1 <i>Non-Determinisms of Signal Detection in Software Supervision</i>	<i>12</i>
	2.1.1.1 <i>Signal Detection.</i>	<i>13</i>

2.1.1.2	<i>Signal Permutations During Abstraction</i> . . .	15
2.1.2	<i>Specification Non-determinisms</i>	16
2.1.2.1	<i>SDL hierarchy</i>	17
2.1.2.2	<i>Constructs of SDL Processes</i>	18
2.1.2.3	<i>Dynamic Semantics of SDL</i>	19
2.1.2.4	<i>SDL Non-determinisms</i>	20
2.2	<i>Belief-Based Supervision</i>	21
2.2.1	<i>BSDL Abstract Machine</i>	23
2.2.2	<i>Non-determinisms and beliefs creation</i>	24
2.2.2.1	<i>Signal detection nondeterminism</i>	24
2.2.2.2	<i>Spontaneous transitions</i>	24
2.2.2.3	<i>Arbitrary decisions</i>	25
2.2.2.4	<i>Concurrence</i>	25
2.2.2.5	<i>Non-deterministic channel delays</i>	25
2.2.3	<i>Termination of beliefs</i>	26
2.3	<i>Critical Evaluation</i>	28

CHAPTER III	<i>Fuzzy Belief-Based Software Supervision</i>	29
3.1	<i>Informal Introduction</i>	30
3.1.1	<i>Benefits of the Approach</i>	31
3.1.2	<i>Limitations</i>	32
3.2	<i>Generic Framework for Belief-based Supervision</i>	33
3.2.1	<i>Original Belief-Based Supervision From the Point of View of Generic Framework</i>	34
3.2.2	<i>Fuzzy Belief-Based Software Supervision</i>	34
3.3	<i>Feasibility Factor</i>	36
3.3.1	<i>Inheritance Rules for Feasibility Factor</i>	36
3.3.2	<i>Belief Set Combination and Resulting Feasibility</i>	37
3.3.3	<i>Failure in Fuzzy Belief-Based Supervision</i>	39
3.3.4	<i>Matching and Change of Feasibility</i>	39
3.4	<i>Matching</i>	44

3.4.1	<i>Output Signals</i>	44
3.4.2	<i>Behavior Representation and Capture: Output History</i>	45
3.4.3	<i>Performance Time-Outs of Signals in BSDL Model</i>	46
3.4.4	<i>Timing Out of the Observed Output Signals</i>	48
3.4.5	<i>Matching Individual Signals</i>	49
3.4.6	<i>Timing Aspect of Individual Signal Matching</i>	51
3.4.7	<i>Decomposition of Output History Matching Into Channel Matching</i>	52
3.4.8	<i>Matching Same-Channel Output Histories</i>	55
3.4.9	<i>Matching of Histories With Multiple Channels</i>	63
3.4.10	<i>Matching Behaviors of Multiple Belief Sets</i>	64
3.4.11	<i>Combined Matching Algorithm for Fuzzy Supervisor</i>	67
3.5	<i>Competition Algorithm</i>	67
3.5.1	<i>Cutoff Threshold Algorithm</i>	68
3.5.2	<i>On Necessity of a Variable Cutoff Threshold</i>	69
3.5.3	<i>Calculation of Cutoff Threshold Value</i>	70
3.6	<i>Fuzzy Supervisor Reports</i>	73
3.6.1	<i>Health of the Target System</i>	73
3.6.2	<i>Immediate Failure Report</i>	74
3.6.3	<i>History Tree and Underlying Fault Localization</i>	75
3.6.4	<i>Average Feasibility</i>	80
3.6.5	<i>Feasibility Distribution</i>	81
3.7	<i>Fuzzy Supervisor Operation</i>	83

CHAPTER IV

Experimental Evaluation of Fuzzy Belief Based Supervision **87**

4.1	<i>Evaluation Environment</i>	88
4.1.1	<i>PBX Hardware Emulator</i>	88

4.1.2	<i>PBX Control Software</i>	89
4.1.3	<i>Telephony Load Generator</i>	90
4.1.4	<i>Keyboard Input Driver</i>	90
4.1.5	<i>Abstracter Process</i>	90
4.1.6	<i>Display Driver</i>	91
4.1.7	<i>Fuzzy Supervisor</i>	91
4.2	<i>Evaluation Methodology</i>	95
4.2.1	<i>Variable Parameters</i>	96
4.2.1.1	<i>Telephony Traffic Simulation</i>	96
4.2.1.2	<i>Failure Seeding</i>	97
4.2.2	<i>Observed and Derived Characteristics</i>	98
4.3	<i>Evaluation Results</i>	98
4.3.1	<i>Results Summary</i>	99
4.3.2	<i>Supervisor Reports</i>	101
4.3.3	<i>Failure Detection Capability</i>	103
4.3.4	<i>Resynchronization and Continuous Supervision</i>	103
4.3.5	<i>Computational Complexity Observations</i>	104
4.3.6	<i>Other Supervisor Reports</i>	105
CHAPTER V	<i>Further Research and Conclusions</i>	107
5.1	<i>Further Research Directions</i>	107
5.2	<i>Concluding Observations</i>	108
APPENDIX A	<i>An SDL Specification of the Small PBX</i>	109
REFERENCES		121

List of Figures

Figure 1:	N-version programming	6
Figure 2:	Audits	6
Figure 3:	Software Supervision	10
Figure 4:	Example of Specification Layers in a Typical Telecom Specification ..	10
Figure 5:	Carrier Specification	11
Figure 6:	Detection of a Simple Signal	11
Figure 7:	Stimuli Transfer in Software Supervision	13
Figure 8:	Signal Abstraction	14
Figure 9:	Scenarios of Signal Detection	14
Figure 10:	A Simple SDL system	19
Figure 11:	SDL abstract machine	20
Figure 12:	Internal Organization of Belief-Based Software Supervisor	21
Figure 13:	Belief-Based Supervisor Abstract Machine	23
Figure 14:	Evolution of Supervisor State Upon Observation of Three External Signals	27
Figure 15:	Dynamics of Feasibility Factor of a Belief Set After One Mismatch and Three Matches	43
Figure 16:	Potential Life Intervals of Signals and Time Matching	52

Figure 17:	First-In-First-Out Signal Transmission in SDL Channels	53
Figure 18:	Abstraction of Same-Channel Signals in Supervisor	54
Figure 19:	Decomposition of Behavior Matching Into Same-Channel Matching	54
Figure 20:	Sequence of Attempted Matches	56
Figure 21:	Discarding Unmatched Prefixes and Matched Signals	57
Figure 22:	Expected and Observed Output Histories Before and After a Matching Sweep	58
Figure 23:	Matching Within a Belief-Based Supervisor	66
Figure 24:	Cutoff Thresholds	69
Figure 25:	Belief Sets Generation and Global Matching History	76
Figure 26:	History Tree Manipulation	78
Figure 27:	Average Health During a Period of Time T at times t0 and t1	80
Figure 28:	Reporting Feasibility Distribution	82
Figure 29:	Interactions of Processes In Experimental Environment	89
Figure 30:	High-Level OMT Diagram of the Fuzzy Supervisor	92
Figure 31:	OMT Diagram of the Feasibility-Related Data Structures of a Fuzzy Supervisor	94
Figure 32:	An Example: EOQs, CBSs, CBS Statistics Supervisor Statistics Objects in a Fuzzy Supervisor With Four Coexisting Belief Sets	95
Figure 33:	Fuzzy Supervisor Reports During Output Failures of the Target System	102

List of Tables

TABLE 1.	SDL Constructs	18
TABLE 2.	Experimental Results for Failure Detection Capability	99



I

Introduction

A dramatic increase of the size and complexity of software products on one hand and the growth of reliability-critical applications of software (such as navigation systems or telecommunications) have created a number of paradoxes. A more complex activity requires a more sophisticated software system to carry it out. But the bigger the system gets, the harder its testing becomes. For larger systems testing increasingly expensive and complete testing is technically impossible.

Various approaches have been suggested for lowering the risk of a failure of engineered systems. Faults can be prevented from getting into systems with the help of *rigorous engineering* and *fault avoidance* techniques. Damage from operating a system that possibly contains faults can also be minimized. *Fault-tolerance* techniques lower chances of system failure by isolating the potential failure to a limited area within the complex system (*failure isolation*), implementing critical fragments of the system with built-in *redundancy* of various degrees (as in *N-version programming*), and by the early detection of failures by *software audits* and *oracles*. These techniques are discussed in more details in section 1.6.

This thesis is concerned with automatic detection of software failures in real time. A software/hardware system that we are going to observe for the purpose of detection of its failures will be called the *target system*. Events that the target system obtains from its environment will be called *inputs*, and externally visible events that are directly caused by operation of the target system will be called target system's *outputs*.

1.1 Software Supervision

For a larger software system exhaustive testing is not possible due to an infinite or extremely large number of possible states of the system. If its failure-free operation is important, an approach supplemental to exhaustive testing called *software supervision* has been suggested in [1], [4]. Software supervision consists of real-time observation of the system's external inputs and outputs (i.e. its observable behavior), and judging the correctness of its behavior based on a model of the target system's expected behavior. The model of expected behavior used by the supervisor is derived from the target system requirements specifications.

The goal of software supervision is to detect the instances when the specification of the model can no longer explain the observed behavior of the target system, i.e. when a failure occurs.

Some simple systems could be adequately specified with a deterministic model. In this model, if a given sequence of inputs is introduced to the target system, there will be only one possible legal sequence of outputs.

But sometimes correct behavior may be not just a single one due to inherent non-determinism of target system specification. Non-determinism is an important feature of specification formalisms. It allows the specification writer to avoid stating irrelevant aspects of the behavior of the system being specified. Specification non-determinism, in turn, makes the task of software developer easier by permitting

choices of behavioral alternatives for the implementation which are less costly or otherwise desirable.

Non-determinism must be considered by the software supervisor. The supervisor should be able to accommodate the possibility that at any moment of time the target system may be in one of several states, all of which are permitted by the specified model. The Belief-based supervision is an approach that is based on simultaneous existence of multiple beliefs (or hypotheses) explaining the legal behavior of the target system. That allows exhaustive coverage of all possible behavior alternatives permitted by a non-deterministic specification.

1.2 Belief-Based Software Supervision

For a given set of inputs, each behavior permitted by specification represents a hypothesis (or in other words *belief*) about the possible current state of the target system.

Belief-based software supervision handles the supervision of non-deterministically specified systems through support of co-existence of multiple beliefs about the state of the target system. As the system operation goes on, new inputs and outputs are observed and the beliefs evolve accordingly: from time to time new beliefs are created with reception of new events, and older beliefs are terminated when the observed behavior invalidates the hypothesis they represent.

In the belief-based software supervision, failure is detected when all the hypotheses are invalidated and their corresponding beliefs are terminated. At this point no legal explanation of the observed behaviour remains

1.3 Scope of the Research

The boundaries of this research effort were as follows.

The research addresses belief-based, input-driven software supervision of software systems specified in formalisms based on communicating extended finite state machines. For concreteness, the UTU-T SDL specification formalism was used for application-level specifications [6]. Specifications of the external behaviour will be supplemented by a maximum response time constraints. In this work all maximum response times will be considered identical. The focus will be on telecommunication software systems.

Certain limitations will be imposed on the supported subset of SDL, as dictated by the current state of development of basic belief-based supervision theory.

A testbed simulating a small PBX with corresponding target system software will be used in the evaluation of research results. The simulation will be carried out in UNIX environment.

1.4 Thesis Objective

The thesis has addressed two main shortcomings of the basic belief-based supervision:

- termination of supervision at the moment of failure detection;
- no discrimination between the degree and specifics of the target system failures.

The main objectives of the thesis research were:

- to devise an approach allowing to resynchronize the supervisor model state with the state of the target system and continue supervision beyond the moment of detection of the failure without significant increase in computational complexity of software supervision;

- to develop a method to obtain information for localization of the fault causing the observed failure, mapped to the specification model of the target system;
- to provide a mechanism capable of characterization of the overall health of the target system;
- to evaluate the research results in a testbed with various types of failures and under various operation profiles of the target system.

1.5 Research Contributions

The main research contributions of this work are:

- definition of fuzzy software supervision approach;
- development of the core algorithms for this approach: failure-tolerant multi-channel signal matching, fuzzy belief generation and handling;
- evaluation of the proposed approach in an experimental fuzzy supervision of a control program of a small PBX.

1.6 Related Work

This section overviews several of the techniques mentioned earlier in this chapter.

N-version programming [7] is a collection of approaches that were suggested to eliminate software and hardware failures by utilizing redundant versions of the same system or components and providing a decision-making mechanism that selected which of the system's behavior is to be produced as a result. The redundant systems were built with different degrees of implementation independence including different hardware base, different development teams or independently produced and different components used in the systems. The idea behind N-version programming is that independent production of redundant systems will reduce the risk of similar faults and errors in different systems and decision making algorithm will be able to select

the correct behavior among several possibilities by a voting mechanism. This approach was proven to be practical for certain applications (e.g. numeric calculations such as ballistics), but redundancy makes its use increasingly expensive for complex systems, where replication means massive spending. At the same time when different software is used, this approach is difficult to use for systems involved in complex data/hardware manipulation. Schematic architecture of a system deploying N-version Programming technique is shown in Figure 1 on page 6.

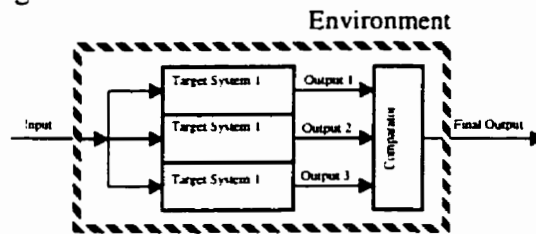


Figure 1: N-version programming

Software Audits [8] are usually a part of the target system dedicated to monitoring of one simple aspect of the target system's internal state. When an inconsistency of data is detected, it is reported and corrective action may be taken, e.g. when redundant data is available, an attempt to restore the correct state can be undertaken or system restart may be performed.

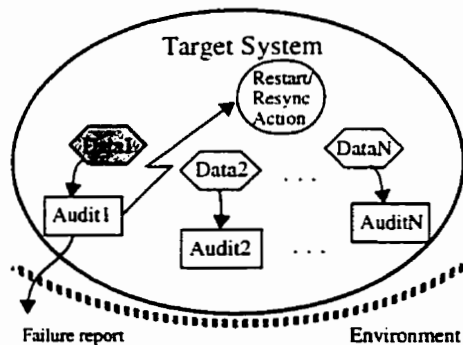


Figure 2: Audits

Audits are a part of the target system, therefore they are not supposed to be modified by the customer. Their scope of error detection and recovery is limited. If it expands, audits become complex and less reliable, which affects reliability of the target system as a whole.

Recovery blocks [9] have similar problems. The successful operation of recovery block relies on the acceptance mechanism, that might be faulty as well.

There are other approaches related to *Software Supervision*: observers[18] and multiple observers [19]. The latter two approaches has been are the closest to software supervision, with the major difference being the capability for handling of non-determinisms by belief-based software supervision approach. Observers approaches do not explicitly deal with non-deterministic specifications.

Many interesting flavors of software supervision have been developed by students and researchers in the Bell Software Reliability Lab. Some of these are:

- look-ahead supervision: supervision is carried out with an observational delay, that allows to select only these alternatives of non-deterministic behavior that correspond to the following observed behavior. This significantly reduces computational complexity of supervision ([26]);
- supervision with resynchronization: several approaches have been provided to allow for resynchronization of the supervisor after an occurrence of the failure based on a set of rules, and rollback mechanism ([22],[5]);
- supervision with failure retraction: the supervisor is equipped with a mechanism of taking over the control of an aspect of the target system in which the latter failed. The control is returned later when some pre-determined state is reached by the supervisor and the target system is also pushed in that state.([23]);
- grey-box supervision: the target system state is made partially visible to the supervisor. This allows to reduce the number of behavior scenarios that are considered simultaneously, thus reducing the computational complexity of supervision([25]);
- reduced specification supervision: only a subset of the target system's functionality is captured by the supervisor model. The model is smaller and that reduces computational complexity ([21]).

Please refer to corresponding research papers and BSR technical reports for more details.

A similar in the spirit, but different in the core, an Autonomous Evolutionary Systems approach has been suggested in [20].

A good general reference to fuzzy applications in engineering can be found in [16], [17].

1.7 Thesis Organization

The thesis is organized as follows:

Chapter 2 presents non-determinisms and their classification, general theory of software supervision and discusses different flavors of software supervision.

Chapter 3 presents the concept of fuzzy software supervision and describes algorithms developed for it.

Chapter 4 overviews the experimental implementation of a fuzzy software supervisor and the results and analysis of experimental evaluation of the fuzzy software supervision.

Chapter 5 summarizes this thesis and outlines suggested directions for the future research.

II

Belief-Based Software Supervision

Software supervision is complementary to software testing. Supervisor observes the behavior of a software system (inputs from and outputs to the environment) and produces a report when a deviation from the expected behavior is noticed. It discovers software failures before their cumulative effects result in major problems, reducing the harmful consequences of fully-developed software failure. Software Supervisor is non-intrusive, therefore its operation will not directly affect the operation of the target system itself.

In the schema of software supervision proposed in [4] a supervisor is separated from the target system. A generalized architecture of software supervisor is shown in the Figure 3 on page 10. Inputs from the environment, that are going into the target system are observed and directed to the supervisor, as well as the outputs of the target system.

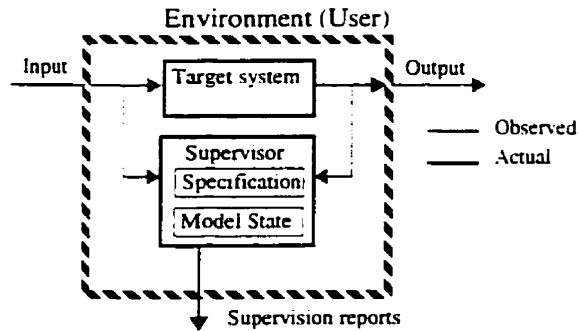


Figure 3: Software Supervision

2.1 Non-determinisms and Belief-Based Software Supervision

To supervise software we have to judge on the correctness of behavior of the target system with respect to some specification. Specification may consist of several layers.

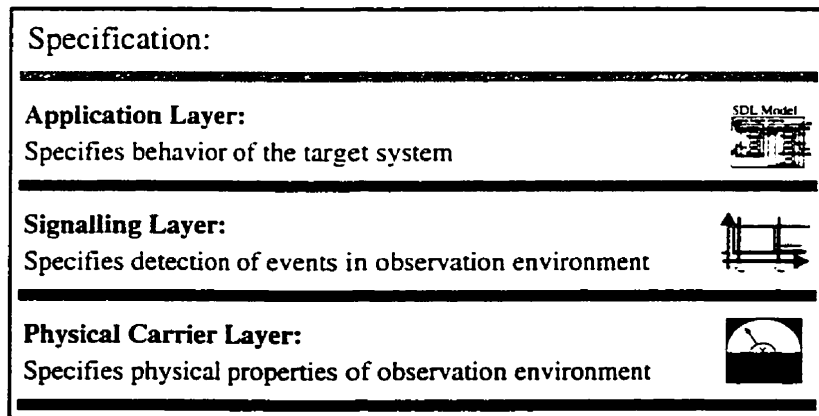


Figure 4: Example of Specification Layers in a Typical Telecom Specification

Physical Carrier Layer will specify the boundaries of observable changes for the target system and its environment. The carrier of the changes may be a variety of things, such as voltage in the line, data structure in the shared memory, data bus, etc. This specification may be thus defining the electrical characteristics of the carrier, frequency bands, etc. Figure 5 is an example of such a specification that defines the minimal voltage level of the physical carrier that is considered a change of the carrier. (In this case signals would be encoded by changing voltage of the carrier.)

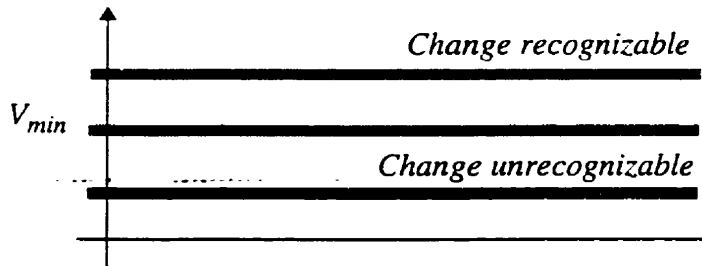


Figure 5: Carrier Specification

Signalling Layer of specification will be responsible for distinguishing events from non-events. As an example, Figure 6 illustrates a specification that sets timing constraints on detection of a signal.

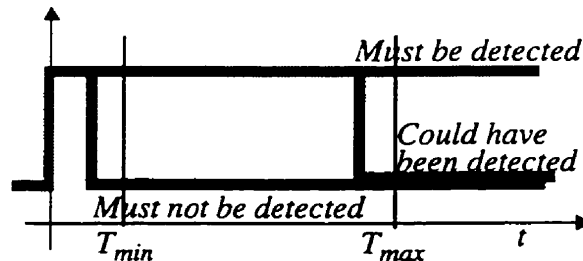


Figure 6: Detection of a Simple Signal

Figure 6 shows three scenarios. In the first change was observed on the carrier for longer than T_{max} and specification tells that this signal would have been observed by the target system. In

the second scenario change was present on the carrier for a time less than T_{min} , and according to the specification this signal would not be observed by the target system. In the third scenario, change was present between T_{min} and T_{max} , so the signal was either observed by the target system or was not. This example shows a non-deterministic signalling specification.

Application Layer of specification is used to specify the correct behavior. Behavior is comprised of events independently occurring in the environment of the target system (inputs), and events that are produced by the target system (outputs), possibly as a response to the input events.

As we discussed in the introduction chapter, specification formalisms support non-determinisms in order to enrich specifiable behavior. An example of such a specification formalism is SDL, a formalism based on communicating extended finite state machines, that will be described later in this chapter. A non-deterministic specification may result in multiple legal yet different behaviors. Software supervisor, therefore, should be able to accommodate multiple alternative behaviors allowed by the non-deterministic specification.

2.1.1 Non-Determinisms of Signal Detection in Software Supervision

In Figure 3 the structure of the Target-Supervisor couple is refined. The black arrows represent signals as they are produced and the grey arrows stand for signals as they are received. Target system receives signals from the environment as does the supervisor. The supervisor also receives signals produced by the target system, as does the environment. The mechanism of the transfer of signals between the environment, the Target System and the Supervisor is expanded in Figure 7, for cases when the

supervisor observes inputs and outputs of the target system at the physical layer level.

2.1.1.1 Signal Detection

When a signal is sent from the environment to the target system, it has to be detected by some scanning algorithm, that converts the signal from its initial form (e.g. analog) into an abstract (symbolic) form (see Figure 8).

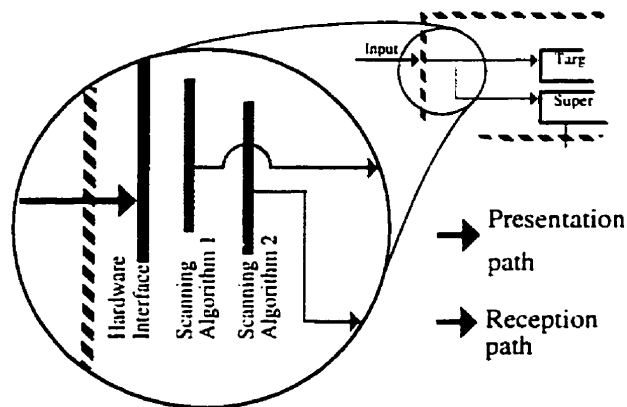


Figure 7: Stimuli Transfer in Software Supervision

A physical signal's appearance on its carrier occurs in time and space. It may be transformed into an abstract signal successfully, can be lost, or can be misinterpreted by the scanner due to too short a duration of occurrence, or noise, or specifics of the scanning algorithm, or other factors. Non-determinisms of the signal

detection are captured in signalling and physical carrier layers of specification.

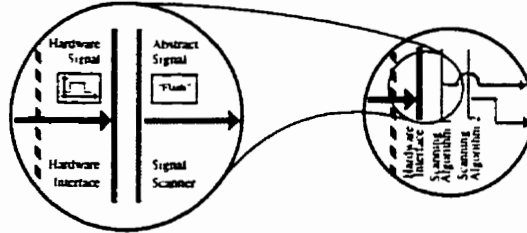


Figure 8: Signal Abstraction

In Figure 9 a sample signal is encoded as a series of three consecutive impulses. In the first scenario signal is lost due to an insufficient duration of signal presentation, in the second signal is properly detected.

