A RULE-BASED DOMAIN SPECIFIC LANGUAGE FOR FAULT MANAGEMENT

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ÖZGÜR KAYA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

FEBRUARY 2014

Approval of the thesis:

A RULE-BASED DOMAIN SPECIFIC LANGUAGE FOR FAULT MANAGEMENT

submitted by **ÖZGÜR KAYA** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** _______________

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** _______________

Prof. Dr. Ali H. Doğru
Supervisor, **Computer Engineering Department, METU** _______________

Examining Committee Members:

Prof. Dr. Göktürk Üçoluk
Computer Engineering Department, METU _______________

Prof. Dr. Ali H. Doğru
Computer Engineering Department, METU _______________

Prof. Dr. Ahmet Coşar
Computer Engineering Department, METU _______________

Assoc. Prof. Dr. Pınar Karagöz
Computer Engineering Department, METU _______________

Assist. Prof. Dr. Ayça Tarhan
Computer Engineering Department, Hacettepe University _______________

Date: _______________

ABSTRACT

A RULE-BASED DOMAIN SPECIFIC LANGUAGE FOR FAULT MANAGEMENT

Kaya, Özgür
Ph.D., Department of Computer Engineering
Supervisor : Prof. Dr. Ali H. Doğru

February 2014, 181 pages

A fault management framework has been developed where a rule-based event processing language is also developed that provides improvement to the existing approaches in terms of time responsiveness. Reference architectures were developed for the fault management domain including fault avoidance capabilities. Such capability is for taking precautionary actions before the fault happens, while most of the fault tolerance techniques are intended for detecting a fault after it happens, hence utilizing the time with less efficiently. High availability is targeted through such measures for mission-critical systems. The need for this study was realized when a family of products were planned for different mission-critical systems to support them by different Fault Management subsystems. A real-time event-processing rule-based language and its processing tools were defined as a requirement during this work and final contribution was dedicated to this area. Requirements for the language did not include high demands on the inference capabilities and very high-level declarative logic, however, its response in the environment that deals with events at the computational platform level was important. The language and its processor were validated with tests and based on the available similar results in the literature, performed superior especially considering its specific purpose.

Keywords: Fault Management, Fault Avoidance, Domain Specific Languages, Rule Based Languages, Checkpointing, Software Product Lines, Event Processing

ÖZ

HATA YÖNETİMİ İÇİN KURAL TABANLI ALANA ÖZEL BİR DİLİ

Kaya, Özgür
Doktora, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Ali H. Doğru

Şubat 2014 , 181 sayfa

Zaman duyarlılığı cinsinden mevcut yaklaşımlardaki gelişimi sağlayan kural tabanlı olay işleme dilinin geliştirildiği yerde bir hata yönetimi çerçevesi de geliştirilmiştir. Hatadan kaçınma yetenekleri içeren hata yönetimi alanı için referans mimariler geliştirilmiştir. Zamanın daha az verimli kullanılmasından dolayı, çoğu hata toleransı teknikleri hatayı oluştuktan sonra tespit etmeye yönelikken, bu tür yetenekler hata oluşmadan önce gerekli tedbirleri almaya yöneliktir. Görev kritik sistemler için bu tür önlemler sayesinde hedeflenen, yüksek dereceli kullanılabilirliktir. Farklı görev kritik sistemler için bir ürün ailesi planlandığında onları farklı hata yönetimi alt sistemleri ile desteklemek için bu çalışmaya gereksinim fark edilmiştir. Gerçek zamanlı bir olay işleme kural tabanlı dil ve onun işleme araçları, bu çalışma ve final katkısı sırasında, bu alana ithaf edilmiş bir gereksinim olarak tanımlanmışlardır. Dil gereksinimleri, çıkarım yetenekleri ve çok yüksek derecede beyan edilen mantık üzerinde yüksek talepleri içermezler, buna rağmen onun hesaplamalı platform düzeyindeki olaylarla ilgilenen çevredeki tepkisi önemlidir. Dil ve onun işlemcisi testlerle geçerli kılınmış ve literatürdeki uygun benzer sonuçlara dayandırılmış, özellikle geliştirme amacı düşünülecek olursa, üstün bir şekilde gerçekleştirilmiştir.

Anahtar Kelimeler: Hata Yönetimi, Hata Önleme, Alan Özel Diller, Kural Tabanlı Diller, Denetim Noktası, Yazılım Üretim Bantları, Olay işleme

To my large and perfect family.

ACKNOWLEDGMENTS

I would like to thank my supervisor Professor Ali H. Doğru for his constant support, guidance and close friendship. It was a great honour to work with him for the last seven years and our cooperation influenced my academical and world view very highly.

And I would like to thank to Professor Adnan Yazıcı for his support.

A lot of people influenced and supported this work scientifically and their contribution were most valuable for me.

My large family also provided invaluable support for this work. I would like to thank specially to my wife Harika Firuzan Kaya. She always make me feel loved and cared. I am also thankful for all the love and support by Şaduman Kaya, Mehmet Kaya, Ayça Kaya, Hülya Odabaş, Celal Odabaş, Şeyda Odabaş, and my lovely son Çağan Toprak Kaya.

And there are a lot of people that were with me in these eight years. They defined me, they made me who I am, they are true owners of this work. It is not possible to write down why each of them is important to me and this work, because it will take more space than the work itself. So I'll just give names of some of them; Selma Süloğlu and Hande Çelikkanat.