The same figure can be used to illustrate another problem with the signal detection, namely, the discrimination of different signals. Say, if we have a two-peak encoding representing signal A and three peak encoding representing signal B, then the sample observation in the diagram “Signal Misinterpreted” could represent Signal A or signal B, that had been withdrawn before three peaks were produced. The situation presented in Figure 9 could be encountered in telephony, for example, when different ringing patterns are used on the same line.

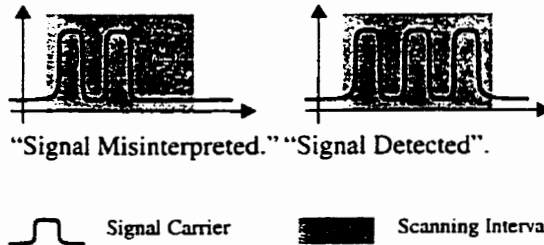


Figure 9: Scenarios of Signal Detection

For the pulse-encoded signals similar to ones shown in Figure 9 we have the following problem. Having the supervisor and the target system as two entities with a single complex signal presented from the environment we are facing a number of possible mutual interpretations of the same stimulus equal to $(M + 1)^2$, where M is the number of possible (all wrong, except for one proper interpretation) interpretations of the signal. One is added for the cases when no interpretation is given (signal is lost). We assume that during its input signal abstraction the supervisor misinterprets in the same way as the target system does.

Non-determinism of signal detection mentioned here is a separate topic and will not be considered any deeper in this work.

2.1.1.2 Signal Permutations During Abstraction

Now, let us consider a situation when not one, but several signals are sent from the environment within a short period of time. Traditionally arrivals of the signals would be sequenced for analysis. Then in addition to the already described possible distortions (namely, loss and misinterpretation of separate signals) we should also consider distortions of signals detection sequences in the supervisor: signals in the supervisor's detection sequence will be a permutation with respect to the detection sequence of the target system. (Such permutations, though, do not necessarily concern us, e.g. when permuted signals belong to unrelated sources or travel over different signal media.) Permutation occurs due to independence of signal abstraction activities of the supervisor and the target system. The number of such permutations grows rapidly with an increase of the considered signal detection sequence, making the task of keeping the supervisor "in sync" with the target system unmanageable, if the ordering of the observed inputs is the only criterion for judgement of correctness of the observed outputs.

We can clearly see here, that it is impossible to judge the target system's behavior only on the basis of the ordering of external observations.

This illustrates the hard task standing before the software supervisor - not only be able to verify the operation of the target system, but also be flexible enough to compensate for possible observation-time distortions of the real behavior of the target system due to the factors named above and do so while maintaining ability to recognize faulty behavior.

2.1.2 Specification Non-determinisms

Communicating Extended Finite State Machine (CEFSM) provide an approach for specification of a target system. CEFSMs are particularly suited for complex, distributed target systems such as the ones found in the area of telecommunications.

The following is a formal definition of an CEFSM extended with variables, conditions and communications ([14], [15]):

CEFSM is a tuple $\{I, O, S, s_0, D, C, T, R\}$ in which:

- I : Set of input signals
- O : Set of output signals
- S : Set of states
- s_0 : Initial state, $s_0 \in S$
- D : Some n-dimensional state, $D_1 \times D_2 \times \dots \times D_n$
- C : Set of conditions $C_i | D \rightarrow \{0, 1\}$
- T : Set of tasks, $T_i | D \rightarrow D$
- R : transition relation, $S | S \times C \times I \rightarrow S \times T \times O$

As it was mentioned in the introduction, in addition to the CEF-SM specification, we will specify the maximum time required for the target system to produce a response on an external stimulus. For the sake of conceptual simplicity in this work we will consider only cases in which the maximum response time is the same for all signals.

SDL is an example of a specification formalism based on CEF-SM. “SDL” stands for Software Specification and Description Language, a formalism introduced by International Telecommunication Union (ITU) (formerly CCITT) [12]. SDL visualizes CEF-SM-based specification as it has two equivalent forms of notation: graphical and textual, that makes the language both computer- and human-friendly. The examples in the thesis will use graphical SDL notation.

2.1.2.1 SDL hierarchy

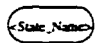
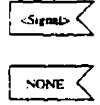
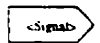

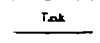
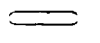
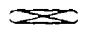
SDL specification establishes a tree-like hierarchy of the following entities:

- system - delimits the system that is being specified from the environment. Receives signals from the environment and passes signals to the boundary via signal channels
- subsystem - is statically defined within a system. does not intersect with other subsystems. Signals are received and passed to subsystem’s boundary via signal channels.
- block - statically defined partition of a system or subsystem. Signals are received from and passed to its boundary via signal routes.
- process - belongs to a block, has a certain type, may be created and destroyed dynamically. Sends and receives signals via signal routes
- signal channel - links subsystems and blocks to the boundary of the system (environment). Passes signals to the associated internal signal channels of subsystems and associated internal signal routes of blocks
- signal route - links the processes belonging to a block to each other and to the signal channels outside of the block

2.1.2.2 Constructs of SDL Processes

SDL process is an extended communicating state machine placed within an SDL block. The following table summarizes the elements composing the finite state machine.

TABLE 1. SDL Constructs

Construct	Purpose
	State construct
	<p>Transition trigger construct: selects the branch immediately following it. There are two variants of the transitions:</p> <p>Deterministic: fires when the specified signal is received.</p> <p>Spontaneous transition: when "NONE" is used instead of the signal name within the construct, the following branch can be taken without receiving any signals</p>
	Send construct: the specified signal is sent via some signal route (specified explicitly or implicitly) with specified parameters
	<p>Decision construct: a branch is selected that corresponds to the value contained within the decision symbol</p> <p>If "ANY" is used instead of the value, any one of the choice branches can be taken arbitrarily.</p>
	<p>Task construct: may contain assignments and timer operations.</p> <p>Timers is an SDL concept that allows to produce a named signal after a specified interval of time. Timers can be set and reset within the task construct.</p>
	Start construct
	Stop symbol: process is terminated when a stop symbol is reached.

An example of a simple SDL system is given in Figure 10.

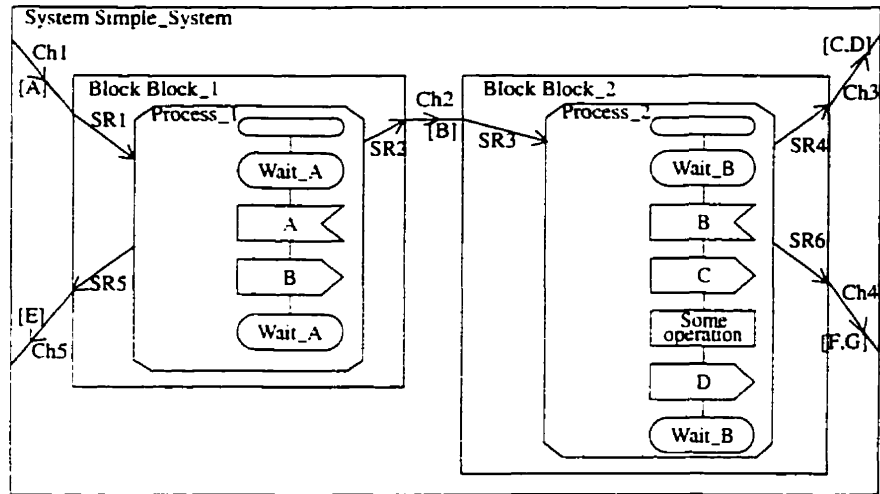


Figure 10: A Simple SDL system

2.1.2.3 Dynamic Semantics of SDL

SDL specifications are executable. Rules of execution, or dynamic semantics, of the SDL specifications are defined operationally by means of definition of SDL abstract machine.

SDL abstract machine is defined in META-VI as six synchronously communicating processes:

- *system*: a unique process that creates and removes SDL processes, performs signal routing.
- *input-port*: receives the signals sent to its SDL process and stores them in a queue.
- *path*: responsible for support of non-deterministic signal delivery delays in channels.
- *view*: handles allowed visibility of process variables to other processes
- *sdl-process*: interpreter of SDL process.
- *timer*: handles time-related aspects of SDL abstract machine: tracks time and time-outs.

The following diagram represents the abstract machine and shows interactions between its components.

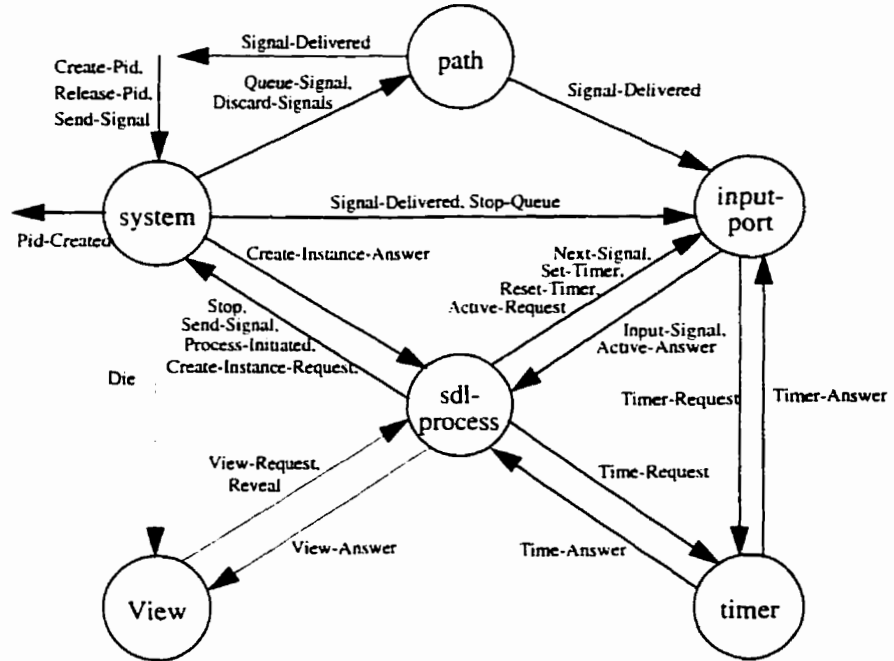


Figure 11: SDL abstract machine

Refer to SDL literature ([6],[12]) for particular details of SDL language description.

2.1.2.4 SDL Non-determinisms

The following are the non-determinisms supported in SDL [13].

- **Signal channel delays:** signals sent over signal a signal channel will be delivered after a non-deterministic delay. Sending order is preserved within the channel.
- **Concurrency:** order of arrival of signals produced at the same time and sent to the same destination point over different delaying paths cannot be predicted

- Spontaneous transitions: "NONE"-transition is chosen with no particular trigger, in non-deterministic fashion.
- Arbitrary decisions: selection of a path as a result of "ANY" decision is non-deterministic.

2.2 Belief-Based Supervision

The Belief-Based Supervision [1] (abbreviated as BBS in this work), was suggested to handle non-determinisms encountered in software supervision. The idea behind the theory of belief-based supervision is to create a set of hypotheses that would represent effects of a non-determinism encountered. The created hypotheses (or *beliefs* as they are called in BBS theory) are then compared with the observed behavior and discarded if they fail to explain the observed behavior. If at least one hypothesis that explains the observed behavior of the target system at any moment of operation of the target system exists, the behavior of the target system is considered correct.

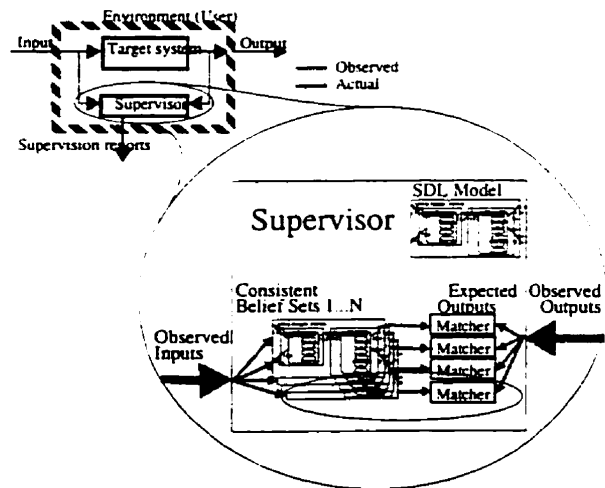


Figure 12: Internal Organization of Belief-Based Software Supervisor

Belief-based software supervision is an iterative process, continuing indefinitely along with operation of the target system. It may be driven by signals observed on the boundary of the target system (inputs, outputs, or both). Observed inputs are fed in the executable specification and every time when a non-deterministic construct of the specification is encountered, hypotheses are generated representing all behaviors that would become legitimate after execution of the given non-deterministic construct by the existing hypotheses.

Belief-Based Supervision theory, presented in [1], [4] has addressed non-determinisms that could be expressed in SDL specifications.

Within the supervisor the target system was modelled as an SDL specification, composed of blocks and processes interconnected with signal channels and signal routes.

Every belief was representing a hypothesis about the state of a certain SDL process from the specification: state of the CEFSM (values of timers, variables, state of FSM), and a set of signals that are thought to be “in transition” - sent to the process, but not received by it, and travelling somewhere in signal routes and channels. Introduction of “in transit” sets was justified by possibility of non-deterministic delays of signals on their way from a sender to the receiver in permitted in SDL formalism. Algorithms described below will clarify use of “in transit” sets.

Similarly to the composition of blocks and processes into an SDL system, beliefs representing complimentary (non-contradictory) hypotheses about different processes were grouped into *belief sets*.

Similarly to SDL definition, rules of execution of the belief-based supervisor model were defined by BSDL abstract machine.

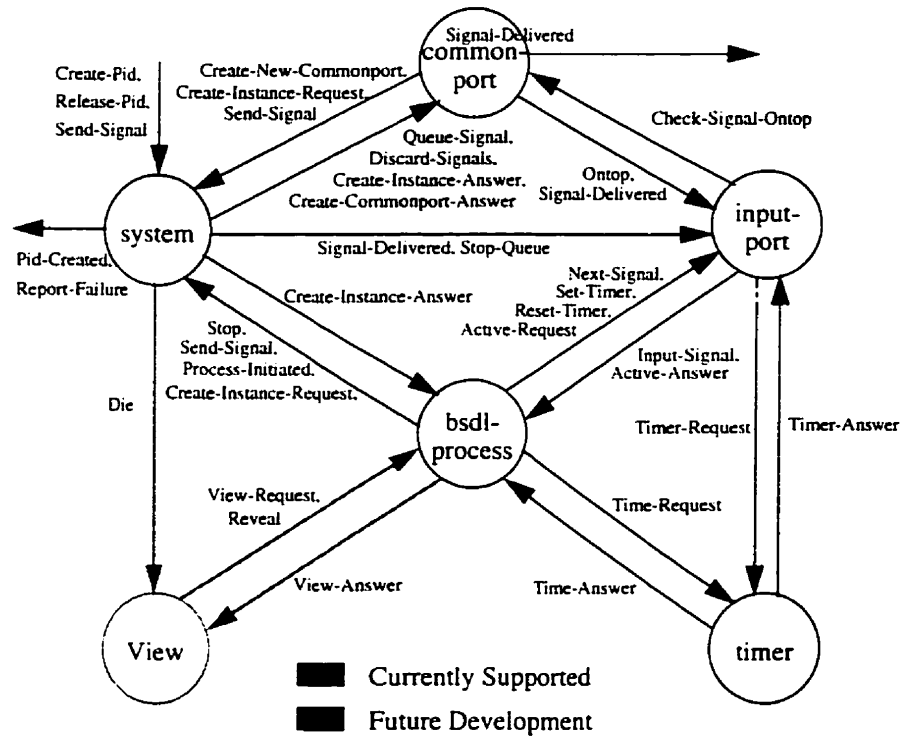


Figure 13: Belief-Based Supervisor Abstract Machine

2.2.1 BSDL Abstract Machine

The model used in software supervisor is interpreted. The interpreter implements the belief-extended SDL abstract machine (shown in Figure 13).

The responsibilities of the components depicted in Figure 13 are:

- **System**: a unique instance of a process that handles interactions with environment, creates beliefs, maintains consistent belief sets,
- **Common port**: responsible for generation of all permissible signal permutations of signals coming to a particular input port.

- Input port: keeps track of signals sent to process and timers set in process.
- BSDL process: represents the state of a target system as a state of an SDL diagram. Keeps track of variables, timing. Sends and receives signals.
- Timer: tracks time
- View: shown for completeness only. Present in SDL abstract machine, but not implemented in Supervisor.

2.2.2 Non-determinisms and beliefs creation

2.2.2.1 Signal detection nondeterminism

This is a signalling layer specification non-determinism. As we saw, depending on sending conditions and timing, signal can be detected, missed, or misinterpreted by the target system. Belief-based supervision deals successfully with the first two possibilities, creating two beliefs for every input signal observed by the supervisor. In one signal is successfully detected, and the second is that it is not detected by the target system. We will not consider the third option in this work, as it is valid for a certain class of target systems and adds significantly to the complexity of the belief-based supervision theory.

2.2.2.2 Spontaneous transitions

When possible “NONE”-transition in a SDL process, a belief should be created in which the “NONE” transition has been fired.

2.2.2.3 Arbitrary decisions

When an “ANY” decision is encountered while executing an SDL process, new beliefs will be created, one per every branch of the “ANY” decision. In every of the newly created beliefs its corresponding branch of the “ANY” decision will be chosen as a result of the “ANY” decision.

2.2.2.4 Concurrency

Signals travelling over signal routes do not experience delays. This introduces possibility of concurrent arrival of signals. For example, when signals are sent to the same destination over two or more different signal routes at exactly the same moment of time, they would arrive immediately, and if all of these are expected in the same state, the resulting branch would be selected non-deterministically. Different beliefs have to be created to account for all possible permutations of signal consumption.

2.2.2.5 Non-deterministic channel delays

When a signal is sent over a signal channel during execution of an SDL specification, time of its arrival is non-deterministic. When some other signals were sent over different channels during execution of the specification, the order of arrival of these signal and the signal travelling over the signal channel may be different. This will cause multiple legitimate scenarios of signals arrival. Beliefs will be created to cover all of the legitimate permutations of signals arrival. Two beliefs are originally created: “signal has been delivered”, and “signal is still in transit”. In transit” state of signals is used to produce all possible permutations. Such state indicates that a signal has been sent to its destination through a delaying path but did not arrive yet. When other signals are sent to the same destination beliefs will be created in which arrival of the “in-transit” signal will be permuted with arrivals of other “in-transit” signals and the newly-sent sig-

nals in all possible ways, producing a new belief per every permutation. The current “in-transit” signals for a given destination are kept in an “in-transit” set that is unique for the given destination belief. Once “in-transit” signals are “successfully delivered” in a child belief, they are removed from the “in-transit set” of the given child belief. “In-transit” sets are handled by common ports.

2.2.3 Termination of beliefs

Some of the beliefs produced by the supervisor prove to be invalid in the course of supervision. Real state of target system is unique, and beliefs that cannot explain indications of the true state have to be terminated.

Also, when an additional specification is provided imposing a maximum response time of the target system, beliefs that have signals in “in-transit” state longer than the maximum response time limit would violate such a specification and should also be terminated.

Beliefs are terminated under the following circumstances:

- If a belief has been terminated, all belief sets it belonged to are terminated also.
- If all sets a belief belonged to has been terminated, a belief is terminated also.
- If an output signal unexpected by a belief has been observed in the target system, a belief is terminated.
- If a signal expected by a belief has not been observed in maximum response time, a belief is terminated (performance time-out).
- If two beliefs have been found to be identical (so that the states, variables values, timers, signals in transit, etc. are identical in both beliefs), one of the two will be terminated and the surviving belief will be included in the belief sets of the original two beliefs.

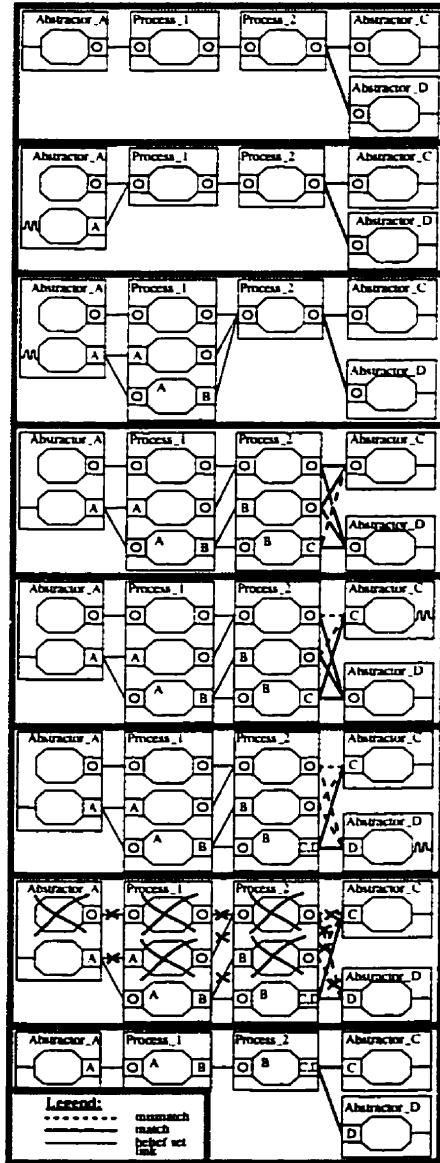


Figure 14: Evolution of Supervisor State Upon Observation of Three External Signals

Example: Figure 14 represents evolution of software supervisor's beliefs occurring upon observation of three external signals. Refer to Figure 10 for a simplified SDL specification of the target system for this example.

The sample system contains two SDL blocks, each containing one process. Input signal "A" is observed on the target system, that is followed by output signals "C" and "D". All three signals are external with respect to the supervisor

2.3 Critical Evaluation

Having all the inputs of the target system and its outputs it has a wide picture, unavailable for N-version programming-based system or audits.

BBS is sound: supervision does not continue, if the target system has any deviations from the specified behavior within the precision of the matching algorithm. But soundness can be a drawback, as the described supervisor cannot continue operation beyond the discovery of a failure.

- Post-failure supervision is impossible: Signal loss or misinterpretation most will result in termination of all belief sets and a halt of supervision. And since the information about the target system resides in the collection of belief sets, no information will be left to carry on the supervision.
- No discrimination of degrees of failure severity: If we are supervising a telephone exchange, failure will be reported and supervision will be halted in both cases of a faulty line behavior and a massive shutdown. It would be of a great benefit, if there were means of determining the measure of the failure and continuing supervision if the failure is not significant.

III

Fuzzy Belief-Based Software Supervision

The theory of belief-based supervision presented in the previous chapter has a major drawback: it is coarse, purely qualitative, and not quantitative to any degree. What this means is: we are able to tell when the target system is behaving incorrectly, but cannot explain as to how much incorrect the behavior is or even why it is incorrect.

When a target system is complex enough, a partial failure may not affect most of the functionality of the system, while it will be classified as a failure by the belief-based software supervisor and supervision will be ceased without explanation of what exactly caused the supervisor to halt or how bad was the failure.

If the target system is perceived to be producing an unspecified behavior, all the “correct” belief sets will be killed as a result of mismatches between correct expected behavior and incorrect observed of the target system. The matching procedure labels every existing belief set with one of the two values, as either dead or alive. What

will happen if matching could produce a real number instead of just two values?

Further in this work we will suggest an extension to the Belief-Based method in order to increase supervisor's flexibility that will allow for signal detection non-determinisms in signal detection, would allow for continuous supervision of software with behavior not exactly compliant to the given specification and also allow for a natural resynchronization of the supervisor with the target system after a failure was detected. This approach will be introduced as an extension to and improvement of the Belief-Based method.

3.1 Informal Introduction

Let us imagine a belief-based supervisor, in which every currently existing belief set is supplied with some *feasibility factor* based on the history of the observed behavior and history of behavior of the given belief.

Every time when a deviation mismatch occurs between the observed behavior of the target system and the behavior expected by the given belief set, feasibility factor of the affected belief set would be decreased. Vice versa, when a "match" is successful, feasibility factor of the belief set would be increased.

Now we would have an opportunity to compare belief sets between each other and to select those which match the behavior of the target system most closely. Termination of belief sets will be determined by competition instead of single-time mismatch.

With such a setting, the well-being of the target system is reflected in the feasibility factors of all belief sets existing in the supervisor at the given moment. If a failure occurs, feasibility factors will be decreased, and if we keep the "correct" hypotheses around long enough, they will have a chance of improving their feasibility factors in the case of continued correct operation of the target system.

The described idea enhances the original BBS method: if some belief set was following the target system very closely, and an observational distortion discussed in section 2.1.1 occurs, or even some kind of a failure of the target system (such as an uncritical loss of a signal), the formerly successful belief set will not be terminated just because of that; instead, only its feasibility factor will be decreased, and if it is high enough after the decrease, and the consequent behavior of the target system is matching the hypothesis of the belief set, the set will have an opportunity to recover from the damage and will survive.

What can we do if a massive observational distortion or multiple faults or a failure of the target system occur during the supervision? To avoid an immediate death of the supervisor, we could decrease the termination threshold for the surviving belief sets, allowing a greater number of beliefs to outlive the moment of the failure and recover later, if the system manages to recover after the failure. We still could issue warnings about the malfunction, but the supervision will continue. To prevent the supervisor from being flooded with incorrect hypotheses we would allow only the fittest hypotheses will survive, and it will be up to us to decide how high should the “survival plank” be raised.

This opens up a possibility of continuous re-synchronization of a supervisor and a target system, along with continuous supervision.

The novelty is in an application of a dynamic threshold competition mechanism to the existing BBS supervision that can be used also in any other belief-based approach, where the state of the target system is estimated by a population of belief sets, or beliefs, or any set of hypotheses.

3.1.1 Benefits of the Approach

Once again, what does this approach promises us so far?

Optimistic supervision: when system deviates from the specified behavior, our supervisor will not cease the process of supervision due to exhaustion of hypotheses population. Sufficient number of beliefs will survive and supervisor will return to normal operation if the target system does.

Distortion-tolerant supervision: this approach will accommodate supervision of systems with “hard-to-discriminate” output signals (such as in example on Figure 9, section 2.1.1). Failure to detect such a signal, or a misinterpretation of a signal will only lower the feasibility factor of the belief set, without an immediate termination.

Adjustable supervision: by adjusting thresholds values we could control the precision of supervision. This way it is easy to control the trade-off between precision and durability of supervisor, since thresholds could be dynamically determined. Also, we could assign different decrement values to different types of discrepancies, that will differentiate the impact of discrepancies on the supervisor’s judgement of the target system operation.

3.1.2 Limitations

The more tolerant the supervisor becomes, the less precise its operation becomes. This would happen due to the fact that tolerance is achieved by imposing less stringent conditions on the survival of hypotheses, therefore allowing “incorrect” hypotheses survive along with “correct” ones. This feature can be used for an advantage or a disadvantage, depending on the goal of supervision, and should be tuned and used accordingly. Nevertheless, the supervision sensitivity can be made as close to sound supervision as needed, as we will show later.

3.2 Generic Framework for Belief-based Supervision

Below are the suggested general elements definition of which would establish a Competitive Hypothesis-Based Supervision for reactive systems:

- Hypotheses representation: For the target we will select a representation capturing hypotheses about the state of the target system.
- Feasibility factor: on every hypothesis we should introduce a factor reflecting the degree of correspondence of the hypothesis to the actually observed behavior of the target system.
- Hypotheses generation: we should define how externally observed stimuli of the target system will alter existing hypotheses and how new hypotheses will be generated and how the feasibility factor will be assigned to the new hypotheses.
- Matching algorithm: we should define how the observed behavior is matched to the hypothetical behavior and how the matching influences the values of the feasibility factors of hypotheses.
- Hypotheses termination rules (competition algorithm): we should define how the population of hypotheses is maintained on the basis of values of feasibility factors. Survival threshold function should be specified, that will determine the threshold of the feasibility factor sufficient for survival of a hypothesis. This threshold may be dependent on the overall feasibility factor values and on the number of surviving hypotheses.
- Evaluation rules: when does a supervisor classify the behavior of the target system as a malfunction and how does it continues the supervision after this.

No doubt, this is a very loose generalization, but this vagueness is intentional. This view will be general enough to be used for analysis of the BBS method and for building an extension to BBS method - a Fuzzy Belief-Based Supervisor.

3.2.1 Original Belief-Based Supervision From the Point of View of Generic Framework

- Hypotheses representation: belief sets (as described in section 3.2)
- Feasibility factor: $\chi_i \in \{0, 1\}$

The following is an interpretation of belief generation and termination rules of BBS according to the Generic Framework:

- Hypotheses Generation: belief generation rules as defined in section 3.2, page 33. The initial (empty) hypotheses has its feasibility factor value set to 1. All derived hypotheses upon creation have their feasibility factor values inherited from the parent hypotheses.
- Matching Algorithm: when a mismatch is detected in a belief set (see section 3.2 for explanation of mismatch), its feasibility factor is set to 0.
- Competition Algorithm: the threshold function is constant and equals to 1. Every belief set, that experienced a mismatch is terminated. Any deviation from specified behavior sets competition criterion of a belief set to 0.
- Evaluation rules: system is operational if belief set population is not empty. Once it becomes empty, supervisor classifies the target system as malfunctioning and terminates its operation.

3.2.2 Fuzzy Belief-Based Software Supervision

We shall build a supervision schema suitable for implementation, so that advantages described in section 3.1.1 will be realized.

The schema of software supervision suggested here is called “fuzzy” for the reasons for the resemblance of the competition algorithm to the fuzzy logic approach to decision making.

Fuzzy Belief-Based Supervision will be shown to be an extension of the original Belief-based supervision.

Here we follow the steps on page 34:

- Hypotheses representation: belief sets as in the original BBM
- Feasibility factor: $\chi_i \in [0, 1]$
(Note: it's an interval, not a two-value set.)
- Hypotheses generation: see belief and belief sets generation in the original BBM. in the beginning of supervision the initial "idle" Belief set has its feasibility factor set to 1, as in BBS. The derived belief sets inherit values of feasibility factors from their respective parents.
- Matching Algorithm: every mismatches is assigned some *mismatch value*, are when a mismatch is detected in a belief set (see section 3.2 for explanation of mismatch), its feasibility factor is decreased by a corresponding mismatch value. Similarly, all the occurrences of successful matching are classified and mapped to some *match value*, so that a feasibility factor of a belief set is increased accordingly when the given matching succeeds.
- Competition algorithm: the threshold may be a function of different arguments, such as a number of existing belief sets, or distribution of feasibility factors, or even a constant. Its value should be less or equal to 1. Different heuristics could be applied. When a feasibility factor of a belief set fall below the current threshold, it is terminated according to the rules of belief termination (see section 3.2, page 33).
- Evaluation rules: system is operational if belief set population is not empty. Supervisor will issue a warning every time when the maximal feasibility factor becomes close to the threshold, taking some action to avoid extermination of all hypotheses. When sufficiently many hypotheses have feasibility factors higher than the threshold it will be increased. Then at any time the evaluation of "soundness" of the target system will be reflected by the threshold and the distribution of feasibility factors of the belief sets.

The proposed outline of contains many heuristic components that require further theoretical support and empirical evaluation. In the

following sections we will expand the composite parts of the proposed Fuzzy Supervision approach and develop algorithms for it. Experimental fuzzy supervisor built for evaluation purposes will be described in the next chapter and the results of the experimental evaluation will be presented in the chapter following that.

The main objective will be to overcome termination of supervision after occurrence of failure within the target system (the observation of unspecified behavior). This represents a requirement formerly not imposed on belief-based supervision approach and will have to be addressed in the fuzzy supervision algorithms afresh.

3.3 Feasibility Factor

The key difference of the Fuzzy Belief-based supervision approach from the earlier-suggested belief-based supervision approaches is the presence of interval discriminator in every belief set, that represents correlation of behavior represented (or expected) by the given belief, and the observed behavior of the target system. This discriminator was called *feasibility* earlier in this work.

From now on, let us consider, that the greater the feasibility factor is, the closer did the observed behavior matched the expected behavior in the given belief, so that when the two behaviors match completely, feasibility factor will be equal to 1.

3.3.1 Inheritance Rules for Feasibility Factor

As new belief sets are generated by the fuzzy supervisor, their feasibility factors should be assigned some values.

As the newly-generated belief does not produce output at the instance of its creation, it will share expected and observed output history with its parent belief set.

Therefore, by our definition the value of its feasibility factor should be the same as in its parent's belief set.

$$\chi_{child} = \chi_{parent}$$

This sets the inheritance rule for the feasibility factor.

3.3.2 Belief Set Combination and Resulting Feasibility

Sometimes two or more different hypotheses have identical expectations of the observed behavior. This happens when difference between these hypotheses lies within the SDL model internal state and does not manifest itself externally. Such hypotheses may co-exist in the supervisor until their evolution makes their expected behavior different from each other and they become distinguishable externally. But evolution may also change their internal state, so that they will become identical internally and externally. In binary belief-based software supervision such belief sets were merged into one. Their expected behavior remained perfectly matched with the observed behavior.

This mechanism was important, as it eliminated unnecessary redundancy among the existing hypotheses while maintaining sufficient coverage of all state and behavior alternatives possible in the formal model of the target system.

Will such a mechanism be a necessary part of a fuzzy supervisor? Yes, as we plan to allow survival of some of hypotheses that do not describe the observed behavior particularly well. This increases possibility that different hypotheses will evolve into ones with identical internal state and expected behavior, therefore the original problem still remains: we have to merge them, as their consequent evolution will be identical.

To makes things more difficult, these may have different past histories and, as a result of that, different feasibility factors.

If we are to merge several into one, what should be different in the merging algorithm for the fuzzy supervisor?

Not much. The major difference between belief sets in the binary software supervisor and beliefs sets in fuzzy software supervisor is the presence of the feasibility factor and a track of past history, should we decide to keep the latter.

We suggest that the feasibility factor be used to decide attributes of which belief set we are going to inherit in the result of a merger. Feasibility reflects how well the associated belief could explain behavior of the target system in the past, by definition, and it is logical to chose this “better” past history as the attribute of the resulting belief set. Informally, this means, that we will inherit the best possible feasibility and history of explanation of the target system that a belief set with given internal state and expected behavior can have in the supervisor. The rest of attributes will be exactly the same. so merging actually will consist of selection of one belief set among several.

The new algorithm for the merger then looks like this:

```
1:  algorithm select_merge_result(merged_belief_sets)
2:      best_belief_set = head(merged_belief_sets);
3:      for (every belief_set in merged_belief_sets) {
4:          if (< ( feasibility(best_belief_set),
5:                feasibility(belief_set)))
6:              then {
7:                  best_belief_set = belief_set;
8:              }
9:      }
10:     return best_belief_set
10:  end select_merge_result
```

3.3.3 Failure in Fuzzy Belief-Based Supervision

Failure in basic belief-based supervision theory manifests itself in discarding of all existing beliefs in the supervisor. This could have been caused by explicit mismatch or performance time-out of signals in output queues or in-transit signal sets.

This simple algorithm of failure detection will not work in Fuzzy Belief-Based supervision, as none of those belief death causes will be valid in FBBS.

What would happen in the Fuzzy supervisor in the situation that would have been recognized as a failure by a binary belief-based supervisor? Should it be a mismatch or signal expiration, it would result in reduction of a feasibility factor of the affected belief set. And *failure* in terms of binary software supervision will be equivalent to a simultaneous occurrence of feasibility factor reduction in all belief sets present in the target system.

3.3.4 Matching and Change of Feasibility

Supervision is expected to be continuous and last beyond any detected failure of the target system. Since feasibility factor may take a continuous range of values and some beliefs will expect a different behavior from the one produced by the target system (should it be a result of a malfunction of the target system or the unrealized behavioral permutation produced by the belief generation algorithm), it would be necessary to define the rules that will be guiding changes of the feasibility factors under various circumstances. The matching algorithm itself will be defined elsewhere and will not be of a concern for us at this moment.

We should devise an algorithm that will balance decreases and increases of feasibility factors and will allow for recovery of feasibility factors to high levels in the post-failure operation should the system recover.

When a mismatch is detected in some belief between its expected behavior and the observed behavior of the target system, feasibility should decrease, and when a match is detected feasibility should increase.

$$\begin{aligned}\chi_{new} &= \chi_{old} + \Delta\chi_{match} \\ \chi_{new} &= \chi_{old} - \Delta\chi_{mismatch}\end{aligned}$$

It will complicate the domain significantly if we discriminate between the influence of different types of matches and mismatches on the value of feasibility factor. Although the idea is sound, we will assume that any mismatch will influence feasibility in the same way as any other mismatch, and so will a match.

For how long should we retain the information that feasibility factor was decreased at some point? This information may be useful when the target system is highly reliable and it is expected that supervision will not detect any failures. On the contrary, if failures occur often, it is rather meaningless to remember that a failure has occurred. It may be better to keep information about a failure for a certain time and discard it afterwards.

We will consider failure an unlikely event, take the first option and try to reflect information about past failures in the feasibility factor.

If a failure has been detected by a fuzzy supervisor at any point during the life of a given belief set, we can no longer say that the behavior that has been expected by this belief set matches observed behavior. Therefore feasibility factor of such a belief set should never reach the upper limit of 1 again.

Let us try to construct an algorithm that will be suitable for feasibility factor calculation for Fuzzy supervision.

The simplest way to increment or to decrement feasibility is to add or subtract a constant:

$$\begin{aligned}\Delta\chi_{match} &= a = \text{const} \\ \Delta\chi_{mismatch} &= b = \text{const}\end{aligned}$$

Let's call this an absolute change algorithm, as it does not take in consideration anything at all. Absolute change algorithm is simple, and it is easy to predict what will be the value of the feasibility factor for the given belief set:

$$\chi_{current} = 1 + na - mb$$

where n is the number of detected matches and m is the number of detected mismatches.

The problem with this algorithm is: resulting feasibility may overshoot the intended range and become either less than zero or greater than one. This points us to the idea, that we have to consider the value of the feasibility factor of the given belief set when deciding on how much to decrease or increase the feasibility:

$$\begin{aligned}\Delta\chi_{match} &= \alpha(\chi_{old}) \\ \Delta\chi_{mismatch} &= \beta(\chi_{old})\end{aligned}$$

An additional constraint is that the resulting value will still stay within the allowed feasibility range:

$$\begin{aligned}0 \leq \chi_{new} = \chi_{old} + \alpha(\chi_{old}) &\leq 1 \\ 0 \leq \chi_{new} = \chi_{old} - \beta(\chi_{old}) &\leq 1\end{aligned}$$

Also, as we agreed earlier, past mismatches should be reflected in the feasibility value: it should not be equal to 1 ever again once a mismatch was detected.

It is obvious that the class of algorithms delimited by these rather loose constraints is very large. Without attempting to exhaust all of these, we will investigate one particular type of formula in which $\Delta\chi_{match} = \alpha(\chi_{old}) = a(1 - \chi_{old})$ and $\Delta\chi_{mismatch} = \beta(\chi_{old}) = \chi_{old}(1 - b)$:

$$\begin{aligned}\chi_{new} &= F_{incr}(\chi_{old}) = \chi_{old} + a(1 - \chi_{old}), 0 < a < 1 \\ \chi_{new} &= F_{decr}(\chi_{old}) = \chi_{old} - \chi_{old}(1 - b), 0 < b < 1\end{aligned}$$

What are the properties of these algorithms? For one, new feasibility will never leave the permitted feasibility range. In the case of a match, $\Delta\chi_{match}$ is always a fraction of the difference between 1 and the χ_{old} , therefore a sum of the delta and χ_{old} will not be greater than 1, or will be strictly less if $\chi_{old} < 1$. More formally:

$$\begin{aligned}0 < \chi_{old} < 1, 0 < a < 1 \\ 0 < (1 - a) < 1 \\ 0 < (1 - \chi_{old}) < 1 \\ \chi_{old}(1 - a) < 1 - a \\ 0 < \chi_{old} + a(1 - \chi_{old}) < 1\end{aligned}$$

Similarly, $\Delta\chi_{mismatch}$ is always positive and is a fraction of χ_{old} , therefore the result of subtracting the delta from χ_{old} will be always positive and will never be greater than 1:

$$\begin{aligned}0 < \chi_{old} < 1, 0 < b < 1 \\ 0 < (1 - b) < 1 \\ 0 < \chi_{old}(1 - b) < \chi_{old} \\ 0 < \chi_{old} - \chi_{old}(1 - b) = \chi_{old}(1 - 1 + b) = b\chi_{old} < 1\end{aligned}$$

Figure 15 illustrates changes of feasibility of a belief set after one detected mismatch and three matches when $a = 0.5, b = 0.5$ and starting feasibility is equal to 1.

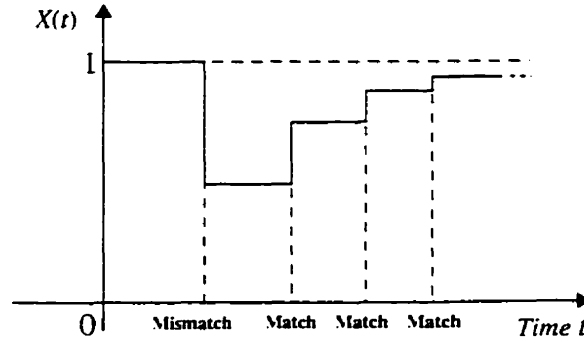


Figure 15: Dynamics of Feasibility Factor of a Belief Set After One Mismatch and Three Matches

As a general observation, increasing constant a will increase the rate of recovery of feasibility as a result of a match, while increasing constant b will slow down the decrease of feasibility as a result of a mismatch.

As we see in these algorithms, a recovery of feasibility from a detected failure would depend on the continuation of (possibly, partly) successful operation of the target system. If the target system continues to operate properly, that is, if signals that are produced by the rest of the system are matched with the corresponding signals expected by the supervisor, the damage to the feasibility of the surviving beliefs would be compensated by successful matching,

In general, with the algorithms selected above the important factor determining the feasibility change in any given period of time would be not even the number of mismatches (failures) that have occurred in given time, but the ratio of successful matches to the number of mismatches that have occurred. For more discussion on this topic see section 3.5.3.

3.4 Matching

Matching in belief-based supervision is a process of comparison of behavior expected by the supervisor and the observed behavior.

As it was set in our objective, matching has to continue after a mismatch has been detected and has to handle signal distortions that were discussed in section 2.1.1 of Chapter II.

3.4.1 Output Signals

What information will be meaningful in output signals? The following attributes characterize expected output signals produced by BSDL model of supervisor:

- name: unique identifier of a signal
- parameters: types and values
- sending path: allows to discriminate signals sent via different channels and identical otherwise
- sending time (time stamp): discriminates identical signals sent at different moments of time
- belief set that produced the signal: consistent belief sets may represent different hypotheses about behavior of the target system and therefore identical expected output signals may be generated by different belief sets and have to be distinguished.

Then, for the reference purposes we'll denote an expected output signal as:

signal:<name, params, path, time, cbs>

where the elements in the brackets denote attributes just listed above.

3.4.2 Behavior Representation and Capture: Output History

As was mentioned before, any consistent beliefs set represent a complete hypotheses about the state of the whole target system.

As there may be many consistent belief sets at any given time there also will be more than one expected behavior existing in a software supervisor during its operation.

Observed behavior as well as expected behavior of the target system can be represented as sequences of signals.

Expected output queues consist of signals that were produced by belief processes of a consistent belief set and were sent to the environment.

In the case of observed behavior, signals come from an abstractor process or processes, that scan the interface memory of the target system and interpret physical characteristics of the interface memory as application of certain output signals.

Discussion on issues with signals abstraction conducted in section 2.1.1 of Chapter II illustrated that the abstracted behavior may suffer from various distortions such as misobserved signals, missing signals, and phantom signals.

Also, a concurrent execution of several abstractor processes may cause permutations of the abstracted signals in the observed behavior queue. The permutation problem also occurs when several processes of the same belief set deliver output signals through the same channel. This illustrates the importance of properly aligning of the model to the target system (e.g., directing signals of different nature into separate channels and properly selecting scanning algorithms) in order to eliminate problems like that. For clarity we will consider that the model we have has these properties.

If we represent behaviors as sequences of signals, and as it was shown, there may be multiple possible expected behaviors, we may have several expected output queues. The observed behavior also may result in multiple queues, if we consider building hypothesis of what was observed in a way similar to hypothesizing on what has happened in the target system. But simple observed behavior will suffice if we eliminate non-determinisms of forming an observation through appropriate mechanisms of abstraction and well-suited supervision model. We will consider that only one version of observed behavior is sufficient and assume that observational non-determinisms have been eliminated.

Output behavior captured as a sequence of signals ordered by their timestamps will be called *output history*.

Here is the notation we'll use to represent output history:

data output_history: <signal_1, signal_2, signal_3, ..., signal_N>

3.4.3 Performance Time-Outs of Signals in BSDL Model

Signals sent via BSDL signal channels among others, would generate beliefs that the signal is still in transition, and these would live indefinitely. Practical considerations tell us that if we send something via some transport route we would expect it to arrive in the point of destination in some time, finite, if not pre-determined explicitly.

This prompted introduction of the time-out mechanism that discarded all consistent belief sets which had signals that have been staying in "in-transit" state longer than a certain time after their departure from the sender processes. This mechanism was termed *performance time-out*.

Below is the "old" performance time-out algorithm for expected output signals and signals in "in-transit" list:

```
1: algorithm old_check_performance_timeout(signal):
2:   if (<(time(signal), -(current_time(), performance_timeout)))
3:     then discard(cbs(signal))
4:   end old_check_performance_timeout
```

where *time(...)*, *cbs(...)* are selectors of signal attributes, and *discard(arg)* is a function that deletes passed data object *arg*.

In the Fuzzy belief-based software supervision timing out will carry out functions that were non-existent in the original belief-based supervision: as we expect supervisor to continue operation in presence of signal distortions and possibly faulty behavior of the target system and beyond the moment of detection of behavioral deviation, we should be prepared to match output histories in which some signals do not have a matching counterpart due to a faulty behavior or observational distortion. Such signals should also be removed by the performance time-out algorithm.

What should be different in the performance time-out algorithm if it is to be used in Fuzzy belief-based software supervision? In one statement, performance time-out of a signals should not be allowed to cause an unconditional termination of associated CBS as that would prevent CBSes from freely competing with each other.

This change directly affects performance time-out algorithms for “in-transit” signals and expected output signals:

```
1: algorithm check_performance_timeout(signal):
2:   if (<(time(signal), -(current_time(), performance_timeout)))
3:     then process_performance_time_out(signal)
4:   return
5: end check_performance_timeout
```

where *time(...)*, *cbs(...)* are selectors of signal attributes, and *process_performance_timeout(signal)* modifies feasibility due to a timed out signal in a certain way, that is discussed in section 3.3.4.

3.4.4 Timing Out of the Observed Output Signals

A mechanism similar to performance time-outs was used to clean up observed output history from older output signals, and also to destroy CBSes in which these timed out observed output signals were never properly matched.

For signals in the observed output history the algorithm used in the binary belief-based supervisor was:

```
1: algorithm old_check_performance_timeout(signal):
2:   if (<(time(signal), -(current_time), performance_timeout))
3:     then discard(cbs(signal))
4:   return
5: end old_check_performance_timeout
```

where *time(...)*, *cbs(...)* are selectors of signal attributes, and *discard(arg)* was a function that deleted passed data object *arg*.

Again, this algorithm has to be modified for the purposes of Fuzzy software supervision as it was explained in section 3.4.3, replacing the unconditional destruction of the belief sets that haven't matched the timed out signals.

Instead of immediately deleting the cbs that has an observed output signal that timed out, the fact will be merely registered in the cbs, and an immediate destruction will not occur:

```
1: algorithm check_performance_timeouts(history):
2:   for (every signal from history) {
3:     check_performance_timeout(signal)
4:   }
5:   return
6: end check_performance_timeouts
```

```
1: algorithm check_performance_timeout(signal):
2:   if (<(time(signal), -(current_time(), performance_timeout)))
3:     then process_performance_time_out(signal)
4:   return
5: end check_performance_timeout
```

where *time(...)*, *cbs(...)* are selectors of signal attributes, and *discard(arg)* is a function that deletes passed data object *arg*, and *process_performance_timeout(signal)* modifies feasibility due to a timed out signal in a certain way, that is discussed in section 3.3.4.