And finally very special thanks to a group of people who've taught me real meaning of friendship lately. Special thanks to the residents of the most joyful office ever, A-210 Admin Office : Gökdeniz Karadağ, Can Eroğul, Merve Aydınlılar, Hüsnü Yıldız, and İlkcan Keleş.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AIS	Active Instance Stacks
AIS	Application Interface Specification
API	Application Programming Language
AST	Abstract Syntax Tree
AOF	Activation Order Flow-graph
BC	Backward Chaining
CCE	Core Composite Event
CEP	Complex Event Processing
CP	Chekpoint
CTR	Concurrent Transaction Logic
ESB	Enterprise service Bus
DSE	Domain Specific Engine
DSK	Domain Specific Kits
DSM	Domain Specific Motor
DSL	Domain Specific Language
DST	Domain Specific Tool
EA	Event Action
ECA	Even-Condition-Action
ETA	Event Tree Analysis
EPS	Event Processing Service
FC	Forward Chaining
FM	Fault Management
FMS	Fault Management Systems
FMEA	Fault Modes and Effects Analysis
FTA	Fault Tree Analysis
FTS	Fault Tolerance Systems
HA	High Availability
HPI	Hardware Platform Interface

KSL	Knowledge Specification Language
LHS	Left Hand Side
MTBF	Mean Time between Failure
MTTR	Mean Time to Repair
NFA	Non-deterministic Finite Automata
OPS	Official Production System, Version
PRS	Procedural Reasoning System
RAL	Rule-Extended Algorithmic Language
RHS	Right Hand Side
RTP	Real Time Process
SA	Service Availability
SAM	Safety Argument Manager
SFA	Software Factory Automation
SMP	Symmetric Multi-Processors
SNE	Stateful Network Equipment
SPL	Software Product Line
TNCES	Timed Net Condition Event System
UML	Unified Modeling Language
XML	Extended Mark-up Language

CHAPTER 1

INTRODUCTION

This thesis study was conducted in parallel with a research for developing a Software Product Line (SPL) to support the fast development of Fault Management Systems (FMS) to be used in Safety Critical Applications. Besides many components of the environment, an event-processing rule-based language and its processing tool were developed with a real-time responsiveness as the main contribution. Consequently, this study is related with a number of fields that are introduced partially in this Introduction Section and in the following Literature Survey Section.

1.1 Fault Management

Software carries defects. Except for trivially small and simple applications, it is not possible to prove that an application is error free and it is generally accepted that bugs cannot be completely eliminated. These errors also referred to as defects once the application is delivered to the user, cause faults during execution. For some applications, such faults are tolerable – starting the program all over is acceptable. However, for other applications, faults are not acceptable; they can lead to the termination of the mission, causing material losses and even human life. As a result, techniques are necessary for fault tolerant designs: since it is not possible to eliminate errors, effort should be dedicated to prevent their causing faults or to prevent the faults inflicting serious consequences to the system. It is possible to design software for tolerating faults [114].

Fault Tolerance Systems (FTS) are often related to high availability that is especially important in many safety critical, mission-critical, and real-time systems. In order for a system to have fault-tolerance capability, the system should operate in the presence of software or hardware failures. In case of hardware failures, the system with fault tolerance will avoid a system crash or interrupt through switching the faulty component with a spare one transparently [55]. This type of an FMS corrective action will incur additional costs. Such problems are results of fault-error-failure chains. Hence if the chain is broken before the failure occurs, while the cost of the system is decreased availability of the system is also increased. In order to provide this fault prevention feature, some precautionary actions should be taken before the fault occurs. Such an action may be periodically conducted, without necessarily waiting to detect any fault conditions. These conditions can cause frequent intervention such as switching a redundant unit or restarting a functioning software unit those resulting in extra operational loads. Common recovery techniques will be revered to including redundancy, with its variations such as N, 2N, N+1, N+M [55]. These techniques also include the cold, warm, and hot reboot, and checkpoint [20] options for saving state information and loading the state during reboot.

1.2 Software Product Lines

Generally software developers specialize in a certain area to be an expert on that field. Domain orientation and Software Product Line (SPL) engineering are basic specialization leverages for software developers. Embedded, real time, or safety critical systems are further complex software systems and organizations specialize on those systems to continue specialized kinds of development for long terms. Organizations usually employ domain engineering practices to support their SPL. This way it is convenient to manage the common and the variable constituents of their family of products. Also, to construct each product, besides the domain engineering, product engineering techniques efficiently help the organization to exploit the domain assets. Consequently, an organization involved in SPL continuously enhances its domain assets for its infrastructure and conducts product engineering for different products. Setting up the SPL has additional cost but this investment usually pays off after the development of a small number of products. Since a part of this study about the fault management domain, a domain model for fault management to satisfy the requirements for developing systems with varying high-availability demands have been constructed. Actually our target organization uses fault management as a supporting capability for its mission critical systems. The required domain seems to be a union of two separate domains (FMS and mission critical systems). However, rather than considering those as a united domain, it is more advantageous to consider them as two different domains, not completely isolated but affecting each other and still providing a lot of cohesion for either: Different mission critical systems will continue to be produced being within the core competency of the organization – an SPL of such is a different consideration. Fault management capabilities will be required in all those products. It was decided to have a fault management infrastructure that would support all their products with its own SPL that would help to configure a different fault management system for each product.