3.4.5 Matching Individual Signals

Behavior captured as output history consists of output signals. In order to define matching of behaviors we have to define when do we call two signals matching each other.

As defined in SDL, signals may carry parameters and when sent travel a certain path determined by SDL specification. As matching is a responsibility of the BSDL abstract machine, sending time, delivery path, and information on the signals parameters is available when matching is performed.

Observed output signals must be produced during signal abstraction that uses specific knowledge about the target system and the supervisor model to interpret changes in the interface memory of the target system as signals.

The best possible meaningfulness of matching of individual signals may be achieved when the observed output signal contains is associated with exactly the same types of information as the expected output signal. Then no information is wasted, and matching of two signals may be decomposed into matching of individual signal attributes such as signal channel, timestamps, parameters, etc.

We will consider the signal abstraction and supervisor model perfectly aligned: that is if the target system is operating properly, abstracted signal would be attributed with the same identifier (name), channel, parameters and timestamp close to that of the corresponding expected output signal produced by the target system.

Matching algorithm for two output signals will look like this:

```
1:  algorithm SignalMatch(signal_1, signal_2):
2:    return(and(  match_sig_time(time(signal_1), time(signal_2)),
3:              match_path(path(signal_1), path(signal_2)),
4:              match_name(name(signal_1), name(signal_2)),
5:              match_parm(parm(signal_1), parm(signal_2))))
6:  end SignalMatch

1:  algorithm match_name(name1, name2):
2:    return(=(name1, name2))
3:  end match_name

1:  algorithm match_path(path1, path2):
2:    return(=(path1, path2))
3:  end match_path

1:  algorithm match_parm(parm1, parm2):
2:    return(=(parm1, parm2))
3:  end match_parm
```

where *name(...)*, *path(...)*, *time(...)*, *params(...)* are simple selectors of signal attributes, and *match_time(...)* is an algorithm comparing timestamps of the two signals and deciding if they can be matched from the point of view of their timestamps. See section 3.4.6 for discussion of timing aspect of signal matching.

This algorithm calls for an explanation. We assumed that signal abstraction is perfectly aligned with the target system. That means, as we defined, that two signals will be identical in everything but maybe their timestamps, and expected output signals will have an

extra attribute - CBS ID that will be meaningless in the observed output signal, as the observed output history will be unique as we agreed. That is exactly what the algorithm does: it compares names, parameters, paths, and compares observation and sending timestamps of the two signals as discussed in section 3.4.6.

Although the CBS attribute is not significant in the matching of individual signals, it is important for formation of output history for different consistent belief sets.

3.4.6 Timing Aspect of Individual Signal Matching

Speaking informally, when we are trying to match two signals with each other, timing of their detection is important. It is impractical to try to match two signals that have been received in distant moments of time. This is similar to time-out expiration of output signals discussed in section 3.4.6. Thus we may set some threshold time distance between timestamps of two matched signals beyond which matching of these signals will not be attempted. This threshold may be different with respect to the performance time-out thresholds. Although, if it is more than the performance time-out threshold, it will be useless, as the signals separated by this threshold distance will never be matched with each other, as the earliest of these two will be discarded by the performance time-out mechanism. This is illustrated in Figure 16: T_d stands for detection time of signal i , T_e stands for performance expiration time of signal i . As we see, it is impossible that matching of signal 1 and signal 3 will ever be attempted as signal 1 will be expired by the time signal 3 is detected.

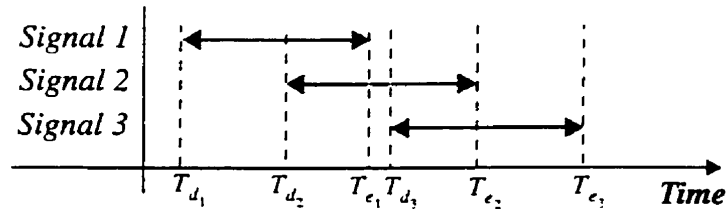


Figure 16: Potential Life Intervals of Signals and Time Matching

On the contrary, signals 1 and 2 may coexist in supervisor at the same time as their potential life spans overlap. Attempt of matching of these two signals may be prevented by sufficiently short critical temporal distance threshold, namely, it should be shorter, than the temporal distance between their respective detection timestamps.

Here is the algorithm that performs matching of timestamps referred to in section 3.4.5.

```

1: algorithm match_sig_time(time1, time2):
2:   return (< ( abs(-(time1, time2)),
3:             timeout))
4:   )
5: end match_sig_time

```

In this work we will consider performance and temporal distance thresholds equal, so that temporal distance checking will become redundant.

3.4.7 Decomposition of Output History Matching Into Channel Matching

Representation of behaviors as sequences of signals makes the task of matching relatively simple. The idea is to simultaneously progress in two output history lists while matching eligible individ-

ual signals contained within each of the sequences. We have to define which signals will be eligible for matching and how will a match or a mismatch affect the output history lists. For now we will only consider that there are only two sequences to match and will address the problem of output matching for multiple belief sets later.

A problem we have to solve in the matching algorithm is the possible absolute delivery order permutation of signals travelling via different channels: even though several signals could have been sent in the right order, but through different channels, they will emerge from the channels they travelled through in a different order, due to the non-deterministic in-channel delay.

Channels may delay, but do not permute signals sent through them. Therefore, the delivery ordering of two signals sent through the same channel will be the same as the departure ordering (see Figure 17).

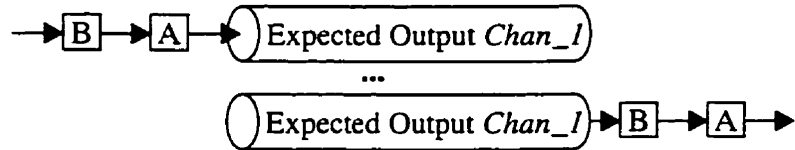


Figure 17: First-In-First-Out Signal Transmission in SDL Channels

Signal abstraction and specification are aligned, as we postulated, that means that abstraction does not permute observations of signals that would have been expected to emerge on the same channel in the SDL model.

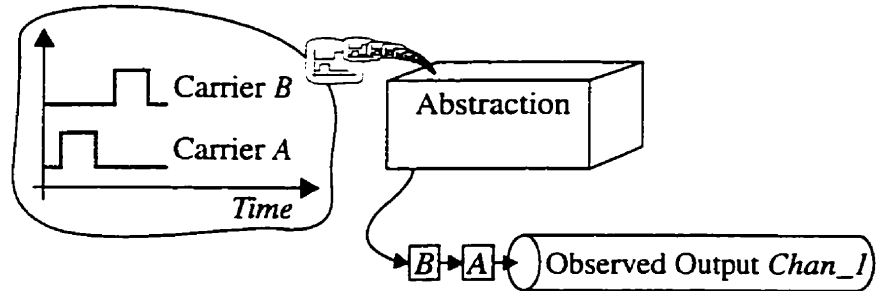


Figure 18: Abstraction of Same-Channel Signals in Supervisor

In Figure 18 two signals can travel along channel Chan1, although they have different signal carriers in the target system. Nevertheless, the ordering of these two is the same as the ordering of their appearance in the carriers. We will assume, that signals for which it is impossible to guarantee non-permuting detection will be sent to separate channels within the SDL specification.

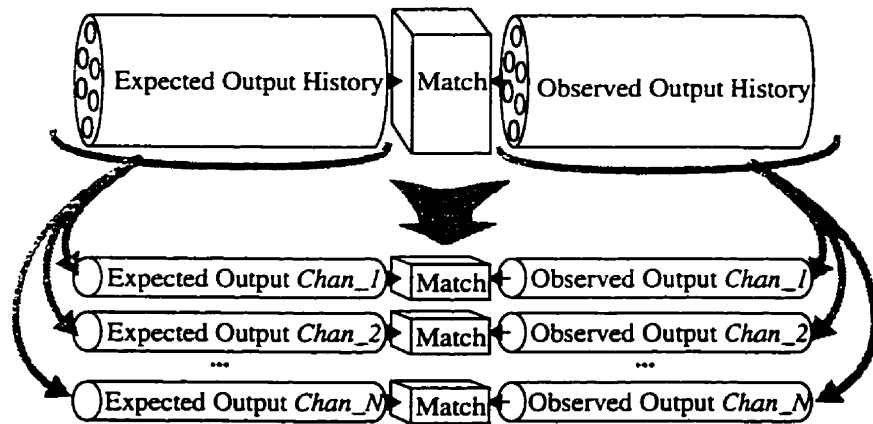


Figure 19: Decomposition of Behavior Matching Into Same-Channel Matching

Now, since inter-channel signal permutation is ruled out, we can split signals in the output histories into per-channel sub-histories and decompose matching of two output histories into matching of their same-channel sub-histories. Later we will explain how to recombine the results of same-channel matching into a result of output history matching.

3.4.8 Matching Same-Channel Output Histories

As before, when trying to match two sequences of signals we should expect that some of the signals present in one sequence will be missing in the other, or the order of signals in one will be different from the order of signals in the other.

Permutation of output signals in one same-channel sequence with respect to the order of signals in the other will indicate a faulty behavior according to our postulate presented in section 3.4.7 and should be caught as a mismatch.

Matching will be an iterative process of matching of individual signals starting from the “earliest” ends of signal sequences and progressing as necessary further in the sequences. Since signals may be missing in both sequences, the candidate signal for matching will be selected from either sequence in turns, thus the two sequences play equal roles in matching algorithm except for the selection of the very first candidate.

Once a candidate for matching is selected in one queue, an attempt will be made to match it with the next untried signal in the other queue. If the attempt fails, the matching turn is passed to the other sequence. Once a candidate has been unsuccessfully matched with all of the signals in the other sequence, a new candidate is selected, that is the signal immediately following the ex-candidate in its sequence.

Note, that only portion of the opposite sequence starting from its candidate and ending at the end of the sequence should be considered for matching with the current candidate of the given sequence, as the signals from the beginning of the opposite sequence and up to its current candidate have been tried already with all of the signals constituting the given sequence.

Thus, if matching is not successful, the sequence of attempted matches is shown in Figure 20, where S_j^i is the j -th signal in sequence i , n_i is a number of signals in sequence i , *SignalMatch* is the individual signals matching algorithm presented in section 3.4.5. The shown matching sequence will occur when $n_2 < n_1$, and $n_2 n_1$ is odd and absolutely no signals match in the two sequences.

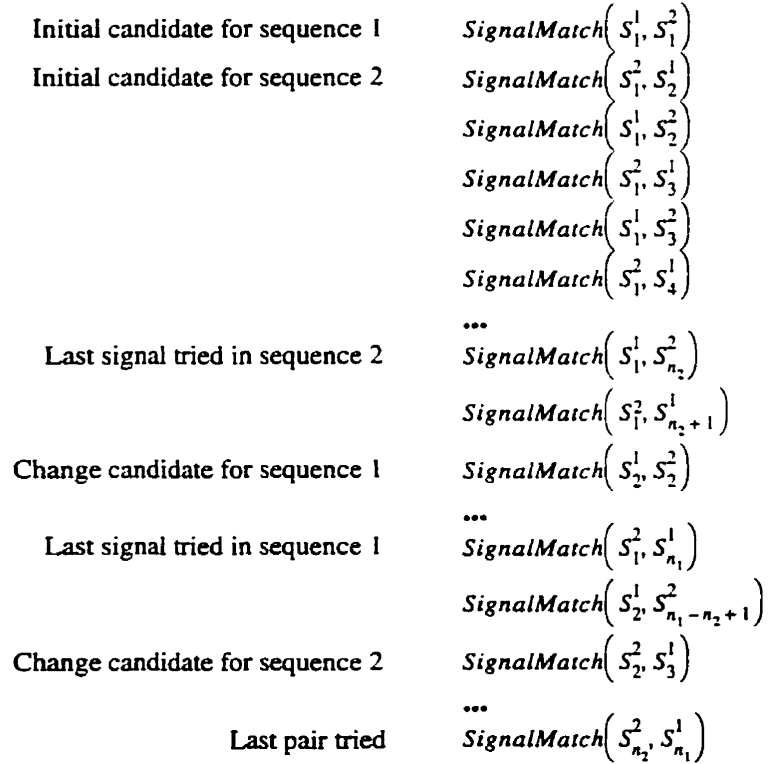


Figure 20: Sequence of Attempted Matches

What should happen once we did find matching signals somewhere in the two sequences? The way our match search algorithm has been designed guarantees that no matching pairs were found for signals from the start of the sequences up to the matched signals; these unmatched prefixes will have to be discarded to prevent matching with them at a later time (this prevents a successful match of two permuted sequences).

We should consider every discarded signal a mismatch and modify feasibility accordingly. When unmatched prefixes (if any exist at all) are removed from the sequences, the matched pair is registered as a match followed by an appropriate increase of feasibility.

After that the matched signals are also discarded to prevent repetitive matches with the same signals. Figure 21 illustrated consequences of a detected match.

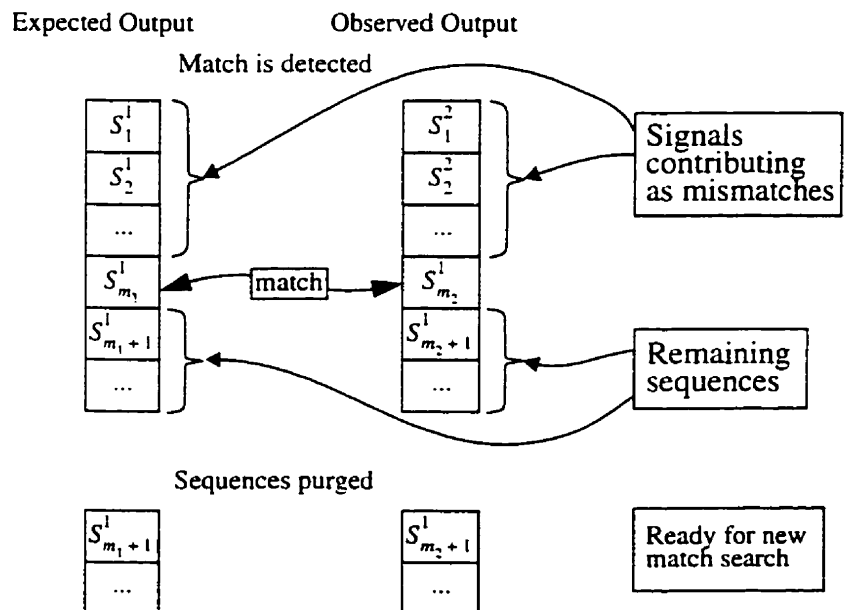


Figure 21: Discarding Unmatched Prefixes and Matched Signals

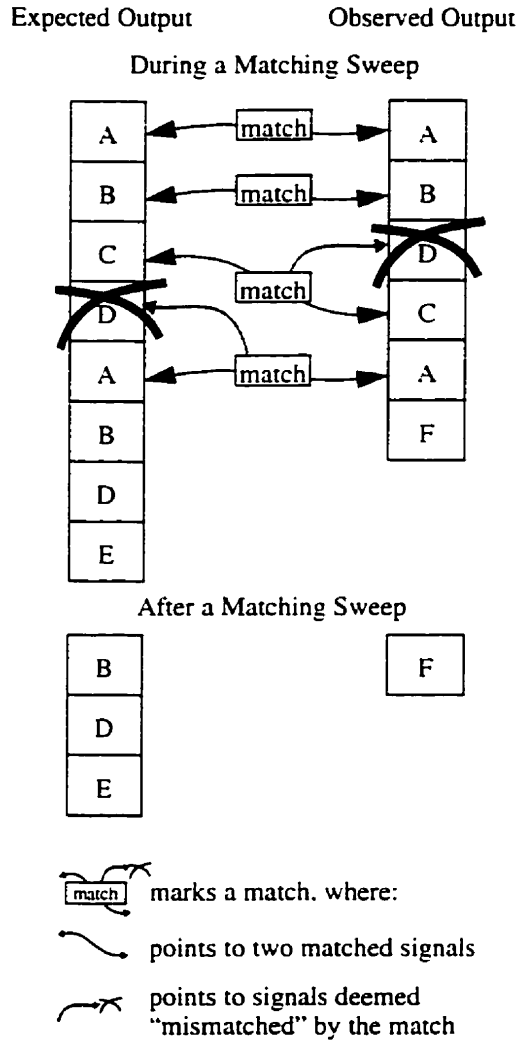


Figure 22: Expected and Observed Output Histories Before and After a Matching Sweep

The idea of same-channel signal matching is clear now. Let us give a more formal presentation of the same-channel matching algorithms discussed in this section (some algorithms have been described in other sections)

Algorithm *match_channel_history* is the top-level algorithm that checks for performance time-outs in both histories and finds all matches in the two remaining history lists.

```
1: algorithm match_channel_history(history_1, history_2):
2:   check_performance_timeouts(history_1);
3:   check_performance_timeouts(history_2);
4:   match_not_found = false;
5:   while (not(match_not_found)) {
6:     match_not_found = find_channel_match(history_1,
                                           history_2);
7:   }
8:   return;
9: end match_channel_history
```

Algorithm *purge_channel_history* purges passed history from the beginning and up to the matched signal, processing all mismatches and the final match and cleaning up the history from processed signals.

```
1: algorithm purge_channel_history(matched_signal, history):
2:   history_head = head(history);
3:   while (not(history_head = matched_signal)) {
4:     process_mismatch(history_head);
5:     history_head = next(history_head);
6:     remove(history_head, history);}
7:   process_match(matched_signal);
8:   remove(matched_signal, history);
9:   return;
10: end purge_channel_history
```

Algorithm *find_channel_match* successfully finds one match and processes matches and caused mismatches, and cleans up the histories from processed signals (description of variables used in the algorithm can be found immediately below the algorithm itself):

```

1:  algorithm find_channel_match(history_1, history_2):
2:      candidate_1 = head(history_1);
3:      candidate_2 = head(history_2);
4:      match_1 = candidate_2;
5:      match_2 = next(candidate_1);
6:      match_found = false;
7:      match_not_found = false;
8:      turn = true;
9:      until (or (match_found, match_not_found)) {
10:         if (turn)
11:            then {
12:                if SignalMatch(candidate_1, match_1)
13:                   then {
14:                       match_2 = candidate_1;
15:                       match_found = true;
16:                   }
17:                else {
18:                    if (or((candidate_2 = NIL), (match_2 = NIL)))
19:                       then {
20:                           match_not_found = true;
21:                       }
22:                    else {
23:                        advance_match_candidates (candidate_1, candidate_2,
24:                                                match_1, history_2);
25:                    }
26:                }
27:            else {
28:                if SignalMatch(candidate_2, match_2)

```

```
29:         then {
30:             match_1 = candidate_2;
31:             match_found = true;
32:         }
33:         else {
34:             if (or ((candidate_1 = NIL), (match_1 = NIL)))
35:                 then {
36:                     match_not_found = true;
37:                 }
38:             else {
39:                 advance_match_candidates (candidate_2, candidate_1,
                                           match_2, history_1);
40:             }
41:         }
42:     }
43:     turn = not(turn);
44:     if (match_found)
45:         then {purge_channel_history(match_1, history_1);
46:             purge_channel_history(match_2, history_2);}
47:     }
48:     return(not(match_found));
49: end find_channel_match
```

The algorithm above is less obvious than the other ones and requires explanation. Arguments *history_1* and *history_2* point to lists of signals constituting output histories. Variables *candidate_1* and *candidate_2* contain references to current candidates for matching in *history_1* and *history_2* correspondingly. Variables *match_1* and *match_2* point to current match prospect for *candidate_1* and *candidate_2* correspondingly and point to signals from *history_2* and *history_1* correspondingly; also these are used to store matched signals once the match is found. Flags *match_found*, *turn*, and *match_not_found* are used to indicate if the match was found, what candidate will be matched in the iteration, and if the match was not

found, just as their names suggest. Here *candidate_i* is the next candidate for the match in *history_i*, *match_i* is the matching prospect for candidate_i. Two pairs of pointers (*history_i*, *match_i*) are needed here in order to perform interleaving top-down matching of *history₁* and *history₂*.

Algorithm *advance_match_candidates* performs advancing of matching candidates through the history lists. If we tried all of the elements in the other queue with *this_candidate*, advance *this_candidate* and set the *this_match* prospect to the signal next to the *other_candidate*, else simply advance the match prospect *this_match* to the next element. If advancing is not possible, simply *return* from execution.

```
1:  algorithm advance_match_candidates (this_candidate, other_candidate,
                                     this_match, other_history):
2:      if (this_match = last(other_history))
3:      then {
4:          if (or (next(this_candidate)=nil), (next(other_candidate)=nil))
5:          then
6:              return;
7:          else {
8:              this_candidate = next(this_candidate);
9:              this_match = next(other_candidate);
10:         }
11:     }
12:     else {
13:         this_match = next(this_match);
14:     }
15:     return;
16: end advance_match_candidates
```

where *next(signal_arg)* finds signal standing next after *signal_arg* in its history.

The notation of the presented algorithms is rather awkward and implementation-like. It had to be chosen due to the nature of the algorithms and should be transparent enough to express fine details of these algorithms.

3.4.9 Matching of Histories With Multiple Channels

What should be changed in the algorithm developed in section 3.4.8 if we were to match histories that have signals that travelled through different channels? Not much, luckily. Before attempting to match two signals we should check if they have same channels. If so, we will proceed as before, if not, nothing should be done, and matching of the next pair of signals should be attempted. The old *find_channel_match* algorithm, actually, will work just fine with multiple channel histories, as attempts to match two signals from different channels will fail according to the algorithm suggested in section 3.4.5. Only purging algorithm has to be changed, as only signals with the same channel as the matched one should be purged from the history, and presence of other-channel signals will not conflict with the successful match.

Overall, matching algorithm with multiple channel histories will be equivalent to simultaneous execution of several same-channel matching algorithms.

We will not rewrite all the matching algorithms here. The only change that will be necessary is to replace the involved same-channel algorithms by their multi-channel analogs.

Here is how the new algorithms look like (the longest *find_next_match* has been omitted, as the old *find_channel_match* is trivially derived from *find_channel_match* with minimal changes that become obvious from the examples below):

```
1: algorithm match_history(history_1, history_2):
2:   check_performance_timeouts(history_1):
3:   check_performance_timeouts(history_2):
4:   match_not_found = false;
5:   while (not (match_not_found)) {
6:     match_not_found = find_next_match(history_1, history_2);
7:   }
8:   return;
9: end match_history

1: algorithm purge_history(match, history):
2:   signal = head(history);
3:   while(not (signal = match)) {
4:     if (channel(match) = channel(signal))
5:       then {
6:         process_mismatch(signal);
7:         remove(signal, history);
8:       }
9:     signal = next(signal);
10:  }
11:  process_match(match);
12:  remove(match, history);
13:  return;
14: end purge_history
```

3.4.10 Matching Behaviors of Multiple Belief Sets

Matching will work in cycles repeating after a match is detected and sequences are purged, until there is no match possible between the signals remaining in both sequences.

Figure 21 illustrates a complete matching sweep with two sequences, that detects matches and mismatches.

Let us address the problem of matching of expected output behavior then there are multiple belief sets. In section 3.4.2 we stated that observed behavior will be a unique entity. This poses a potential problem, since outcome of matching of two behaviors will be unique for every belief set. Replication of the observed behavior is a sufficient and simple solution for this problem. Then every belief set will be able to keep and maintain its own version of remaining output history and matching may be unique. When new output signals are observed we simply will append these to the end of the local copy of observed output history list in every surviving belief set.

```
1:  algorithm append_observed_output(new_observed_outputs,
                                   all_belief_sets):
2:    for (every belief_set from all_belief_sets) {
3:      append(new_observed_outputs,
              local_observed_history(belief_set))
4:    }
5:    new_observed_outputs = NIL;
6:    return;
7:  end append_observed_output
```

where *append(...)* takes two lists as arguments and appends first argument to the end of the second argument, and *local_observed_history(...)* is a selector of local copy of observed output history associated with the given belief set. We will refer to this algorithm in the combined supervisor matching algorithm.

Figure 23 illustrates matching for several consistent belief sets. One CBS is expanded and localized matching is illustrated with histories sorted by the channel of the signal. Observed output history has been duplicated in every belief set.