During this thesis work [61], we developed a fault management reference architecture that is also supporting fault avoidance capabilities as part of an SPL, to be integrated with real-time, embedded, and safety critical systems. Moreover, a feature model for the fault management domain has been developed and presented in terms of a feature tree diagram [60]. In order to provide variability management and programmable configurability in the architecture's components with Domain Specific Languages (DSL), the architecture was developed based on the Domain Specific Kits (DSK) of the Software Factory Automation [3] approach. Most of the capabilities have been implemented and recently the prototyping work started on the checkpoint capability [62]. Checkpoint is an important and difficult to implement capability that manages the saving of the state and recovering it for different units in a system. Therefore it was carried on separately and as the latest component of the fault management system to build [29].

Ideas presented here have been designed and developed and some applied in industrial settings. A distinctive feature is the fault prevention capability. A critical capability is checkpoint design and implementation is done, defines our novel approach due to its being achieved through a rule-based domain specific language.

Moreover, when a language was needed for fault management of mission critical systems, alternatives on many possible attributes quickly reduced to a relatively easy decision. Rule based paradigm was decided for its advantages in supporting domain specific expertise: Different fault management functions such as diagnosis or monitoring, needed separate but similar languages supporting the architecture that displayed a layered and distributed fashion. The facts and inference decision however, involved low-level concepts pertaining to the computational platform and some characteristics of the executing applications that could be classified as infrastructural. Besides the difficult task of processing events, the relative simplicity of inferences, small number of events and operation types characterize the requirements.

Performance speed is an important issue that lead this research to exploit the requirements toward the design of algorithms especially concerning the run-time. The architecture was decided to utilize the components of the Software Factory Automation (SFA) [3], [4] where our layers correspond to domains such as monitoring, each supported with a Domain Specific Language, and a specific interpreter for that language (Domain Specific Engine).

1.3 Language Related Considerations for this Research

The architecture, together with the rules running on it, needed to quickly deploy to new applications of similar missions. Early experimentation with the ideas resulted with acceptable outcome for the verification of the architecture [61]. When it was time to support this development with higher-level rule based languages, a quick evaluation of the state-of-the-art suggested a fast and reliable processing of events, yet performing along the newer improvements on the rule-based language processing. The present times are witnessing the effort for reaching better performances in processing events, comparable to the gains such as the RETE algorithm [38].

The motivation for this study included a fast processing capability for real-time constraints. Initially, a non-event processing lower-level syntax language was developed by the collaborating team and it was realized that more capabilities were needed. An event-algebra needed to be implemented and some higher-level expressive style would be helpful. Such considerations paved the way for this work. Simpler mechanisms were devised compared to RETE but later, fortified with other data structures to eliminate run-time search needs. Mostly, predictability for worst-case time requirements during the processing of each incoming event was established. However, thanks to the properties of the lower-level scope of the FMS domain, simplicity was enforced that could accommodate the required complexity. Assumptions such as any incoming event could be processed and deleted without saving related state due to the low arrival frequencies and fast operating routines, could be easily made. Such assumptions proved correct during our verification tests. Processing times were measured that outperformed the existing approaches as far as we could access any published results.

1.4 Contributions listing

In this thesis work, various contribution has been achieved. Some are related with Fault Management and others related with Complex Event Processing.

- A Fault Management Reference Architecture was developed.
- Fault Prevention mechanism was added to Fault Management(FM).
- Rule based event processing languages were designed for various FM tiers.
- A Complex Event Processing (CEP) rule-based language and its processor was developed and tested for real time applications.

1.5 Organization of the Thesis

This doctoral study is organized as follows: Chapter 2 gives a detailed literature survey about Fault Management, Rule Based Systems and languages, Domain Specific Languages, and Checkpoint mechanism. Fault Management Reference Architecture which is developed for this doctoral study is given in Chapter 3. Chapter 4 defines methods, techniques and a roadmap for the architecture Rule Based Usage, and Checkpoint Approach based on this reference architecture. Designed and developed language and its processor is explained in detail in Chapter 5. Evaluation, measurements and comparison are given in Chapter 6. Finally Chapter 7 gives conclusion about the study.

CHAPTER 2

LITERATURE SURVEY

In this thesis study, literature survey was conducted on four different topics: Fault Management, Rule Based Systems and languages, Domain Specific Languages, and Checkpoint mechanisms. Although there is strong relationship among these three topics each should be explained separately.

Firstly we are going to explain Fault Management paradigm with its definitions, the techniques used, measurement and evaluation techniques and formulas, and methodological experience and recommendations. The publications are evaluated in terms of these classes.