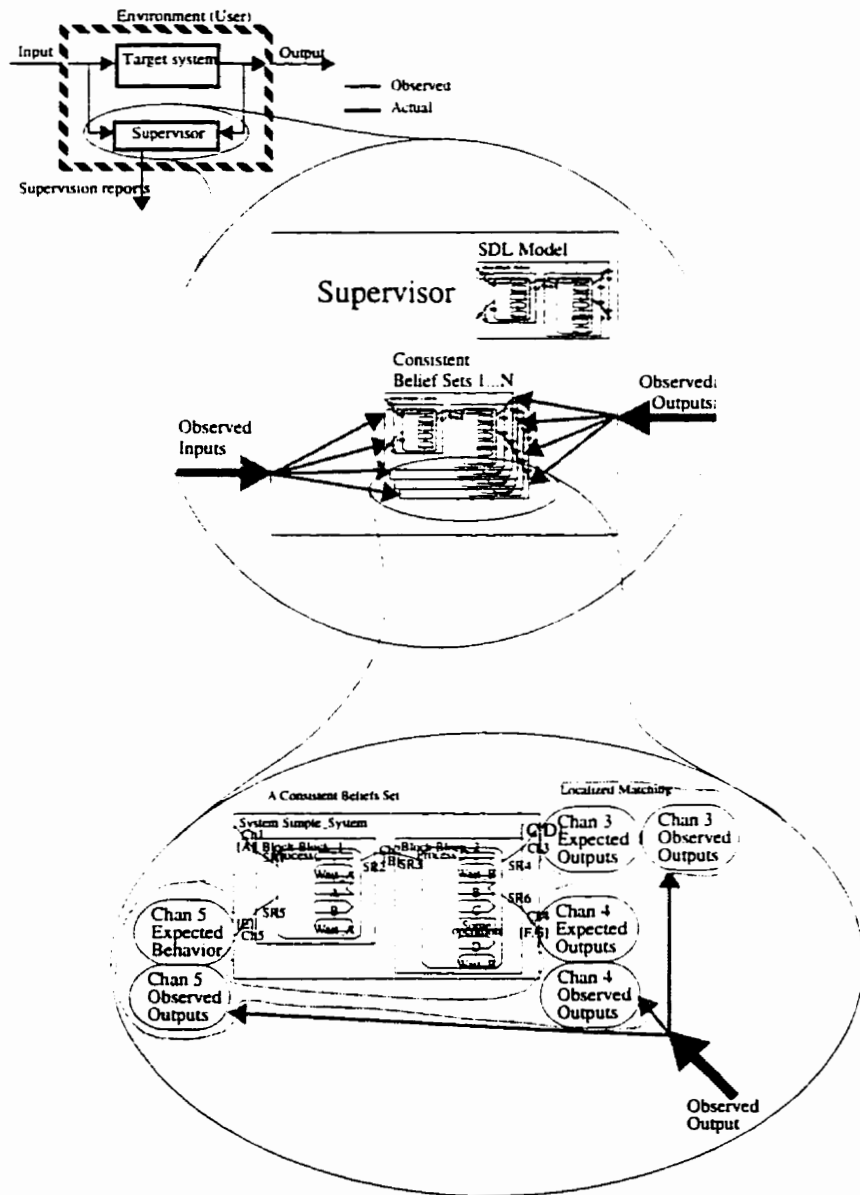


Figure 23: Matching Within a Belief-Based Supervisor

3.4.11 Combined Matching Algorithm for Fuzzy Supervisor

We may combine all the pieces together and construct an algorithm that will suit our needs for a fuzzy belief-based supervisor, that may have multiple beliefs with histories consisting of signals from multiple channels:

```
1:  algorithm perform_matching (new_observed_outputs, all_belief_sets):
2:      append_observed_output(new_observed_outputs, all_belief_sets);
3:      for (every belief_set from all_belief_sets) {
4:          match_history(local_expected_output(belief_set),
                        local_observed_output(belief_set));
5:      }
6:  end perform_matching
```

This concludes the development of matching algorithm.

3.5 Competition Algorithm

Mismatches, in-transit and performance time-outs have lost their purpose for immediate belief set termination in fuzzy supervision. This leaves us with a problem: how to discard belief sets that have failed to explain the observed behavior of the target system.

As defined, feasibility factor attributed to a given belief set expresses how close were the output expectations of this belief to the observed output of the target system, and the closer these were in the past, the higher the feasibility should be.

Generation of new beliefs is a way of operation of a belief-based supervisor. If we had unlimited computing resources we could allow a fuzzy supervisor to produce beliefs and not worry about their numbers, as we still would be able to select these that still explain the observed behavior the best. This is not the case, though, and we

have to set rules of restriction of population of belief sets to make supervision practical.

In order to limit the number of belief sets existing in a supervisor, we can either restrict generation of new belief sets or will have to remove some of the existing ones. The first option seems doubtful, as belief generation is a key mechanism to provide exhaustive coverage of alternative behaviors that are permitted by the non-deterministic specification. Thus we will develop the second option.

3.5.1 Cutoff Threshold Algorithm

Supervisor's goal is to explain the behavior of the target system. If we were to choose candidates for termination among the belief sets, it will be logical to select the ones that do not explain the state of the target system very well. This in turn will be expressed in the lower values of their feasibility sets.

This provides us with a simple but efficient way of separating "bad" beliefs from the "good" ones: we will introduce a threshold algorithm that will weed out all belief sets that have feasibility lower than a certain threshold value.

```
1: algorithm perform_cutoff (all_belief_sets, cutoff_threshold):
2:   for (every belief_set from all_consistent_belief_sets) {
3:     if (<(feasibility(belief_set), cutoff_threshold))
4:       then {
5:         remove(belief_set, all_belief_sets)
6:         discard(belief_set);
7:       }
8:   }
9: end perform_cutoff
```

Figure 16 illustrates the competition algorithm. Belief sets are sorted by their feasibility factor value and only sets with feasibility factor higher than X_{cut} will survive.

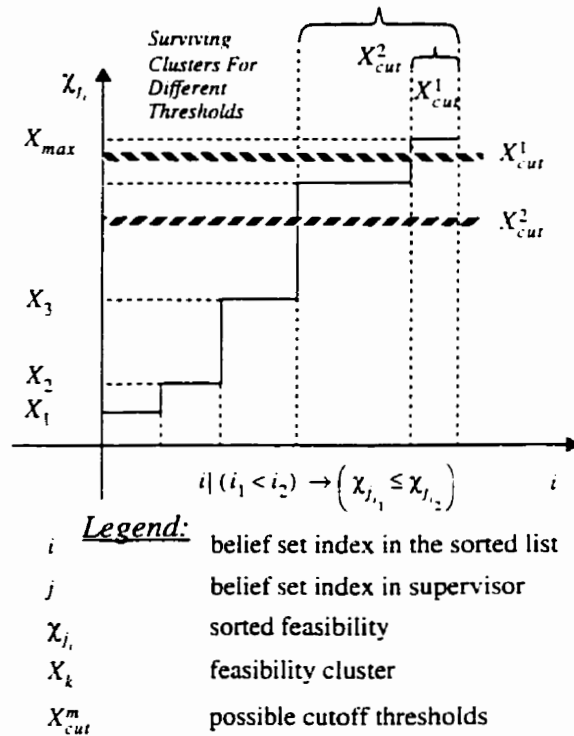


Figure 24: Cutoff Thresholds

3.5.2 On Necessity of a Variable Cutoff Threshold

It is easy to notice that value of cutoff threshold influences how many belief sets are going to survive after a cutoff procedure.

Also, it is obvious that we cannot make thresholds fixed: the feasibility change algorithm that we selected may bring feasibilities of

all existing belief sets below any preset level should an occurrence of multiple failures in a short period of time happen. Should the threshold exceed the maximum feasibility among the surviving belief sets, all sets will be discarded after threshold cutoff procedure, and supervision will stop.

If we have to vary cutoff threshold, what should we consider when setting it?

3.5.3 Calculation of Cutoff Threshold Value

In this section we will discuss principles and constraints that we should consider when selecting values for cutoff threshold.

First of all, in order to avoid destruction of all belief sets, we should set the threshold lower than the maximum feasibility factor existing in the supervisor. This constraint does not ensure anything than there will be at least one surviving belief set after the cutoff threshold algorithm is applied.

$$0 < X_{cut} < X_{max}$$

When a failure does occur, the observed behavior of a target system may be better explained by some of the “incorrect” hypotheses that are present in the supervisor, at least at first. Thus, the feasibility of belief sets expressing these incorrect hypotheses in the first moments may be higher than that of the “correct” beliefs. (“Correct” and “incorrect” here, as everywhere else in this work relate to intended behavior as opposed to performed behavior. This a belief may be incorrect, when it does not describe behavior that the target system intended to provide, but did not do so due to a malfunction. These qualities are assigned hypothetically, as the target system is a black box for the supervisor and it is impossible to say what did it intend to do when observing it from the outside. “Correctness” of a

hypothesis may be confirmed later when the target system recovers from the failure and continues to operate as expected by the “correct” belief.)

This observation hints to us that the cutoff threshold should be set in such a way, that an average failure will not cause an immediate termination of “correct” beliefs, but will allow them to grow their damaged feasibility factors to a higher level with new successful matches and outlive “incorrect” beliefs with the continuing operation of the target system.

This introduces a dependency between the variable cutoff threshold mechanism, feasibility change algorithm, matching algorithm and even the specific characteristics of the target system such as failure frequency, ratio of number of mismatches to the number of matches as a result of a typical failure, and the model used by the supervisor. Dependency is not a trivial one, and has probabilistic features, as it is impossible to predict exact failure profiles of the target system with absolute certainty. That means that there will be a possibility that in some relatively rare situation the calculated cutoff threshold value will happen to be too high, and the “correct” belief set will be destroyed after all. (This prompts us to provide a mechanism that would allow supervisor to eventually recover from destruction of a “correct” belief set should the latter occur due to some unfortunate circumstances. We will discuss such a mechanism in later sections.)

We shall not attempt to discover a complete cutoff threshold dependency in this thesis, as it would be a very complex scientific endeavour on its own. Instead we will impose some constraints on the involved variable parameters and try to come up with a simplified algorithm that will produce an acceptable value for the variable cutoff threshold.

Let us say that predominant failure type produces N mismatches as a result of one match. If the original feasibility in the belief was

χ_{old} then according to our feasibility change algorithm the resulting feasibility immediately after discovery of a mismatch will be a superposition of F_{incr} and N -recursive superposition of F_{decr} that are the functions we defined in section 3.3.4 for changes of feasibility factors

$$\chi_{new} = F_{incr}(\underbrace{F_{decr}(F_{decr}(\dots(F_{decr}(\chi_{old}))))}_{N})$$

Simplification after substitutions show that this comes down to the following formula:

$$\begin{aligned}\chi_{new} &= (F_{incr}(\dots(F_{decr}(\chi_{old})))) = (F_{incr}(\dots(\chi_{old}b))) \\ \chi_{new} &= F_{incr}(\chi_{old}b^N) = \chi_{old}b^N + a(1-\chi_{old}b^N) = \chi_{old}b^N(1-a) + a\end{aligned}$$

This provides us with the following constraint:

$$0 < X_{cut} < \chi_{old}b^N(1-a) + a$$

This constraint will ensure that a belief with feasibility χ_{old} will survive N mismatches followed by a match.

In general case, when we fix a scenario in which some belief with feasibility χ_{old} has to survive, the cutoff threshold should be lower than the feasibility resulting from this scenario:

$$0 < X_{cut} < F^{scenario}(\chi_{old})$$

where $F^{scenario}(\chi)$ is a superposition of the predicted feasibility changes as a result of matching in the fixed scenario.

And if there are several such scenarios, we have to take an absolute minimum of their predicted outcomes as the upper boundary for the cutoff threshold.

3.6 Fuzzy Supervisor Reports

An outcome of classic belief-based supervision was a failure report produced when all beliefs died as a result of a failure of the target system.

Fuzzy supervisor can retain much more information about the target system than the binary supervisor could, as fuzzy supervisor maintains a population of beliefs which carry information about the past behavior of target system.

In this section we will present various types of information that can be obtained during operation of a fuzzy supervisor, and possible interpretations of this information for the reporting purposes.

3.6.1 Health of the Target System

Feasibility of any belief reflects the “health” of the target system from the point of view of the given belief. The belief with the highest feasibility will represent the best explanation of the target system behavior that supervisor managed to produce.

So, a maximum of the feasibilities of the whole population of belief sets will reflect the estimation of the overall health of the target system from the point of view of the supervisor at the given instant of time.

```
1:  algorithm report_system_health(all_belief_sets):
2:      current_health = 0;
3:      for (every belief_set in all_belief_sets) {
4:          if (< (current_health, feasibility(belief_set))
5:              then current_health = feasibility(belief_set);
6:      }
7:      report(current_health);
8:      return;
9:  end report_system_health
```

where $report(arg)$ is a function that uses passed argument arg to produce a report of a fuzzy supervisor.

The algorithm above produces a value within the allowed range of feasibility factor, that may be used as output of the supervisor. As the feasibility factor itself, higher result will mean better operation of the target system from the point of view of a fuzzy supervisor.

3.6.2 Immediate Failure Report

As we discussed in section 3.3.3 a deviation of behavior of the target system from the behavior expected by a given belief set immediately manifests itself in reduction of associated feasibility factor.

When a failure occurs in a target system, no belief set can explain behavior of the target system as every existing belief set expects behavior different from the behavior produced by the target system, that will be equivalent to a simultaneous reduction of feasibilities of all existing belief sets.

We may use this simple observation to produce another type of a failure report every time when the maximum feasibility drops:

```
1: algorithm report_immediate_failure(all_belief_sets, curr_system_health):
2:   if (< (system_health(all_belief_sets), curr_system_health))
3:   then {
4:     report('failure');
5:   }
6:   curr_system_health = system_health(all_belief_sets);
7:   return;
8: end report_immediate_failure
```

The algorithm produces a failure report when the new value of system health is less than the old value $curr_system_health$, and updates $curr_system_health$ with the new health of the system.

3.6.3 History Tree and Underlying Fault Localization

The maximum feasibility provides an answer for “what is the health of the system”. It may be meaningful to also answer the question like “why the system is not healthy” or “why is the health of the system what it is”.

Feasibility of a belief set is originally inherited from the parent of the belief set and then is modified depending on the outcome of matching of expected output of the belief set and the observed output. If we trace all the outcomes of matching events in a genealogical tree of a given belief down to the original belief set of the fuzzy supervisor, we will obtain a complete log of matching events that resulted in the current feasibility of the given belief set.

If we consider health of the system as defined in section 3.6.1, for the best belief set (the set with the highest feasibility) the log we just constructed will serve as an explanation of why the health has the value it has. The log would be able to explain what differences and similarities were detected between the expected output of the best belief set and the observed output of the target system.

In any case, such a log provides information on what functionality of the target system has deviated from the specification. And if a mapping of functional specification (such as an SDL model) to the implementation of the target system exists, matching history can be used for fault localization.

As an example, consider an SDL specification of a small PBX, with multiple phone handlers. Multiple signal mismatches on channels associated with the same phone handler would indicate a problems with service on a particular phone line due to software or hardware problems. Type of mismatch (e.g. fast busy instead of slow busy, or missing power ringing) will point to the underlying fault (missing ringing, for example, will indicate a burnout in a ring

trates mapping between the two. It also shows the information that can be placed in every element of the matching history.

History tree will change along with the changes of population of belief sets. Death of a belief set will wipe out the leaf branch corresponding to the dead belief set. When a belief set is killed due to combination of two belief sets into one, the belief with the highest feasibility will survive, so will the branch of the history tree associated with it. Information is partially lost, but we kept the belief with "better history" so the better of the two histories will be kept.

A general rule for maintaining integrity of the matching history tree is that elements constituting history of a living belief should stay, while elements that are not used by any living belief should be removed.

Figure 26 illustrates algorithms for reduction and expansion of a history tree while maintaining its integrity. Every element of the tree is assigned so called branching factor, that indicates how many immediate descendant elements the element has. If the branching factor of an element is greater than one, it will not be destroyed when one of its leaves dies. Branching factor is updated to reflect the most current topology of the tree when new leaves are added or removed.

History tree may present a heavy requirement for memory or storage consumption for a supervisor, as the tree would grow deeper as new outputs are detected but the depth of the history tree may be limited in order to reduce the amount of memory required to store it.

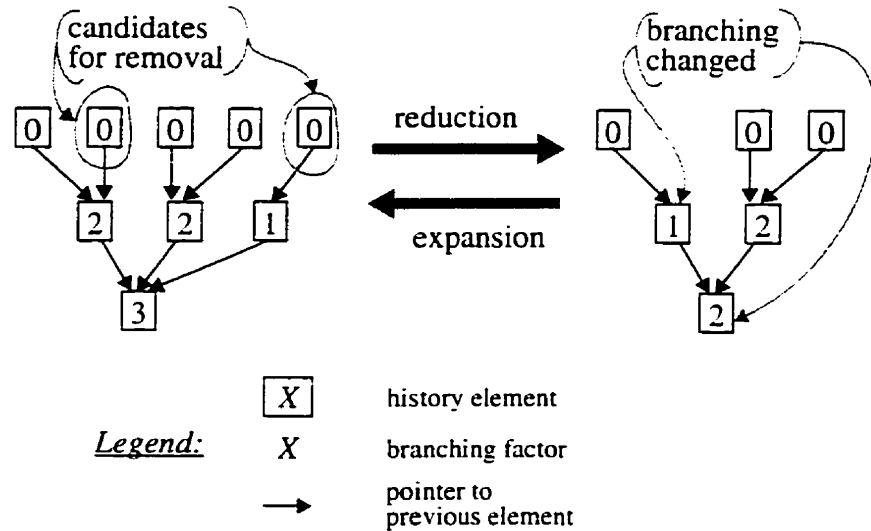


Figure 26: History Tree Manipulation

Algorithm *add_new_element* is used to add a new leaf to the tree:

```

1: algorithm add_new_element(parent_element, child_element)
2:   increment_branching(parent_element);
3:   set_ancestor(child_element, parent_element);
4:   set_branching(child_element, 0);
5:   return;
6: end add_new_element

```

Algorithm *remove_dead_element* allows to start from the leaf of a dying branch and in one recursive sweep with a linear complexity remove all elements in the branch down to the element that is still being used by some living leaf, where the algorithm stops. Note: we will never have a situation where all beliefs in the system would die, therefore at least one surviving branch will be present in the history tree always. Thus we may ignore checking for the existence of the existence of an ancestor of any element, as we will never try to delete the root element of the tree.

```
1: algorithm remove_dead_element(dead_element):
2:   if (<(branching_factor(dead_element), 2)
3:     then {
4:       remove_dead_element(ancestor(dead_element));
5:       discard (dead_element);
6:     }
7:     else {
8:       decrement_branching(dead_element);
9:     }
10:   return;
11: end remove_dead_element
```

Algorithm *decrement_branching(element)* reduces branching factor of *element* by one and algorithm *increment_branching(element)* increases branching factor of *element* by one.

Algorithms *set_ancestor(element, ancestor)* and *set_branching(element, branching_factor)*, as names suggest, set the ancestor and the branching factor of *element* correspondingly.

Algorithms *branching_factor(element)* and *ancestor(element)* return branching factor and ancestor of the *element*. As before, algorithm *discard(arg)* destroys *arg*.

And finally, algorithm *report_matching_history(element)* reports complete matching history of *element*:

```
1: algorithm report_matching_history(element):
2:   report(match_data(element));
3:   if (not(= (ancestor(element), NIL)))
4:     then {
5:       report_matching_history(ancestor(element));
6:     }
7:   return;
8: end report_matching_history
```

Localization of the failure using mapping of functional specification to the target system implementation will not be covered in this work.

3.6.4 Average Feasibility

Instant reading of health of the system may not always reveal a true picture of health of the target system, especially when the range and rate of change of the maximum feasibility in the particular application of a fuzzy belief-based supervisor is significant.

In such cases it may be more informative to calculate an average of the maximum feasibility readings over a recent period of time of pre-determined length and have it reported by the supervisor. The meaning of this report will be similar to the instant estimation of system health, only it will be more inert to changes of the latter. Algorithm for this type of report is trivial and will be omitted here.

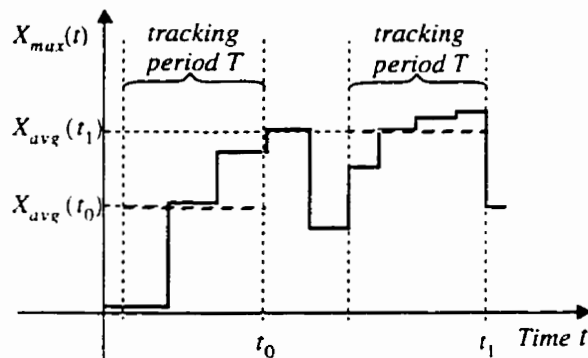


Figure 27: Average Health During a Period of Time T at times t_0 and t_1

3.6.5 Feasibility Distribution

If we had an ideal (or better - Utopian) all-seeing supervisor supervising a perfectly correct target system (no failures, that is), it would have exactly one belief set that will be always able to explain the state of the target system. This is not possible, though since our target system is not perfect and supervisor cannot see what is happening inside the target system. Instead of one we will have multiple belief sets, the number of which, nevertheless, we want to keep to a necessary minimum.

What does the number of belief sets indicate? It indicated the degree of confusion of the belief-based supervisor over the real state of the target system. Fuzzy belief-based supervision does not simplify this problem as with its feasibility factors and threshold cutoff mechanism it allows survival of more “confused” beliefs than the binary belief-based supervisor.

Belief sets with highest feasibility serve as a focus point in the fuzzy supervisor, representing the most successful explanations of the target system. Beside them there will be some beliefs that are slightly more confused but survived after the previous sweep of a liberal fuzzy cutoff threshold algorithm. If the behavior of the target system deviates from the specification, the “correct few” sink to the level of confused majority and the focal point of beliefs with nearly maximal feasibility turns into a larger plateau. This leads to survival of a larger number of beliefs in the cutoff threshold algorithm. So, as degree of confusion of supervisor increases the precision of explanation of the target system’s behavior decreases.

We will use Figure 16 to illustrate the phenomenon. It represents a histogram of feasibility factor of a belief set as a function of index of the belief set in the list of belief sets sorted by their feasibility. N_{lead}^{norm} denotes the number of beliefs in the surviving group in a case when the target system operates properly. N_{lead}^{fault} is the number of beliefs in a case when the target system just suffered a failure.

The number of sets in the highest-feasibility (leading) group can be reported by the supervisor: when statistical data is collected for the number of beliefs in the leading group for a supervision of a target system without failures, it may be used to judge the degree of precision of supervision of the same target system when it starts to suffer failures. The more the number exceeds the “failure-free statistics” the lesser is the precision.

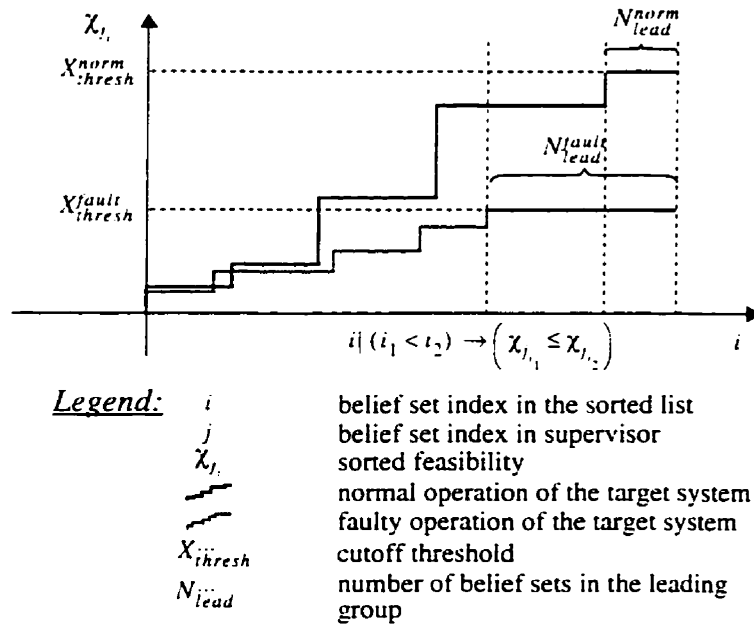


Figure 28: Reporting Feasibility Distribution

This type of reporting would be an interesting topic of research on its own. It is presented here only as a suggestion. Feasibility distributions depend on various parameters of a fuzzy supervisor and dependencies are not obvious at this moment.

3.7 Fuzzy Supervisor Operation

Operation of a fuzzy belief-based supervisor can be described a loop that starts at the same time when the target system starts its operation. Within the loop, signal abstraction derives signals from the environment and sends new input and output signals to input and output signal queues of their destination processes within the supervisor model. Then all ready processes in belief sets are executed iteratively, signals in input signal queues are consumed and trigger transitions within the processes of belief sets, states of processes within belief sets are changed, signals that are produced in the course of execution are delivered to destinations, new belief sets are generated, and execution repeats until no ready processes are left. Until now, operation is identical to that of a binary belief-based supervisor described in Chapter II. Then fuzzy matching is performed, followed by a fuzzy threshold cutoff and garbage collection (combination of belief sets).

Below is a simplified top-level algorithm for fuzzy software supervision that incorporates other algorithm we have developed in this chapter:

```
1:  algorithm fuzzy_supervision():
2:      while(true){
3:          scan_observed_inputs(observed_inputs);
4:          send_observed_inputs(observed_inputs, all_belief_sets);
5:          scan_observed_outputs(observed_outputs);
6:          send_observed_outputs(observed_outputs, all_belief_sets);
7:          execute_processes(all_belief_sets);
8:          perform_matching (observed_outputs, all_belief_sets);
9:          perform_cutoff (all_belief_sets, calculate_threshold());
10:         produce_report();
11:     }
12:     return;
13: end fuzzy_supervision
```

All the algorithms used in *fuzzy_supervision* either have been developed in this chapter or are identical to the algorithms used in binary belief-based software supervisor.

Algorithm *scan_observed_inputs* (*observed_inputs*) obtains all observed inputs that have been abstracted since the last iteration, and stores the list in *observed_inputs*. Algorithm *send_observed_inputs* (*observed_inputs*, *all_belief_sets*) sends signals in *observed_inputs* to the common ports of belief sets contained in *all_belief_sets*. This will trigger belief generation, as discussed in Chapter II.

Similarly, *scan_observed_outputs* (*observed_outputs*) obtains all observed outputs that have been abstracted since the last iteration, and stores the list in *observed_outputs*. And *send_observed_outputs* (*observed_outputs*, *all_belief_sets*) sends signals contained in *observed_outputs* to the observed output queues of belief sets contained in *all_belief_sets*.

Algorithm *execute_processes* (*all_belief_sets*) processes all signals in the input queues of all processes in *all_belief_sets* until none left.

Algorithms *perform_matching* (*observed_outputs*, *all_belief_sets*) and *perform_cutoff* (*all_belief_sets*, *threshold*) were developed and discussed earlier in this chapter in section 3.4 and section 3.5.

In *perform_matching* expected output signals are matched with observed output signals in every belief set, and feasibility factor is modified according to the result of the match.

Algorithm *perform_cutoff*(*all_belief_sets*, *threshold_value*) destroys all belief sets with feasibility less than threshold value calculated in the algorithm *calculate_threshold*() as discussed in section 3.5 “Competition Algorithm” on page 67.

Algorithm *produce_report*() collects statistics and decides whether to produce a report for this iteration, that may be one or several of

the types discussed in section 3.6 “Fuzzy Supervisor Reports” on page 73.

This concludes this chapter. We will evaluate the theory of fuzzy belief-based software supervision in an experimental implementation of a fuzzy software supervisor described in the next chapter.

IV

Experimental Evaluation of Fuzzy Belief Based Supervision

This chapter presents an experimental evaluation of fuzzy belief-based supervision approach.

The following objectives have been set for the experimental evaluation of the fuzzy supervision approach:

- Evaluate the failure detection capability: the FBBS supervisor should detect failures and provide information about the symptoms of the failure.
- Evaluate capability for continuous supervision and resynchronization with the target system: supervision should continue past the moment of failure detection, no false failures should be reported after the continued proper operation of the target system, and new failures should be properly detected.
- Evaluate fuzzy supervisor reporting capabilities: system health, fault localization, failure reports, number of CBSes as a measure of missynchronization of the supervisor and the target system.
- Observe the increase in computational complexity of supervision as a function of different types of failures.

A small PBX with 60 lines capacity has been selected as a target system for fuzzy software supervision.

Collected data is to be analyzed and interpreted, and conclusions are to be drawn from the analysis on the usefulness of the approach, possible drawbacks and benefits.

4.1 Evaluation Environment

Evaluation environment developed in the Bell Canada Software Reliability Laboratory provided a typical target system for software supervision - the control program of a small Public Branch Exchange (PBX), and associated utilities.

The environment was implemented as a collection of communicating processes executing in Sun-OS operating system and resided on a Sun SPARC station. Figure 29 shows processes of the evaluation environment. Duties of these processes are described below.

4.1.1 PBX Hardware Emulator

An emulation of a hardware of a PBX capable of supporting basic telephone service to up to 60 lines was developed as a support software for a course in E&CE Department of University of Waterloo [10], [11]. A shared memory served as an interface between the control program and PBX hardware, accepting stimuli and reflecting changes of state of the simulated hardware. PBX hardware supported three shelves with 30 slots each, with two shelves equipped with telephone line interfaces and one shelf equipped with tone generators, touch tone receivers, maintenance equipment and trunk access slots. Recently the hardware emulation was made capable of supporting up to 1000 telephone lines, but the older smaller version was used in experimental evaluation.

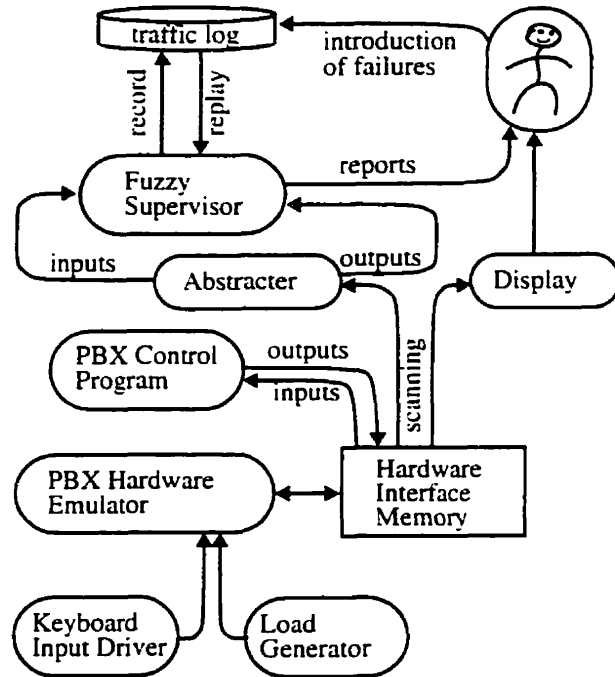


Figure 29: Interactions of Processes In Experimental Environment

4.1.2 PBX Control Software

The PBX control program has been developed to provide control for the small telephone exchange. It serves as a target system for software supervision in the experimental environment.

The program has been developed according to an SDL specification (see Appendix A).

The control software detects input telephony events such as line status changes and recognized digits from touch tone receivers from the shared interface memory of the hardware. It controls PBX connec-

tions and allocation of hardware resources by direct modification of appropriate locations within the hardware interface memory.

The control software used in the evaluation has been developed in C++, and stands at an equivalent of over 4000 lines of un-commented code. Particular details of the implementation of the control software can be found in [10], [11].

4.1.3 Telephony Load Generator

The load generator is a utility process used to produce telephone traffic with various distributions of events. It allows to vary number of calls per telephone per hour, scenarios of telephone calls and distribution of telephone events in time.

The load generator communicates telephone events to the PBX hardware emulator.

4.1.4 Keyboard Input Driver

An alternative way of presenting telephone events to the PBX was provided by the command line interface of the keyboard input driver process. This allowed to enter telephony inputs manually. Keyboard input driver supported simultaneous operations on several lines (such as simultaneous off-hook of a several lines or the whole shelf).

As in the load generator, commands passed to the keyboard process resulted indirectly in modifications of the hardware interface memory.

4.1.5 Abstracter Process

The abstracter process interprets changes in the shared memory interface as telephony events, both inputs and outputs, and delivers events to the supervisor.

The abstracter process does not handle the non-determinisms of signal abstraction we have discussed in first chapters. It simply detects input and output events based on duration of changes in signal carriers in accordance with the specification of the signalling layer.

4.1.6 Display Driver

The display driver process is used solely by a human observer and represents human-friendly variant of abstracter. It monitors activities of the target system by scanning hardware interface memory and extracts information on line status, ringing, voice path connections and touch tone receivers. Display is not communicating with any of other processes.

4.1.7 Fuzzy Supervisor

The fuzzy belief-based supervisor was implemented as a single-threaded process that received abstracted input and output events from the abstracter process and produced reports on the state of the target system. The base portion of the implementation have been developed in a collective effort ([2],[3]) with other graduate students of Bell Canada Software Reliability Laboratory. The author has later modified and extended the base with algorithms of the fuzzy supervision. The fuzzy supervisor consists of an equivalent of over 15000 lines of un-commented lines of source code written in C++, and is constructed of around 70 object classes.

The top-level object model of the implementation is shown in Figure 30. The roles of the classes shown are as follows.

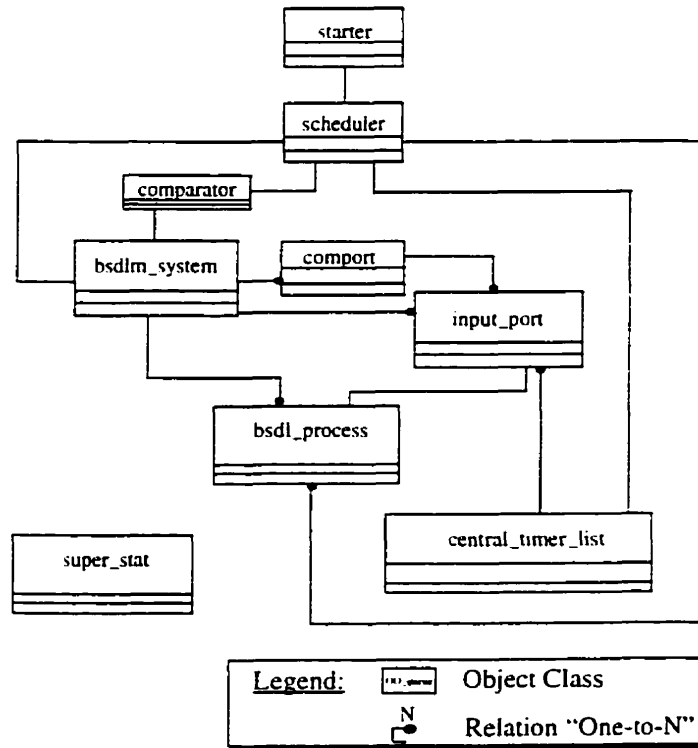


Figure 30: High-Level OMT Diagram of the Fuzzy Supervisor

The starter class has a single instance in the supervisor. The object is responsible for other objects creation and initialization, connection to the hardware interface emulator via the shared memory, collaboration with the Scheduler object that includes time counter management and handling of semaphore operations used for synchronization of the Supervisor operation with the target software system within the experimental environment.

The Scheduler class is also represented by a single instance. The responsibilities of the object include: reception of input and output events from the Abstractor, tracking the Central Timers List for timers

expiration, managing the belief execution (registering, de-registering of beliefs for the execution, and performing the execution), invoking the matching method on the Comparator object to compare expected outputs of belief sets with the observed outputs, and invoking the garbage collection method on the BSDLM system to reduce the number of beliefs by removing redundant beliefs and belief sets.

The BSDLM System instance does the following: it maintains signal delivery routes in a routing table, keeps beliefs database in a set of consistent belief sets, performs dispatching of signals between processes and implements garbage collection - clean up of redundant belief sets.

The Comparator object is also a unique instance of the class. The main purpose of the Comparator is to manage the observed and expected behavior logging and match the logged observed behavior with the expected behavior of the belief sets. See section 4.1.7 for some details on behavior data management. The matching algorithms described in section 3.4 have been implemented in the `check_matches()` method of Comparator class.

The Comport's (that stands for common port) main responsibility is to create all possible permutations of the signals passing to a block when the signal path contains a delaying channel. Multiple instances of the Comport class can exist in the system.

The Input Port manages queueing of signals and timers for a BSDL process. Signals are being queued as they arrive and are consumed in the order they have arrived. Timers can be set and reset; when they expire they are treated as a signal. The Central Timer List is used to detect the expiration of signals

The BSDL Process keeps the state machine of a belief. It uses the BSDL specification and contains values of all variables and an active state for the belief. The BSDL Process is responsible for determining the next state of the belief state machine upon arrival of a signal. The

BSDL Process also produces signals during a state transition when a send signal construct is encountered.

The Central Timer List is a database of active timers. When a BSDL timer is being set, it is registered in the central timer list. Timers are kept sorted by the expiration time. With every increment of time the central timer list is checked for the expired timers. If any timers have expired, a notification will be sent to the input ports of timers origination.

The principal differences with the basic binary belief-based supervisor described in [2] were in the new competitive belief set maintenance and destruction mechanism, and added classes and methods for feasibility management, history tracking, statistics collection, and fuzzy matching algorithm. The operation cycle of the supervisor was modified to invoke competition algorithm at the end of each cycle.

Figure 31 presents a feasibility-related data structure deployed within the fuzzy supervisor. In the diagram, the CBS_stat stands for a “CBS statistics”; there is exactly one instance of CBS_stat for each consistent belief set. This object manages historical data associated with the given CBS, and contains its feasibility factor. CBS_stat objects are organized in a double-linked list sorted by the current values of feasibility factors.

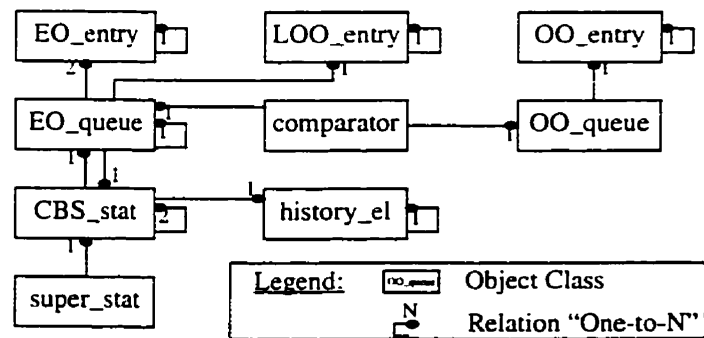


Figure 31: OMT Diagram of the Feasibility-Related Data Structures of a Fuzzy Supervisor

The EO_queue manages storage of expected output (points to the first and the last elements of a linked list of EO_entry objects) and the local copy of the observed output (linked list of LOO_entry). OO_queue stores output observed since the last matching, according to the new matching algorithm.

The Super_stat handles overall supervisor statistic data, such as the maximum feasibility, the average feasibility, etc., and points to the CBS_stat with the highest feasibility. Figure 32 illustrates data structures for a supervisor with four CBSes existing.

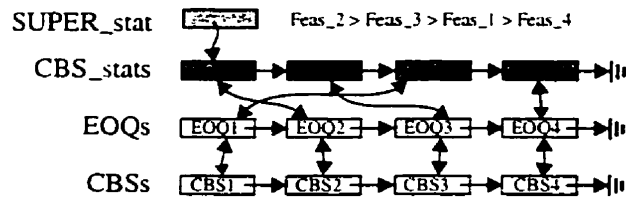


Figure 32: An Example: EOQs, CBSs, CBS Statistics Supervisor Statistics Objects in a Fuzzy Supervisor With Four Coexisting Belief Sets

Lastly, the history_el is an element of the captured past matching history, organized in a single-linked history tree discussed in the section 3.6.3. The latest element of the matched history for every given belief set is pointed to from its CBS_stat.

4.2 Evaluation Methodology

The following was the procedure for obtaining the empirical results:

- event trace acquisition: parameters of the load generator have been varied in order to produce the desired event distribution; the target system has been run with the hardware emulator and the load generator and all the input and output events have been logged;
- failure insertion: the desired number of failures of the chosen type have been seeded randomly within the original event trace;
- supervisor run: the software supervisor has been run with the event trace; supervisor reports have been collected, execution characteristics have been measured.

For each failure set and telephony load configuration combination tried during the evaluation, two to ten experiments have been conducted.

4.2.1 Variable Parameters

The following parameters were chosen to be variable:

- Telephony Traffic:
 - event distribution in time
 - number of originations per line per hour
- Failure types:
 - failure scenario types
 - number of failures injected

4.2.1.1 Telephony Traffic Simulation

Traffic has been simulated with the help of the software described in [1] and significantly enhanced in [21].

The software works with the hardware emulator and supports several modes of traffic simulation, allowing for varying these characteristics:

- number of origination attempts per hour
- maximum number of simultaneous calls in progress

Two traffic generation modes were used: Poisson and limited. Limited traffic limits the number of events that can take place in any given period of time. In Poisson traffic event occurrences have Poisson distribution, thus providing a more random, life-like traffic. More information on traffic simulation can be found in [3].

Different traffic levels were used to compare the degradation of the Fuzzy Supervisor performance (increase of computational load required to conduct supervision) to the degradation of performance of the standard supervisor with load increase. The increase of computa-

tion load causes the degradation of performance of the supervisor when computational capacity is exhausted.

The telephone traffic intensity used in tests have been set from four to six originations per line per hour that is comparable or higher than industry standards specify for PBX exchanges.

4.2.1.2 Failure Seeding

The following mechanism of failure seeding has been used for the evaluation purposes. Fuzzy supervisor has been augmented with event recording/replaying capabilities, that allowed to capture into a log file the input and output signals coming from the abstractor. The observed events have been time-stamped and retained. The captured log could be replayed by the supervisor in stand-alone mode with the effect of supervising a real target system at the time of event log capture. The captured event log file could be modified to simulate various failures of the target system.

Several representative types of failures has been simulated (derived from industrial telecom acceptance tests):

- observation failures (phantom outputs detected by abstractor)
- control program failure scenarios:
 - unprovoked ringing detected
 - no dial tone when receiver goes off-hook
 - idle tone not provided after the first digit is dialed
 - audible ringing tone not provided, termination successful
 - no power ringing on terminator line, termination incomplete
 - no voice path on answer

Note, that unprovoked physical ringing manifests itself in the same way as the observation failures but in real life is caused by a fault in the control program. The failure rates that are found acceptable in the telecommunication industry are less than 1 per 1000 calls. The failure rates adopted in the supervisor evaluation runs have been set 10 to 100

times higher to fully evaluate the capacity of the supervisor. A regular test run consisted of a 1/2 to 1 hour telephone traffic run with several failures of the same type seeded randomly in the replay event log. Failure mixes were not used. Since the experience has shown that the resynchronization has been achieved in most experiments, the previous history of supervision was of no relevance.

4.2.2 Observed and Derived Characteristics

The following data was collected in the conducted experiments:

- Number of belief sets.
- Maximum feasibility within the population of belief sets.
- Failure reports.
- Occurrence of computation overload, a condition when supervisor was unable to evaluate the observed behaviour of the target system in the real time mode.
- Resynchronization with the target system following a failure occurrence: new failures must be reported correctly and no false failures should be detected.
- Failure explanation via history traceback:
 - inputs
 - matches
 - mismatches
 - in-transit signal expirations.

4.3 Evaluation Results

Various types of supervisor reports have been collected and effectiveness of failure detection by the fuzzy supervisor has been investigated.

The collected reports and the results of the evaluation are discussed below.

4.3.1 Results Summary

The following table summarizes results of experiments performed with the Fuzzy Supervisor. It includes data on failure types and frequency, load intensity, and data on demonstrated resynchronization and failure detection capabilities, and coarse computational intensity characterization.

TABLE 2. Experimental Results for Failure Detection Capability

Failure Type	Random Traffic Load (1/2 hour trace)									
	Limited Load					Poisson Load				
	Number of Failures Seeded	Origination Rate	Failures Detected	Resynchronization	Computation Overload	Number of Failures Seeded	Origination Rate	Failure Detected	Resynchronization	Computation Overload
Phantom signals	10	2	10	Y	N	10	2	10	Y	N
	20	4	20	Y	N	20	4	20	Y	N
	50	4	50	Y	N	50	4	50	Y	N
	100	6	100	Y	N	100	6	100	Y	N
	200	6	200	Y	N	200	6	200	Y	N
Unprovoked Ringing	10	2	10	Y	N	10	2	10	Y	N
	20	4	20	Y	N	20	4	20	Y	N
	50	4	50	Y	N	50	4	50	Y	N
	100	6	100	Y	N	100	6	100	Y	N
	200	6	200	Y	N	200	6	200	Y	N
No Voice	1	6	1	Y	N	1	6	1	Y	N
	2	6	2	Y	N	2	6	2	Y	N
	3	6	3	Y	N	3	6	3	Y	N
	5	6	5	Y	N	5	6	5	Y	N

TABLE 2. Experimental Results for Failure Detection Capability

Failure Type	Random Traffic Load (1/2 hour trace)									
	Limited Load					Poisson Load				
	Number of Failures Seeded	Origination Rate	Failures Detected	Resynchronization	Computation Overload	Number of Failures Seeded	Origination Rate	Failure Detected	Resynchronization	Computation Overload
No Power Ringing	1	6	1	Y	N	1	6	1	Y	Y
	2	6	2	Y	N	2	6	2	Y	Y
	3	6	3	Y	Y	-	-	-	-	-
No Ring Tone	1	6	1	Y	N	1	6	1	Y	Y
	2	6	2	Y	N	2	6	2	Y	Y
No Dial Tone	1	6	1	Y	Y	1	6	1	Y	Y
	2	6	1	N	Y	2	6	1	N	Y
No Idle Tone	1	6	1	Y	Y	1	6	1	Y	Y

4.3.2 Supervisor Reports

Figure 33 contains a visualization of some supervisor reports collected when the target system exhibited a number of simple failures: several expected output signals were either not produced, replaced with other signals, or unexpected signals were produced.

Figure 33 represents the following reports:

- Event history: all input signals and successfully matched outputs have been reported. The mismatched outputs have been reported to explain the nature of the occurred failure: not shown in the diagram are the path properties of the mismatched signals that were collected also to assist in failure troubleshooting. Note: Figure 33 does not name signals, while providing only internal signal IDs. The mapping between the names and IDs is one-to-one, though.
- Number of belief sets: report that shows the degree of non-determinism of operation of the target system as a function of time. Informally this showed the degree of confusion of the supervisor over the state of the target system. It also expressed the level of activity in the target system. At the end of the diagram, an injected failure caused a rapid increase of number of belief sets as a result of temporary missynchronization.
- Maximum feasibility as the estimated health of the target system has been captured and is shown in a graph.
- Failure occurrence reports: every time the maximum feasibility drop has been encountered, a failure report has been produced. Reports occurrences are summarized in one graph.

Seeded failure occurrences are shown in a graph for reference purposes. In the example all injected failures have been successfully detected.

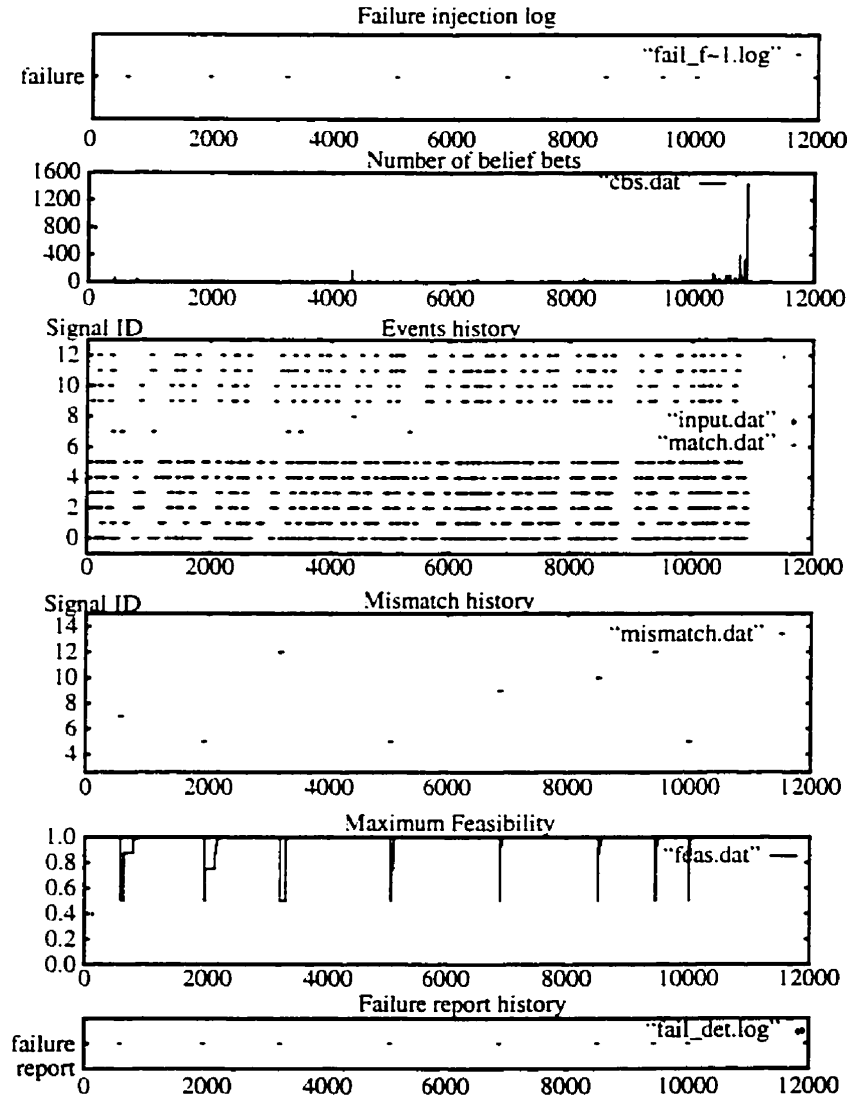


Figure 33: Fuzzy Supervisor Reports During Output Failures of the Target System

4.3.3 Failure Detection Capability

Various failures of the target system have been seeded by modifying the trace sequence of input and output signals observed by the supervisor on a correctly executing target system. Measurements have been taken with different types of loads, and different classes of failures. Impact on the supervisor's failure detection capability has been observed.