2.1 Fault Management

Definitions

Generally, the basic definitions of fault management such as bug, crash and disastrous emerges in the form of a dependent series. For example, a crash occurs because of at least one fault. But not every fault can always cause a crash. These definitions can be found in [103], [54], and [55]. On the other hand, the more advanced definitions for the for top-level quality factors such as system reliability reveals concepts of elementary measurement. For example, Mean Time between Failure (MTBF) and Mean Time to Repair (MTTR) measures are used for availability.

$$Availability\ \ A = MTBF/(MTBF + MTTR)$$

There are other measures that can be used more during the development process:

$$\begin{aligned} Fault frequency\ \lambda &= 1/MTBR \\ Repair frequency\ \mu &= 1/MTBR \\ Reliability(dependontime)\ R(t) &= e^{-\lambda t} \end{aligned}$$

Techniques:

Fault management techniques used in the literature can be divided into two sub categories: Structural and operational. Structural techniques are usually is based on the principle of redundancy. This covers finding of both hardware and software units and managing which unit will be active and when. There are same techniques with different names in the literature which are managing activation of the units: hot boot, cold boot, static/dynamic redundancy. Depending on the coupling mode of the components there are models that compute the reliability of the system. For example; in case of crash of any component connected in series causes a system crash is in an example such computation. One of the structures which is a frequently modelled one calls "K-out-of-N" and is requires working amount of component K's out of N's. This structure is similar to N active and M redundant units which is called N+M redundancy. Operational technical examples are listed below:

- Heartbeat messages
- Checkpoint Rollback (state saving and restoring to a previous state)
- Data controls
- Watchdog timers
- Polling methods
- The determination of the active volume of the procedures (may require voting)
- Periodically reset (reboot without errors)
- Model-based fault diagnosis methods

In addition there are recommended techniques to develop more durable code for software units.

Measuruments:

Depending on the structure of the system, there are some formulas to compute criteria such as total availability time in terms of reliability of the components [103]. These formulas can be change according to binding of the components (serial/parallel/sequential) and methods of the redundancy. In addition, the frequency of errors, such as values that can be measured throughout the development process, and the costs associated with the testing process models have been developed to be compared with [54].

Methods:

Various studies suggest similar and sometimes different measures. However, a development methodology has not been found for generally cover the Fault Management. For example, information such as input classification, to process data before confirmation, error localization, discretization of the memory access, identification of the limit values and to be handled carefully, recording corrupt warning, checking memory, stack and load, modeling of crash types, and tracking and recording of data of fault detection should be stored to be consulted for the next studies. However there are no methods to handle these kind of information that are critical to Fault Management.

The book [103] can be used as a valuable reference for fault tolerance system. It contains mathematical explanations for lower-level mechanisms. Software and software development methodology issues remained weak besides hardware perspective. Moreover, at a higher level on the methodology and framework support that is necessary for fault management is very poor.

[95] is a book which is more software oriented. It also contains a more descriptive introduction. Robustness and diversity issues are given strongly. However it does not contain the general guidance on issues such as methodology or framework.

A fault management structure is recommended in [17]. A middle layer like CORBA [119] is designed provides transparent plug and play structure even if runtime. However some portions of it not yet fully developed. It mentions that combination of real time and reliability is difficult. The authors also presented a study that highlights the real-timeliness. Except fault management some distributed system problems was also mentioned by the authors e.g. Load balancing. Some original contributions:

- Periodically restarting of the software without errors (precaution),
- Stores only changed state information rather than all state
- Do not consider late message sent from sensor units working in parallel
- There should be redundant controller units which control each other
- They developed some mechanism to estimate some faults.

A comprehensive protocol with the necessity compliance with a given architecture is developed in [42]. They give a detailed interface design but some capabilities are missing to meet the requirements. The protocol is designed for especially needs for communications equipment manufacturers' needs.

In [88], on the lower level there are original ideas will be useful for fault management. The idea is setup based on aviation infrastructure. They worked on some mechanisms to diagnose fault without over repeated sensor by using model based computation over a result of the interpretation of the available information.

A design pattern approach for fault management is given in [14]. This publication is useful for designing a fault management system. The authors present special design pattern for fault management and safety. But abilities will be helpful in making decisions about the design information is incomplete. There are useful information about recognition of techniques and concepts for beginners in the subject.

[26] gives a specific study on the layered structure but this approach is not explained clearly. While they also present a graphical architecture modelling language, it is limited to the framework proposed by the authors. The language is not a generic modelling language.

[8] contains brief summaries about the structure of the satellite software. The thesis topic is weak in terms of methodology. It suggest that design should be done by using design pattern and frame sets.

A middle layer system was evaluated in [81]. Almost all necessary parameters to be used in a middle layer are presented for the user. If a middle layer is developed or uses, one can use these parameters.

[54] is a useful source for basic mathematical definitions. Particularly useful for testing the connections between required effort and reliability in a manner that may be contained. They provide a method summary suggesting that it should be started with a model contains crash modes.

[126] gives a high level discussion but does not provide more practical solution. Retry, user and customer consultation and advice to the authorities mentioned are the benefits of such approaches.

In [79], authors suggest that fault management and distributed systems will be supported with QNX software [50]. Moreover, they talked about parsing in memory and other structures and live update.

A very important suggestion was given by [72] that it should be added extra codes about fault detection into while the development of software. Mechanisms such as memory, stack, input data record, observations load level must be found within the code out as much as resources. Moreover they argue that these precautions put in use with polling method. In order to avoid losing the internal state information, some codes should be used to enable alternative fault detection modes before the crash.

A book about Safety-critical Computer System Storey [110] provides an information about all of life phase of a safety-critical system. These phases composed of idea formation and properties, its approval certificate, setup, and even its services. While it provides how to evaluate security forms will be used in the projects, at the same time it defines how long time it takes for a system to be developed meets these security requirements. The book also gives a valuable ground for appropriate techniques to examine security requirements of computer based systems and methods can be used to increase the reliability of these requirements. To highlight all these systems, Nuclear Plants, Aircraft, automotive and consumer products industries belong to the a wide range of industrial sectors including real-world examples and usage examples (Use Case) are given.

In [57], author emphasis difference between safety-critical and high availability systems. He claims that most important difference in both systems is: While a high availability systems maximizes the uptime (service time), does not care about human life. On the other hand in a safety-critical system, the most important criteria is human life. Hence if a fault causes a mortal result, the best solution is shut the system down. The author also examines relationship between fault, error, and failure. And he explains how to design a safety critical system step by step. First for these type of systems, hazard and risk analyses should be done by using Fault Tree Analysis (FTA) and Event Tree Analysis (ETA) methods. The author describes two methods in comparison and talks about hardware and software measures that can be taken systematically.

A high availability design principle on telecommunication infrastructure system is explained in [109]. The author claims that there must be a hot redundant (hot stand-by) application to fast and transparent a fail-over system. In order to provide this type of system, data on all disk and memory of the systems should be replicated synchronously. He also suggest a middleware for design with following functions: Disk data replication, memory state replication, fault audit, and transparent fail-over control.

A technique which is called HAZOP (Hazard and Operability Study) is developed in [35] used to design of a chemical fabric. HAZOP is the most effective first top-down approach using this type of design. Moreover authors give a case study for Fault Tree Analysis (FTA). They talk about methods on autonomous moving robots and SafeSAM project is used for risk analysis to these type of robots. They also suggest a method for safe situation. The widely used security analysis techniques such as FTA, Fault Modes and Effects Analysis (FMEA), HAZOP, and Event Tree Analysis are also studied. This article also talk about SAM (Safety Argument Manager) that is used to modelling and building for above methods and numerous techniques such as generic modelling notation, basic block and line diagrams, data flow diagram and state transition diagram and tables. Moreover it also provides a log mechanism to index for safe situations. The article mentions RATIFI project that is used classify risk analysis techniques and explains assessment of situations with example. Briefly this article is a guide for best practices, tools and languages for critical systems, test and simulation environments, case studies, cost-effective formal methods (such as

RAISE, SADLI, DATUM, MORSE) used in obtaining highly reliable systems. While it provides a state of art for critical systems also it presents four new approaches:

1. Knowledge based systems
2. Functional programming
3. Neural networks
4. Natural Language Programming

[96] explains an HA (High Availability) supported operating system designed for embedded systems and its market requirements and benefits. The operating systems supports 5-nines paradigm as well as:

1. It has a framework providing fault recognition and detection, fault isolation, and recovery
2. Self monitoring
3. High manageable
4. Transparency for service transportation
5. Hot-swap for hardware and software
6. Reliability: Application isolation, normal and super user protection, overrun protection, resource reclamation.

A new term is HA STACK is introduced in [100] for high availability. HA Stack composed of hardware and software layers. Hot Swap, Hardware Redundancy, and System management is in the hardware layer. While hot-swap enables changing the faulting components without shut the running system down, hardware redundancy protects to system from single-point-of failure. In these layers, a star type network topology is used for reliability. The system management layer responsible for monitoring all HA stack layers and collects data (health status, temperature, voltage, etc.) that may be causing the fault from in all layers of the system. It has an intelligent GUI for this. This GUI also can restart the parts on the platform separately. On the other hand, software layer consists of Fault Tolerance, Redundant Host, and Predictive Analysis and Policy-based Management layers. The first layer fault tolerance has fault tolerance components should continuously synchronize very important data. That is in order to provide very good fail over the latest the most reliable data must be synchronized between two redundant components. The fault management layer is also responsible for load balancing. Predictive Analysis and Policy-based Management layer continuously monitors and controls to entire system in real time. It establishes a set of rules and it solves the occurrence of an event or set of events with the help of this set of rules. This information is also used to defining the system capacity.

[41] explains the properties of High Availability (HA) which is developed by Service Availability (SA) forum. HA developed two interface properties:

1. Hardware Platform Interface – HPI
2. Application Interface Specification – AIS

HPI is explained two different groups: Physical overview and Administrative overview. While different users can have the same physical view, they have different administrative overview. Physical overview uses an element descriptor to define each physical part in the system. That is each element in the system has a number or name. All these physical parts have administrative capacity. HPI interface has some monitoring and controlling properties to manage these physical parts. These features in the design highlights are explained in detail. HPI also defines a set of library functions written in C language. This library functions is particularly applicable to each hardware platform. Thus, the HP model and functionality can matched on other special hardware platforms. On the other hand, AIS reflects used application software and the middleware that is bottom of the abstract model. AIS monitors and controls application and middleware with the help of standard functions that it has. AIS provides five different APIs for Availability Management Framework:

1. Cluster MemberShip
2. Checkpoint

3. Event

4. Message

5. Lock

One of the most know fault management techniques "redundancy" have been discussed in [56]. The authors talks about Hardware Redundancy, Software Redundancy, Time Redundancy, and Information Redundancy. For example, automatic control logic circuits and multiple flight computers using in a flight can be example for hardware redundancy. Whereas; multiple algorithms with the same result can be used for a specific purpose is an example for software redundancy. Time redundancy can be done with communication re-transmissions. Finally, information redundancy can be done with backup, checksum, and error correction codes. In order to get rid of software errors N-version programming can be used. However using this technique bring some disadvantages. As another method dynamic redundancy can be applied on transaction (from the beginning to the end of the application process, which is consistent collection of processes). Thus, in case of any error the system will be able to return to the state that is consistent. This method also known as checkpointing is about snapshots of the previous transactions of software status. However this snapshot can be taken if and only if by a previous transaction can be completed without any error. The disadvantage of this method is there may be an error when taking snapshot in checkpoint. In order to prevent this disadvantage the author introduces a new method which is called Checkpoint-Rollback. In this method, while data is send to two different and continuously working bulk-storage drivers (a kind of replica server), transaction number is also stored. In case of any error, either original server or its replica restarts from where it's stopped previously with stored transaction number. The disadvantage of this method is very time consuming to recover system from errors. A process-pairs method is developed to prevent this disadvantage. This method activates both continuously working persistent mass-storage device and directly working hot-backup server. Thus, primary server sends the transaction completed successfully and the new consistent state to the hot-backup server. Both primary server and hot-backup server save these data to their mass-storage devices. One of the most important features of the system that sustains is the heartbeat messages. Primary server continuously send this message to hot-backup server. If hot-backup server cannot receive this message very quickly replaces the primary server. Process-pairs method can attain 5-nines or higher availability levels without user intervention.

In [127] the authors focus on a central complete fault detection rather than traditional recovery techniques which are located different level separately. They called this method is CAFE (Crast-At-First-Error). CARE is a system recovery process which is designed based on core routers working on Internet backbone and it can also be used for other redundancy designs. The article focuses on dual backup system. Another purpose of the system is decrease the fault detection time in the parallel redundant backup systems. They define three different modules which are used in recovery system:

1. Health Monitor
2. Diagnosis Unit
3. Swap Management

Health monitor is a protocol provides coordination to detect fault between active node and standby node by using heartbeat technique. Thus health monitor can sense whether active node closed or not. However, when the heartbeat detecting the active node is closed, it cannot recognize suddenly happened malicious behavior. To avoid this, some of the ideas put forward are: Message queue, buffer pools, task scheduling etc. All these methods store real-time behaviors of the surveillance objects. Some very well-defined patterns (well-defined patterns) may be helpful in developing some definitions are not normal. Diagnosis unit is then required for some of the errors detected by the health monitor. In fact, this unit controls the results of monitoring, examines faults in the modules and finally detects recovery actions and defines the order of these actions by pretending as a doctor. Swap Management process is performed automatically and is done on standby node.

The publication [113] investigates requirements of fault management server can be used in high available communication systems. Authors bring a new approach to philosophy of "who will inspect the controller" because fault management server can also give fault and this fault can trigger other faults. The study is showing that errors that can occur in fault management server will not affect the whole system. So, for a system to be available %99.999, there is no need for fault management system to be also available %99.999. The article explains design phases of a cost-effective fault management server. Fail – safe ratio (fail-safe ratio: an error that may occur in the fault management server, possibility of closing the entire system) and fault coverage ratio (fault coverage ratio: possibility of detection and recovery of the errors within the system by fault management server) of the fault management model using in the system are explained and their specifications are mentioned in this publication. The authors present a Markov Chain model which developed by himself and detailed mathematical modellings which used for system design. In there, their explanations besides numerical results (down-time, cost, 5-nines) could be seen.

A usable model is designed for high available platforms by using 3-level hierarchical decomposition which composed of Markov Chain and reliability block diagrams is explained in [117]. This design model was built on SHARPE software package and all evaluations were done on this model. In order to define effect of critical parameters sensitivity analysis was performed. The authors claim that Monolithic Markov Chain model remains inadequate a detailed fault/recovery behavior is considered for an entire system. Besides reliable block diagram or fault tree analysis when considering large state space model is always can be used instead of Markov Chain model. But reliable block diagrams can be solved very easily, error types recovery repair scenarios is quite limited. So in this model, the top level block diagram and detailed error/recovery scenarios at lower levels with the Markov chain model approach combined with a multi-level hierarchy has been revealed. Detailed hardware architecture and all components used, Carrier Grade HA software architecture and services, highly available features in a system is given in detail. In addition, the system used in the model are provided with all the sub-system modules.

In order to solve the existing restrictions an additional library which is called SNE (Stateful Network Equipment) is added to HA protocols in [80]. This library works under the Linux and developed for firewalls. The previous firewall was working on package information and they have a static structure. Therefore, they were often exposed to DoS (Denial of Service) attacks and were passed. While the proposed architecture is defined in detailed, very problematic scenarios supported by the SNE library is described and experiments on these scenarios and assessments are made. The SNE library provides easily design high available network materials that can be adapted real operational scenarios for network designer. The previously developed two replicas of the critical elements of the network are discussed in the publication. Then the design and architecture of SNE library elaborated. A net-filter framework is developed. Defined events (with a new uplink connection and changing a critical information) and realistic scenarios (primary server failure and backup server to take his place, the backup server error and restarted, the backup server to remain dead, the old primary server again to stand up, the backup server to the primary server communication is lost) were performed and tested by designed SNE and netlink.

Replica-control protocol, recovery strategies, and availability in distributed systems are studied in [116]. The author explains 3 replication strategies:

1. Location-based paradigm
2. Multiple replica classes
3. To achieve high availability as effective

These replica approaches are explained in detail. This publication is revealed two important purposes:

1. Eliminate replication cost: When defining fault and recovering system from fault, they developed protocols that remove the cost between corporations of replicas during the process working. Also those developed protocols did not result in any bottleneck.
2. Improving replica performance: The working time is reduced by allowing clients to reach to closed replicas. With the help of this, a recovery which is independent from fault is also provided.

2.2 Rule Based System

In this section, the literature surveys of two different subjects are discussed. Rule based systems are considered among the main components of adaptable reference architecture and a survey on this topic is conducted. Also, the tools that are commonly used in this domain are presented. In this study, rule based approach are used to support fault management which can be customized with a software pipeline for different applications. Basically, a different fault management application is formed by introducing new rules to the system. There are different rules for different layers of Fault Management Pipeline Architecture. Rules are needed to be formed by predefined languages in order to make rule usage manageable. There are also sub-classifications of these languages for different layers of the reference architecture.

Domain Specific Languages (DSL) is the second topic that is surveyed in this section. Based on the fact that, the language that is utilized for rule definition has a quite restricted domain and is predefined, it should be considered as a DSL.

Rule based approaches are among the techniques that are used heavily in the developing phase of expert systems. Every rule are consisted of two main parts: left part and right part. A representative model of a rule is given below:

Left part => right part

Rules are formed by combination of two statements, "IF" and "THEN" [48]. In this programming approach, the rules represent discovery or invention that is defined as necessary action to take for the given situation. These rules are depended completely on human experience. "If" part of the rule is the expression that defines the event or the data that makes this rule applicable. "If" part reveals the conditions that makes the "then" part valid. Association between events and patterns are called statement association. Expert system tools involve an inference mechanism. This mechanism automatically associates events and patterns, and determines the rules that are applicable according to this. The "then" part of the rule is the part that defines the actions when the rule is applicable. These processes of applicable rules are executed when inference mechanism starts. First inference mechanism picks a rule and then the processes of the selected rule is executed. After that, another rule is selected and its processes are executed. This process continues until no applicable rule remains [108].

2.3 Improvements: RETE and Event Handling

Some research has been conducted to adapt RETE to event processing [120],[12], [130], [90], [102] and also others to achieve similar gains through other means [128], [70], [74], [66], The real time concerns definitely deter the usage of blackboard mechanisms due to the non-deterministic inference times. RETE, a great help toward this goal has not yet supported event processing to the level that is accomplished for rules that exclude events.

Currently Available Tools

RETE algorithm is improved in many ways. Actually, RETE is designed for permanent and changing facts and therefore is not suitable for event processing. Hence, the most important improvement is done by adding event processing and temporal event capability to the RETE. Another improvement is garbage collection enters the picture with event capabilities. RETE is extended with by using temporal operators and garbage collection in many publications. Temporal operators differ in every article. Durable, time point, before, during are some of them. Another term that is widely used is Complex Event Processing (CEP). It is defined as a technology in order to monitor and detect complex events which are composed of causal, temporal or logical relationships in information systems.

Many approaches have been developed to detect complex or simple event in the literature. Small number of approaches is distinguished from others by the mechanism which is used. For example in [45] is a language used for defining both the simple or composite events and temporal events operators by using Finite Automata (FA) as event expressions. On the other hand, in [44] to define and detect complex events in active databases Petri Net modelling language which is called SAMOS was used. The authors define composite or complex event as how many events of a specific event pattern have occurred during a predefined time interval. However, many approaches use tree data structure to represent and detect complex (composite) event structure. The main advantage of using tree for representing complex event is simplicity, functionality, well defined structures and traversal methods. The well-known tree structure for representing coming events are GEM event language [75] and Ready event notification service [47]. GEM is an interpreter language using for event monitoring and also detection of complex event and converting these primitive events. Any type of temporal constraints can be defined in GEM. However event monitoring in GEM is only used for notifying event. Event Processing Service (EPS) [78] is also tree based service using for detection of complex event which consists of simple event and some operators. These operators can be Boolean, conjunction and disjunction, sequence, and repetition. Composite events are commonly represented as trees where the leaf nodes represent primitive event types and the intermediate nodes represent any of the supported event operators. EPS has some operators like "&", "||", and ";" which are called composite event type and has also primitive event types have attributes that their values representing state changes in the system. It has also a garbage collection based on event ageing. EPS uses a shared subscription tree to process coming events. The major advantage of this optimization is that it reduces significantly the cost of scanning the leaf nodes and evaluating sub-events that are common to many event trees. This property eliminates some limitations of GEM language. Another tree based event detector is EVE [46]. It supports composite event detection, management, and registration. Eve is based on Even-Condition-Action (ECA) rules. It defines several composite event types: sequence, OR, AND, repetition, negation, and concurrently. Snoop [22] is also a tree-based event specification language for active databases. It supports temporal, periodic, aperiodic, explicit, and composite events and is composed of a set of primitive events and event operators. It divides a primitive event into six sub events: Database events, temporal events, explicit events, absolute events, relative events, and composite events. Snoop defines five different operators: Or, Sequence, All (combining of sequence and Or), Aperiodic (A, A*), and Periodic. Event detection is done based on event types: Primitive event detection and composite event detection.

In [74] temporal events such as before, after, and during were added to the RETE. These operators have gained the ability to the Rete for comparing the events with temporal facts in the working memory. They used OPS5 [37] as a host system to implement a temporal production system. They claim that each event has a temporal duration so that it should be defined between time intervals. Although they define only three temporal events, their knowledge maintenance routines in the TPS reproduce undefined relationship from these events such as finished-by, overlapped-by, and met-by. However they did not clearly define how to evaluate operators.

[130] argues that RETE does not support time sensitive patterns. Therefore, the authors in order to eliminate these constraints have developed an approach to handle the time-stamped event and temporal constraints between events. The approach helps to writing rules that can process both events and facts. Also to solve the null-join problem (If any of the join links between alpha and beta memories is empty, null activation occurs and this reduces inference efficiency) they developed a method which called "unjoin". The temporal approaches in this article only defining time spot events. However the authors have not developed an algorithm for ongoing events. They used after and before operators define the temporal constraint between events and combination of these operators. They compared their algorithm for RETE. According to authors, their algorithm in some cases performed better than the RETE. [12] also mentions that RETE does not consider time-related event so that it cannot detect temporal events patterns. The author extends RETE with a new algorithm to detect time-stamped events and time-independent facts. Also he developed rule mechanism to figure out an event expiration date. But he did not give a detailed explanation about the algorithm.

RETE is extended with 13 temporal relations in [120]. The authors adapted these 13 temporal relations (X before Y – X equal Y – X meets Y – X overlaps Y – X during Y – X starts Y – X finishes Y – X contains Y – X overlapped by Y – X met by –Y – X started by Y – X finished by Y) from [70] to define complex events. Quantity can be given to operators. For example, how much two intervals overlap initially and at the end can be expressed. Also a CEP (complex event processing) system which is composed of Rule Engine, CEP configuration tool, scheduler, queue, agenda and rules is developed. Moreover a static publish/subscribe system is designed to complex events. It is responsible for deliver complex event to its subscribers. They also designed an incremental garbage collection mechanism to reduce memory usage by removing the events that cannot meet their temporal constraints. They implemented their ideas on Jboss Drools [94].

[102] is a short survey summarizing the ending and the ongoing research on RETE

algorithm. The article also introduces an algorithm which called SnoopIB is a customized event detection algorithm based on Rete. The authors define correlation logic and actions as Event Action (EA). In order to define complex relation rules they use EA languages. While doing this, they takes advantages of logical, temporal, content-based and other operators. The authors combined two rule execution paradigms, event-driven and data-driven and added CEP approach to the RETE. Rather than combining event processing with RETE, they process events in the different graph which is called event detection graph and combine these two graphs in the last node. The event detection graph is a directed a-cycle graph. They located all simple events at the end of the graph. They create a simple event node for a simple event. On the other hand, the complex events are represented at top of the graph. While this graph is responsible for detection of event patterns and definition of properties of events, RETE graph is responsible for rule conditions and matching the satisfy rule conditions to facts.

[12] also introduced a new algorithm to extend RETE with time-stamped event management. The extended algorithm support detection of time-related event patterns and temporal limitations between these events. With the addition of event management to the rules engine, garbage collection enter the picture. An automatic garbage collector added into the engine can detect expired event in the RETE network and deleting it from the working memory. The author claims that maintaining, writing, understanding, and debugging a rule-based programs is easier with his algorithm that has been implemented in the ILOG JRules [16].

Basic specification of composite events is defined based on lambda-calculus like GEM language in [24]. But this approach does not support temporal constraints. A complex event language which is called Core Composite Event (CCE) is developed in [90] to express event patterns that occur simultaneously. In order to do this a composite event detection framework is built onto an existing publish/subscribe middle layer platform. CCE is compiled on automata and provide pattern in the type of regular expression to detect distributed event. The language uses operators like concatenation, alternation, iteration and temporal control, parallelization, and weak/strong event sequences. CCE provides mobile event detectors can detect distributed event which is close to event sources. Another composite event definition is done in [118] by using composite event filter expressions in CCE language [90].

[70] focuses on improvement of rule engine of RETE algorithm. The improvement were done on three different methods: decomposition of rules, hashing of alpha-nodes and indexing of bet-nodes. Firstly, they decompose the rules based on "and" operator. They decompose the complex rules into simple rules by using distributivity law of Boolean algebra. In the alpha-node hashing methods, in case of any alpha-node adding operation to the type node, they add a HashMap which its key is changeless name and value is alpha-node at same time. They claim that this method improvement while decreasing the unnecessary checking operation, increasing the efficiency. The last method is beta-node indexing. This method increases the placement time of newly coming data to the beta-node. They claim that with the help of beta-node indexing method, for elements that are already in the beta-nodes does not need to traverse again so that this decreasing the saving time and increasing the efficiency.

RETE is a forward-chaining algorithm but a backward-chaining algorithm is added to RETE by using OPS5 like syntax in [66]. The backward-chaining algorithm works with RETE's forward chaining algorithm. This interleave working provides switching the most suitable control strategies at rule level. The authors also developed a new class which is called hypothesis is responsible for finding and combining goals that can be accessed together.

2.4 Complex Event Processing

An event can be a symptom for a network security like an IP address 192.168.1.124 trying to reach application server. This type of events can be called security events. Another type of events is used in financial market like reporting of a specific paper price reaching some predefined value, for deciding to sell it. In a supply chain system concerning a supermarket, an event can be an information to be generated for the operator, if the level of a product decreases 35% in last five hours. This event will be used for adding necessary amount to the stock. Briefly, an event can be an output/message from