As seen in the summary table, seeded phantom signals did not cause a computational overload during supervision at any rate of injection, but missing output signals caused supervisor to slow down. For these, the rate of failure injection was kept to the minimum.

It has been noticed that only when unexpected output signals were seeded, a full resynchronization has been achieved with no or little impact on the number of belief sets generated by the supervisor.

When output signals have not been produced, or incorrect signals have been produced instead of expected ones, resynchronization capability was sometimes affected due to snowballing of belief sets number. In 21 scenarios resynchronization has been achieved, only in two the supervisor reported a constantly decreasing health along with steadily growing number of belief sets.

All failures that have been experimented with have been detected and reported correctly in single or sequential occurrences.

4.3.4 Resynchronization and Continuous Supervision

Along with failure detection, the resynchronization capability of the supervisor has been observed.

Resynchronization was considered achieved when capability for correct detection of failures has been restored after a seeded failure has been observed. All failures were seeded, thus known; no spurious fail-

ures must be reported, and all seeded failures must be reported in order to claim resynchronization.

None of the seeded failure types has shut the supervision down, thus indicating the achievement of continuous supervision. Computational overload situations have occurred when idle, dial or ring tones signals were removed from the replay log of a irregularly spaced high load traffic (Poisson traffic).

Resynchronization capabilities of a fuzzy supervision have been demonstrated in the experiments. No false failures have been detected after real failures occurred. The number of belief sets did not always reduce to the levels of a failure-free supervision. This phenomenon has been expected, as no dedicated support for intelligent resynchronization has been devised. Lack of such a dedicated mechanism was a partial cause of supervision slowdown on certain types of failures.

It has been noticed that an even spacing of events in time (provided with limited traffic), helps the fuzzy supervisor to recover from an observed failure. Poisson traffic has been proven much harder to supervise.

4.3.5 Computational Complexity Observations

The growth of number of belief sets during failure-free supervision with the increase of traffic has been comparable to that in the standard belief-based supervision approach up to the moment of failure detection, as expected. Also, no erroneous failure reports have been produced during traffic runs of various intensity.

When input signals to the target system were missed (removed from the replay log) or misinterpreted (replaced by other signals in the replay log) by the supervisor, the resynchronization was not always complete and sometimes was resulting in drastic supervisor slowdown due to a major increase of number of belief sets after an in-transit expiration of the misinterpreted input signal. This is an expected

it was only based on the observable signals, and could not be propagated to the internal elements of the specification. Methods that would be able to provide deeper analysis of the failure are complex and were not covered within this work.

V

Further Research and Conclusions


5.1 Further Research Directions

The fuzzy supervision approach is much wider than what has been covered in this work. It can be linked to research in probability theory and research in artificial intelligence. It would be interesting to establish such links in the future research.

The algorithms explored in this work are fairly simple. It would be valuable to further investigate the intelligent cutoff threshold management, mismatch penalty and match gain functions for different failure scenarios and particularly to observed signal loss scenarios where performance of the supervisor was less than ideal. Further development is needed to improve supervision capabilities for the Poisson traffic.

The resynchronization properties of fuzzy supervision have not been completely examined in this work. Some of the related issues should be looked into in the future. It would be interesting to expand the definition of feasibility to the domain of hierarchical target systems.

Appendix A: An SDL Specification of the Small PBX

 rw /amd_mnt/swen10/u/avorobie/Supervisor/specs/sdl/pbx_spec.sdt


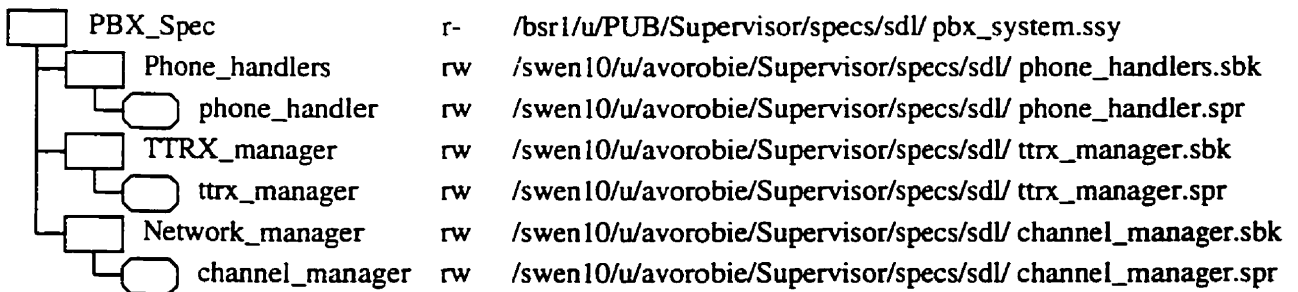
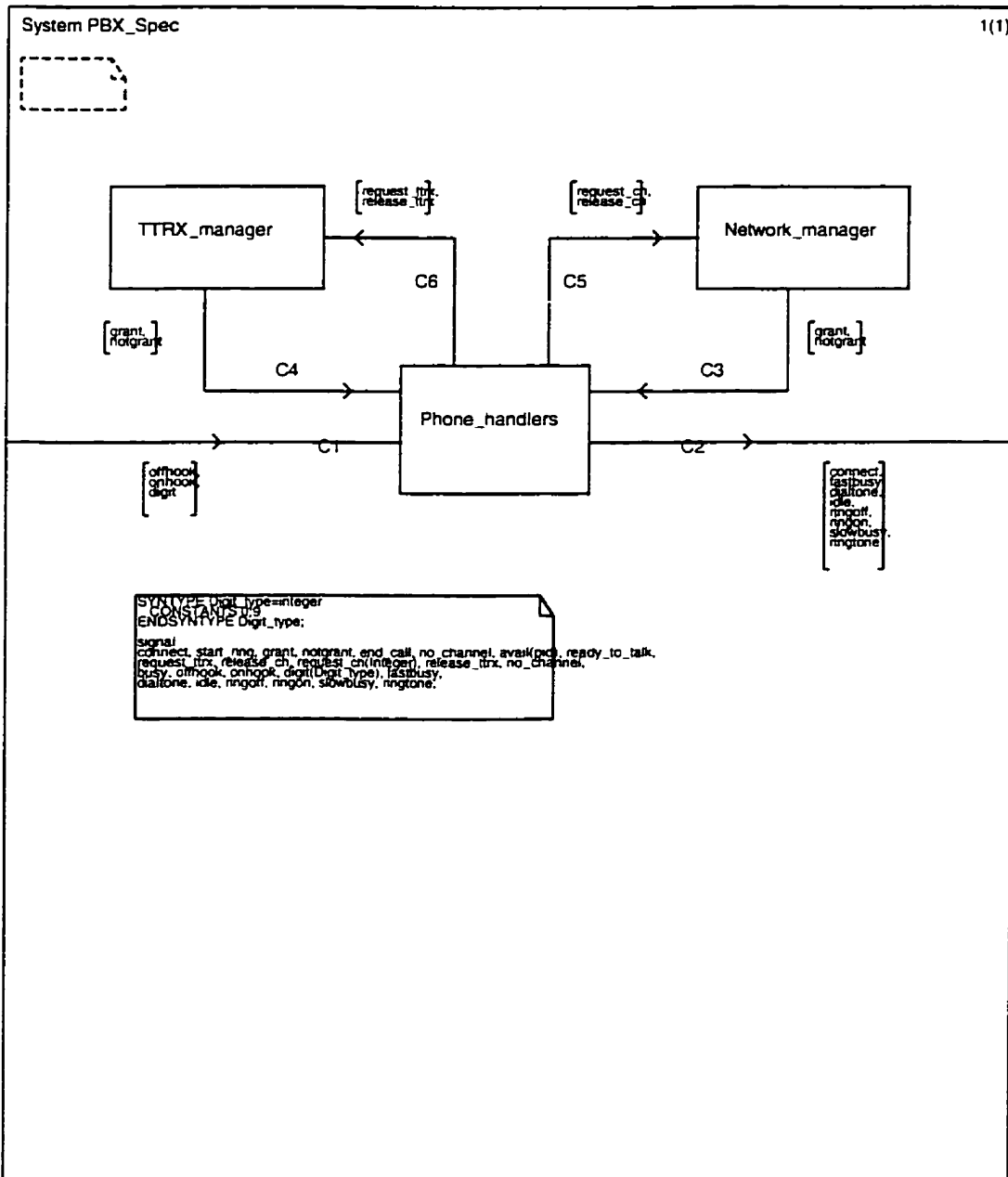
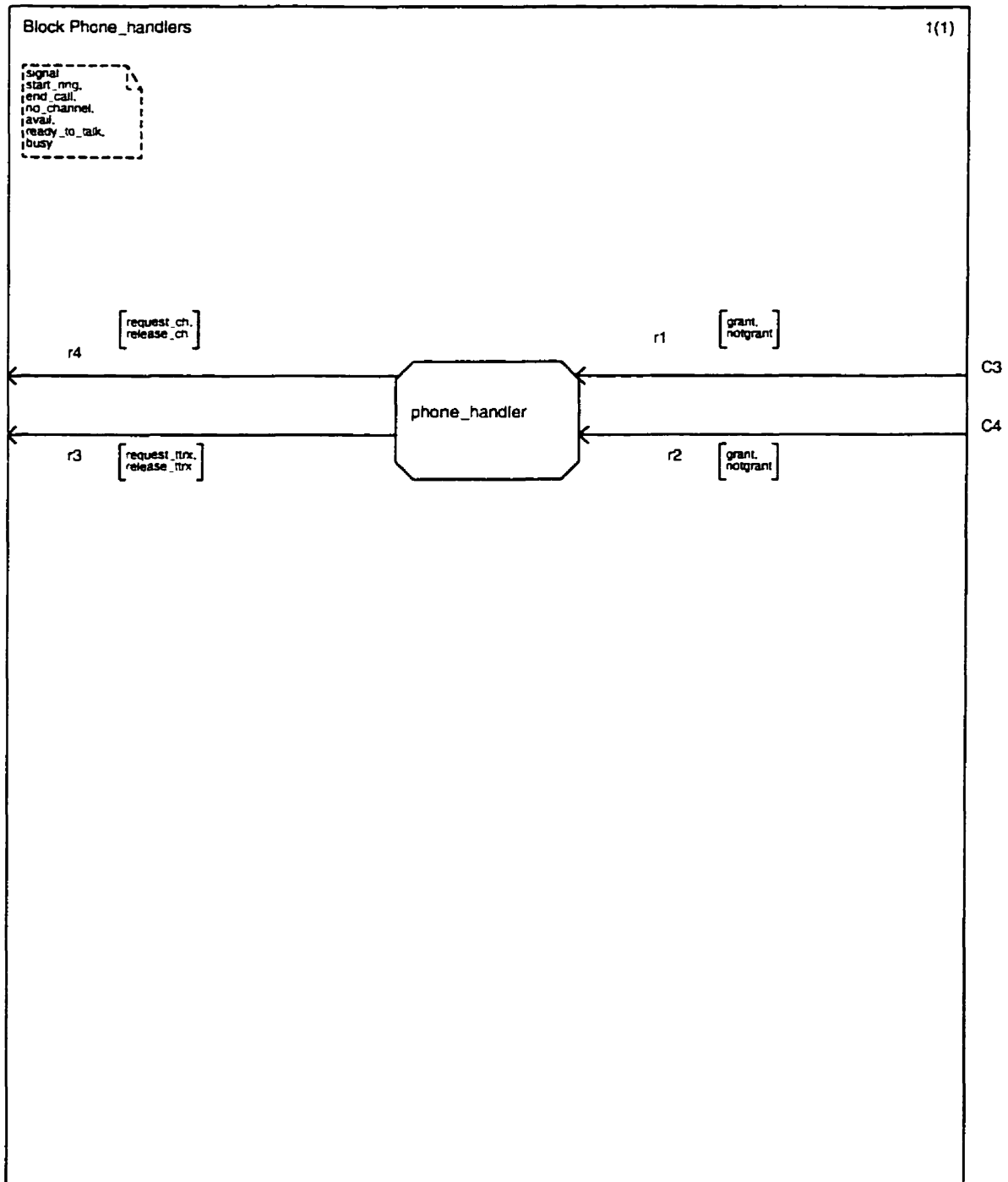
 rw /amd_mnt/swen10/u/avorobie/Supervisor/specs/sdl/

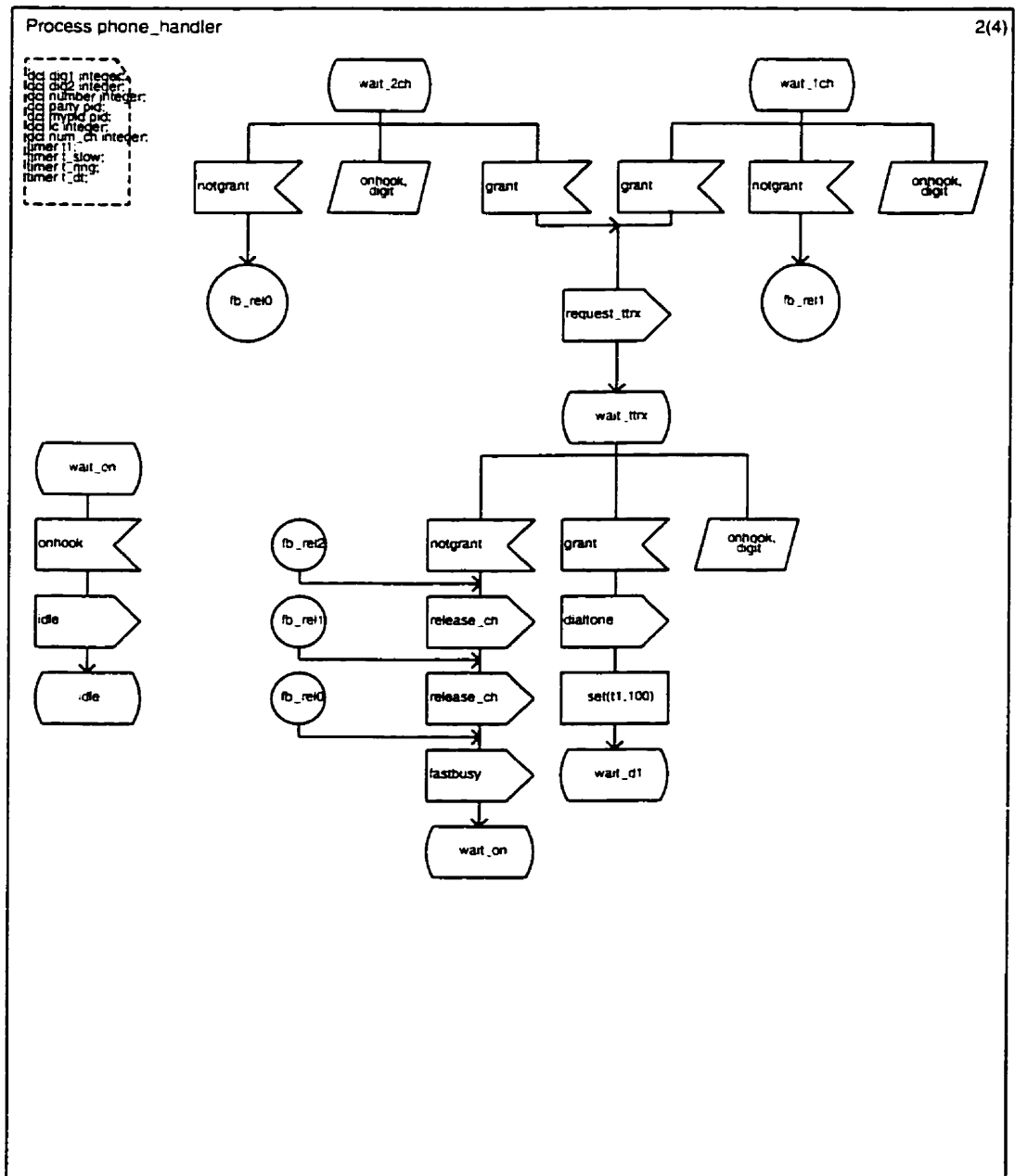
Diagram Structure

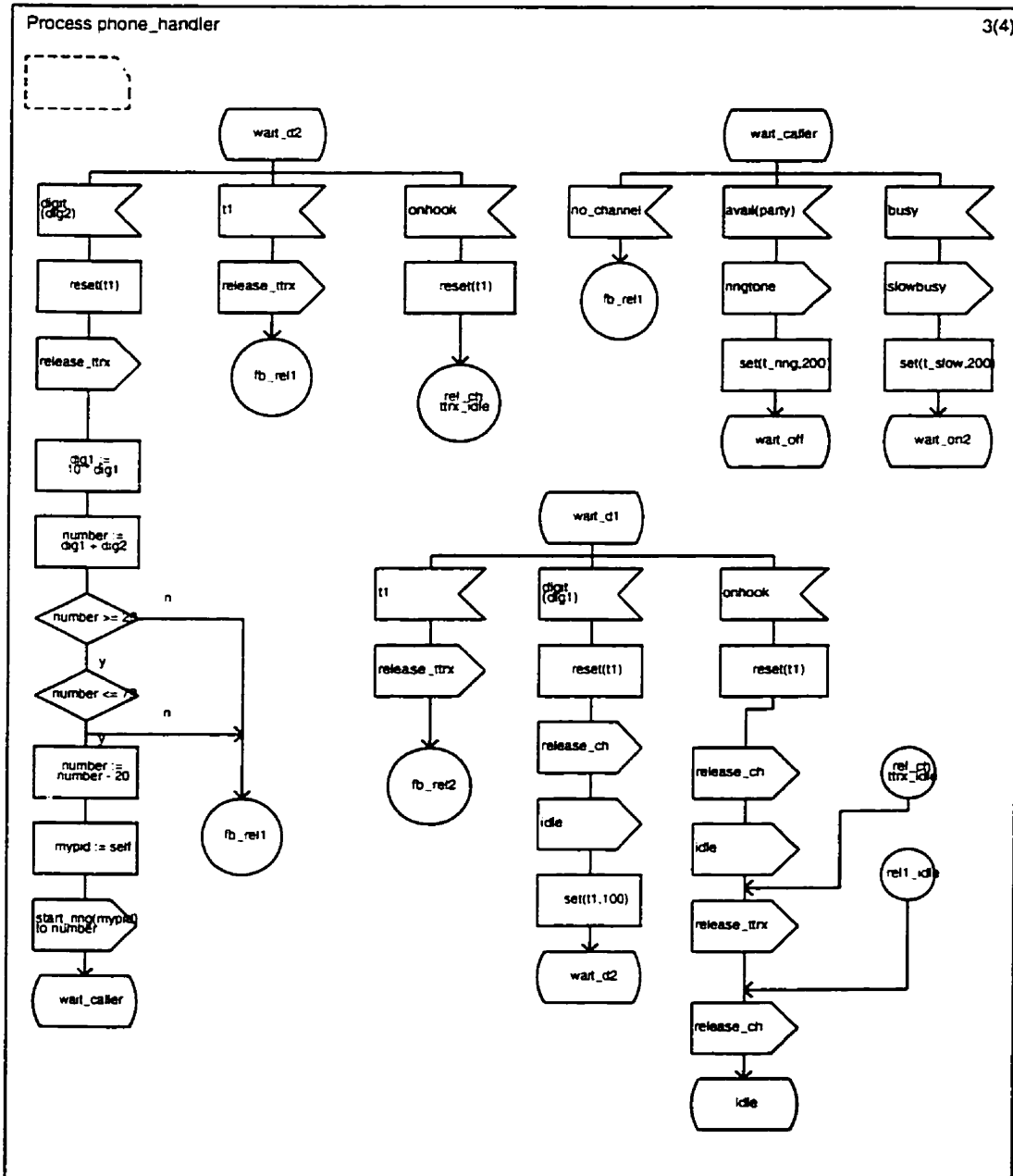


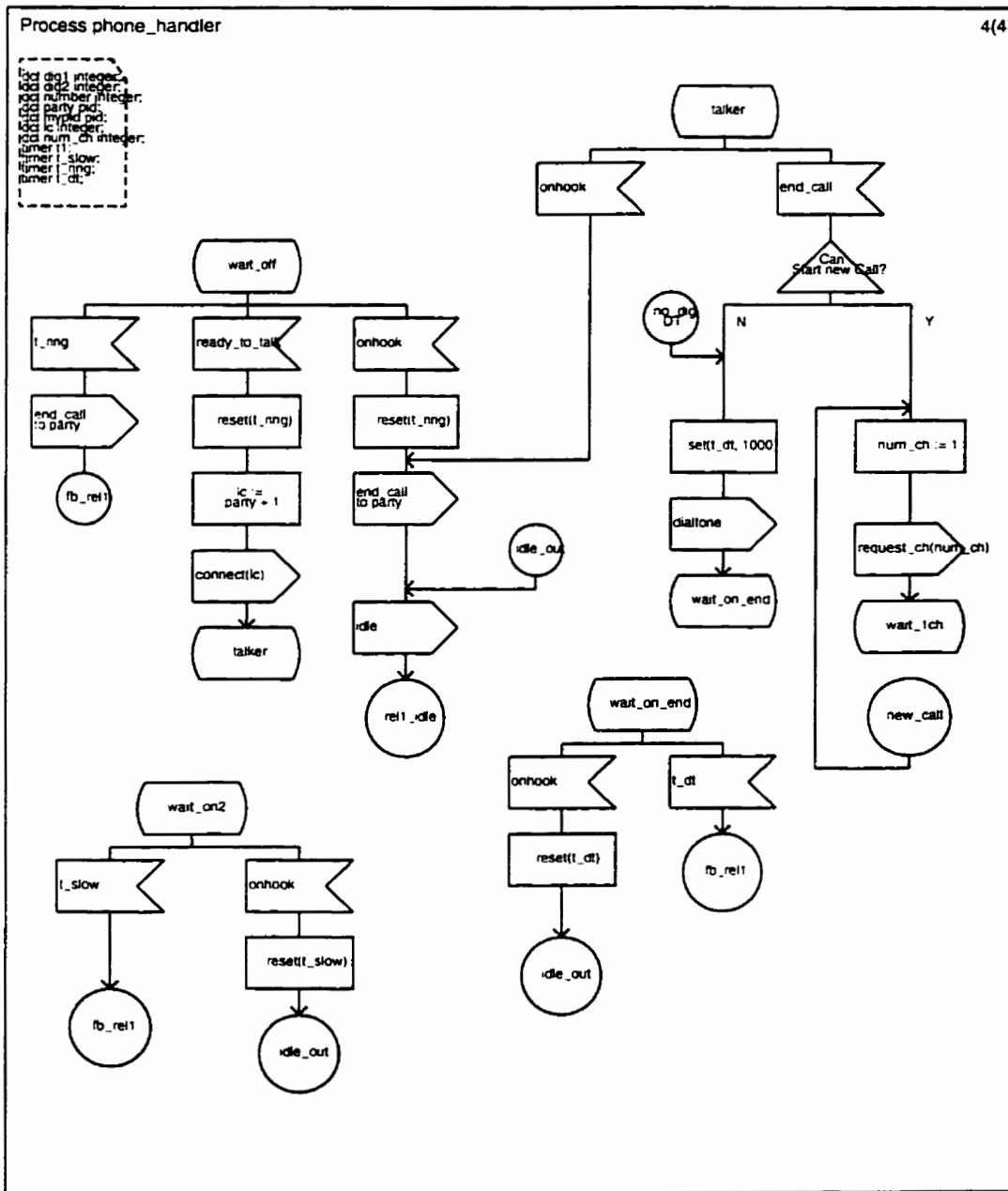
Associated Documents

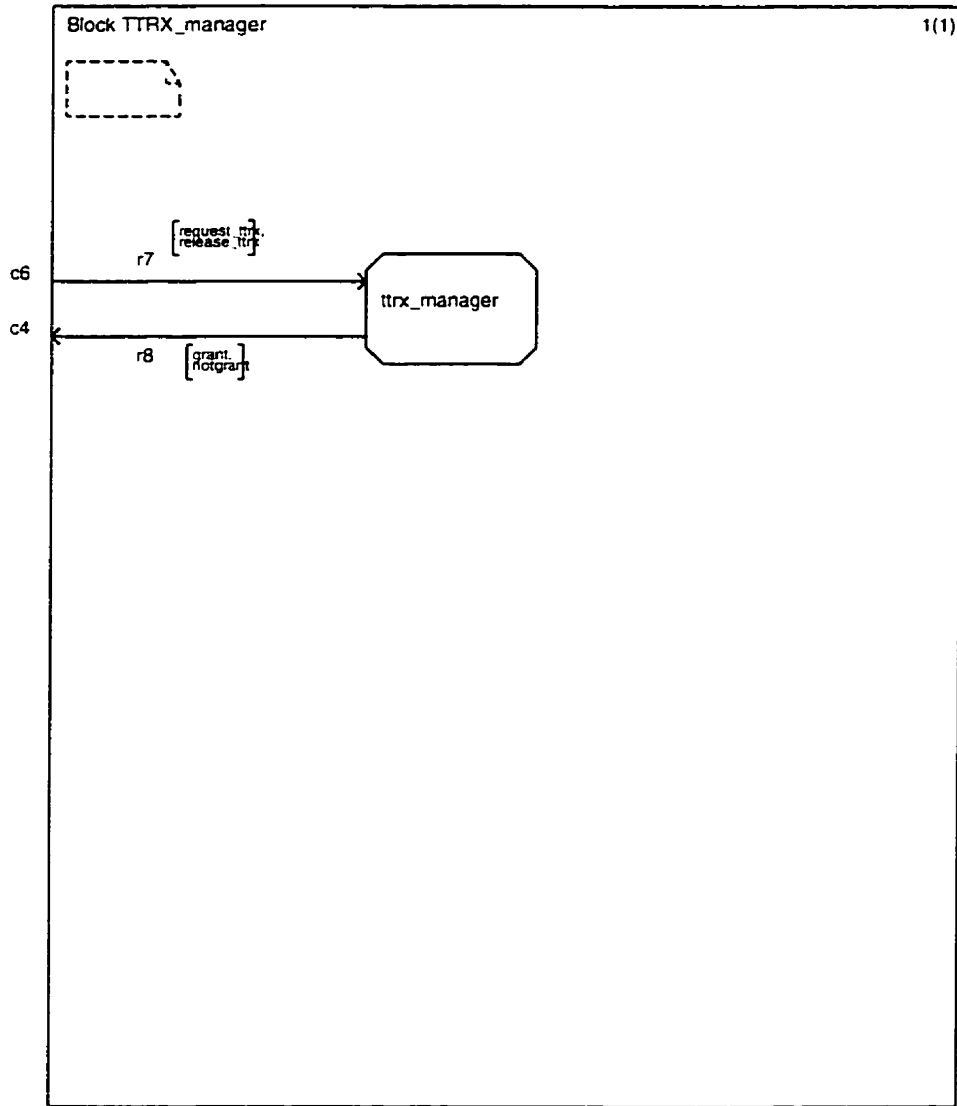


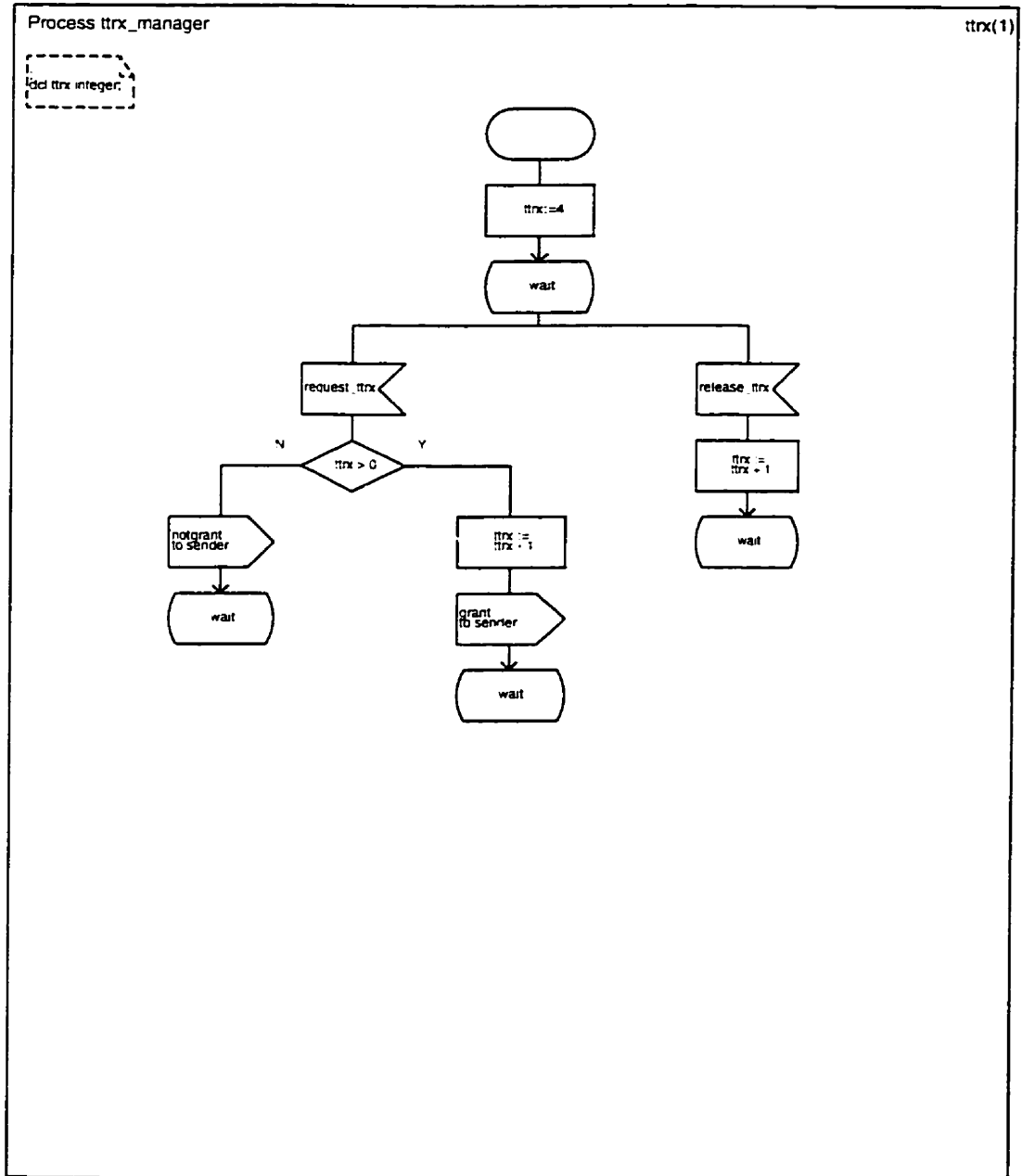


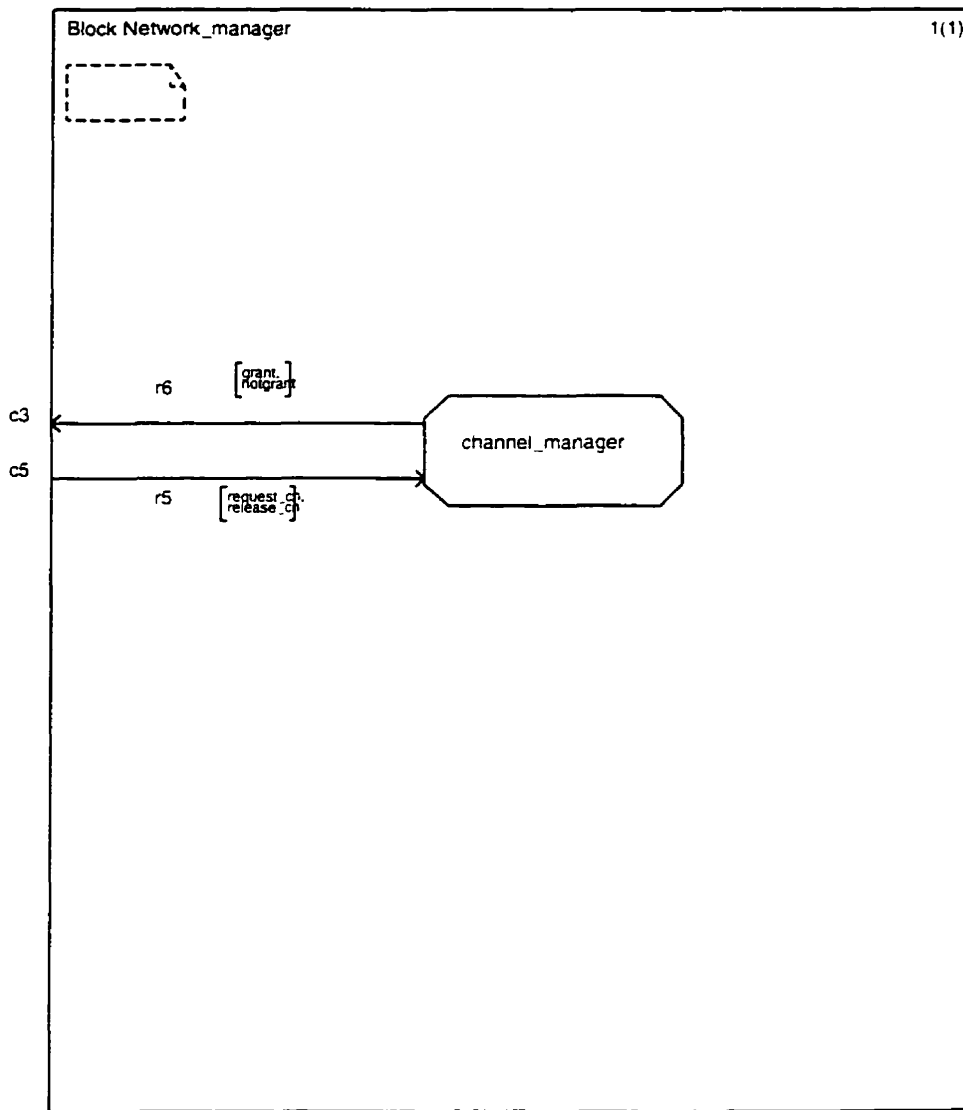


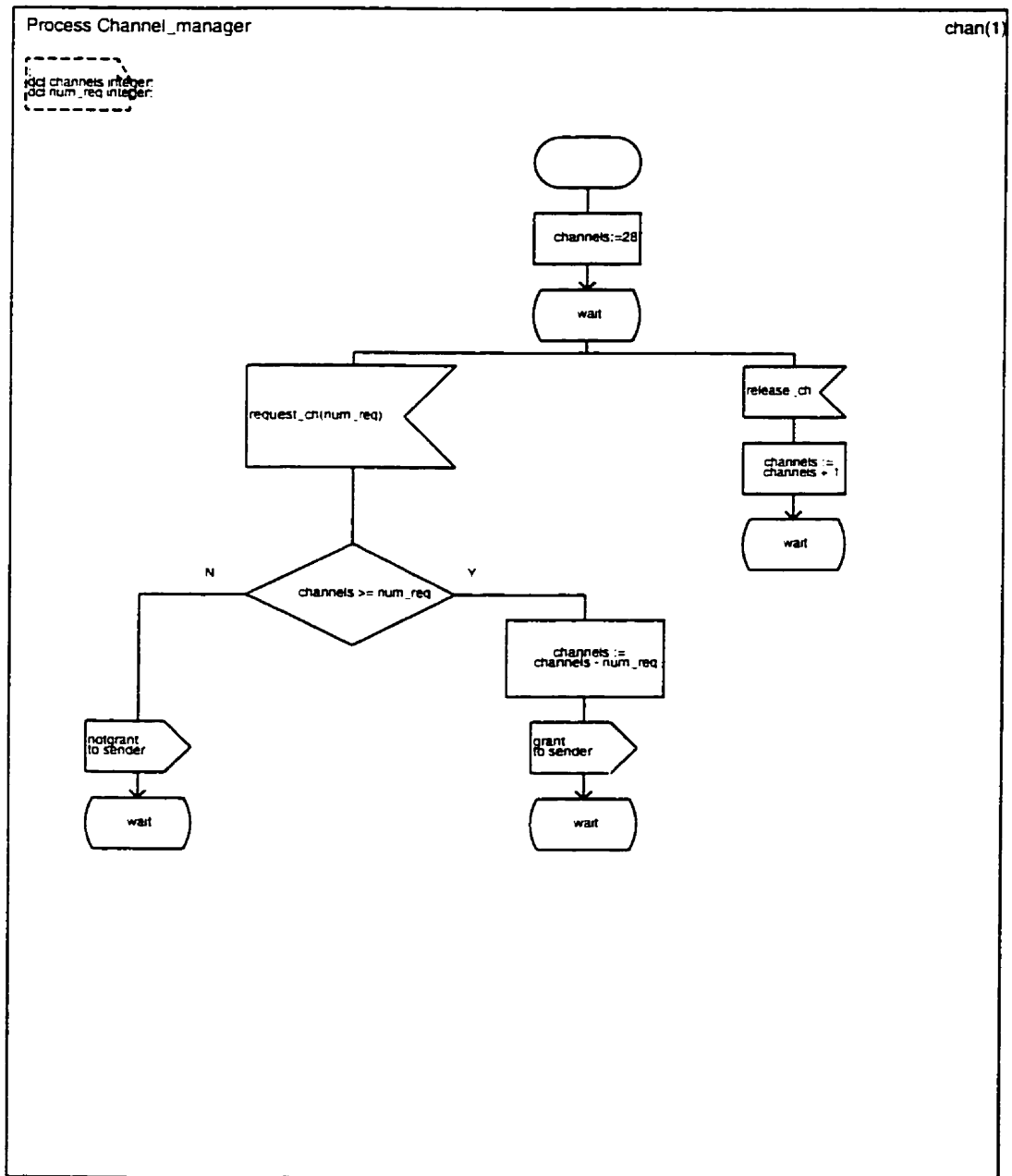












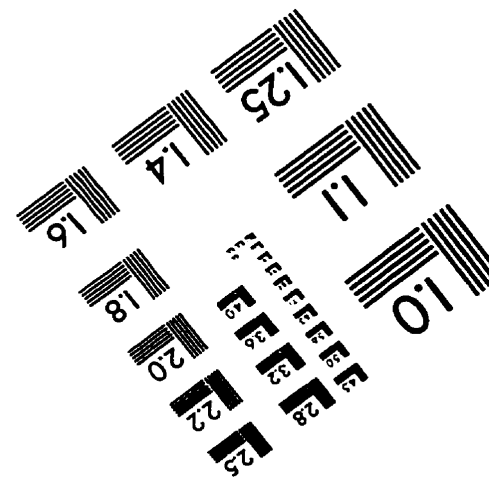
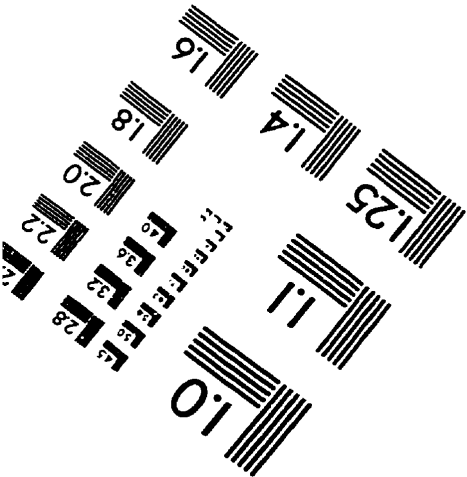
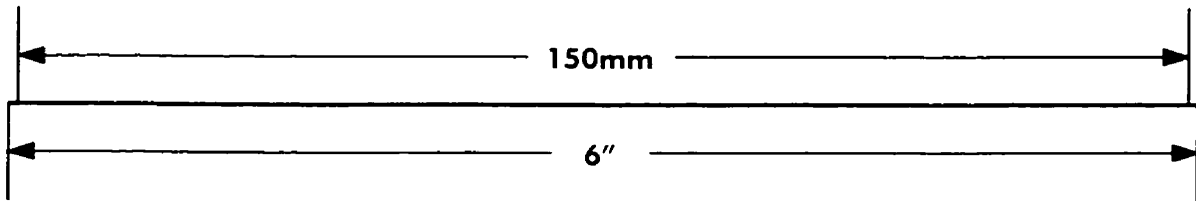
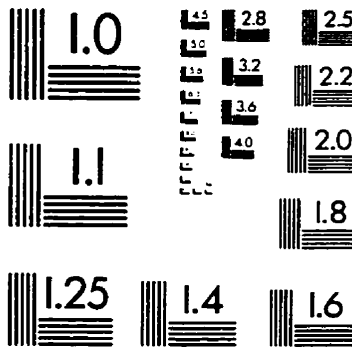
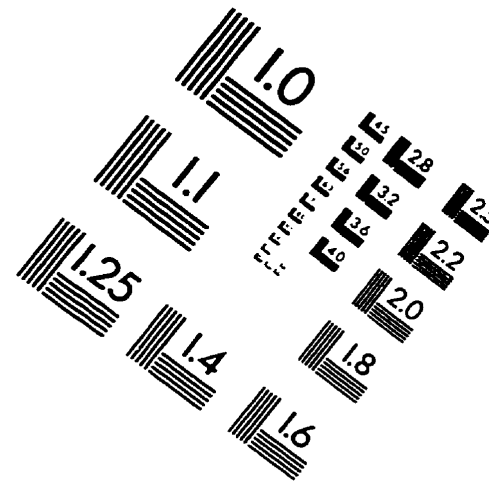
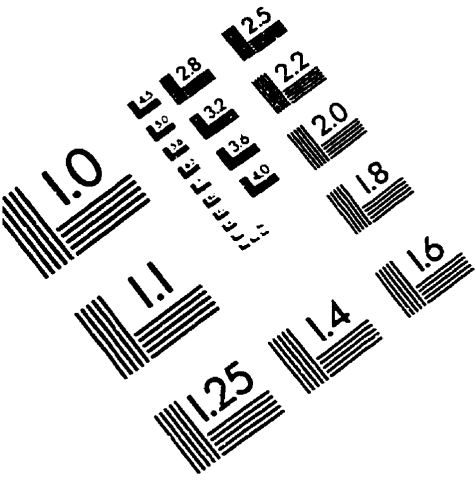
References

-
- [1] D.B. Hay, "A Belief Method for Detecting Operational Failures in Soft Real-time Systems." (University of Waterloo, E&CE Masters Thesis, 1991.)
 - [2] M.Hlady, R.Kovacevic, J.J.Li, B.R.Pekilis, D.Prairie, T.Savor, R.E.Seviora, D.Simser and A.Vorobiev, "An Approach to Automatic Detection of Software Failures." (Technical Report, Bell Canada Software Reliability Laboratory, 1994.)
 - [3] M.Hlady, R.Kovacevic, J.Li, D.Prairie, S.Reid, "An Implementation of a Belief-Based Real-Time Supervisor." (Bell Canada Software Reliability Laboratory Technical Report, Aug. 1995.)
 - [4] D.B.Hay, R.E.Seviora, "A Real-Time Validator." (Proceedings of the Third IEE International Conference on Software Engineering for Real-Time Systems", IEE, 1991.)
 - [5] R.Iorgulescu, R.E.Seviora, "A Resynchronization Method for Real-Time Supervision." (Proceedings of the 6-th. Euromicro Workshop on Real-Time Systems, June 1994.)
 - [6] Ferenc Belina, Dieter Hogrefe, Amardeo Sarma, "SDL With Applications From Protocol Specification." (Prentice Hall, 1991.)
 - [7] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software." (IEEE Transactions on Software Engineering, Vol. SE-11, No. 12, December 1985.)
 - [8] P.A. Lee, T. Anderson, "Fault Tolerance: Principles and Practice." (2nd ed., Springer Verlag, 1990.)

-
-
- [9] J.J.Horning et al., "A Program Structure for Error Detection and Recovery." (Lecture Notes in Computer Science 16, p.171-187, Springer-Verlag, 1974.)
 - [10] Ali, R., Shipp, N., Swaminathan, A., "PBX Software Requirements Specification." (Course Project, E&CE455, University of Waterloo, Spring Term, 1993.)
 - [11] Ali, R., Shipp, N., Swaminathan, A., "PBX Software Design Description." (Course Project, E&CE455, University of Waterloo, Spring Term, 1993.)
 - [12] International Telegraph and Telephone Consultative Committee, "Functional Specification and Description Language, Recommendations Z.100-Z.104 (Blue Book)." (ITU, 1989.)
 - [13] D. Hogrefe, A. Sarma, "Nondeterminism and SDL." (Proc. 2nd Int'l Conference on Formal Description Techniques, pp 157-167, 1989.)
 - [14] S.Leue, "Specifying Real-Time Requirements for SDL Specifications - A Temporal Logic-Based Approach." (Proceedings of the 15th International IFIP WG6.1 Symposium on Protocol Specification, Testing, and Verification, Warsaw, June 1995.)
 - [15] K.-T. Cheng and A.S. Krishnakumar, "Automatic Functional Test Generation Using the Extended Finite State Machine Model." (Proceedings of the 30th Design Automation Conference DAC-93, pages 86-91, 1993.)
 - [16] R Kruze, "Foundations of Fuzzy Systems." (Chichester, West Sussex, England, Toronto: Wiley & Sons, 1994.)
 - [17] T.J.Ross, "Fuzzy Logic With Engineering Applications." (International ed.,New York: McGraw-Hill, 1995.)
 - [18] M. Diaz, "Observer-A Concept for Formal On-Line Validation of Distributed-Systems."(IEEE Trans on Software Engineering, Vol 20, No 12, pp. 900-912,Dec 1994.)
 - [19] C.Wang and M. Schwartz, "Fault Detection with Multiple Observers."(IEEE/ACM Trans on Networking, Vol 1, No 1, pp. 48-55, Feb 1993.)
 - [20] D. Butnartu, "Autonomous Evolutive Systems With Ambiguous States." (Fuzzy Logic in Knowledge-Based Systems, ElsevierScience Publishers B.V., North Holland, page 229, 1988.)
 - [21] S.Reid, "Reduced Model Supervision: Quantifying Trade-Offs in Failure Detection and Computational Complexity." (University of Waterloo, E&CE Master's thesis, 1996.)
 - [22] R.Kovacevic, "A Resynchronization Scheme for Belief-Based Real-Time Software Supervision." (University of Waterloo, E&CE Master's thesis, 1996.)
 - [23] M.Hlady, "A Real-Time Software Supervisor with Failure Retraction Capabil-

-
-
- ities." (University of Waterloo, E&CE Master's thesis, 1995.)
- [24] B.R.Pekilis, "Automatic Monitoring of Response Time Performance in Soft Real-Time Systems." (University of Waterloo, E&CE Master's thesis, 1995.)
- [25] R.A.Ali, "A Gray-Box Approach to Software Supervision." (University of Waterloo, E&CE Master's thesis, 1996.)
- [26] A.Petryk, "Event Trace Driven Software Failure Detection.", (University of Waterloo, E&CE Master's thesis, 1997.)

IMAGE EVALUATION TEST TARGET (QA-3)



APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved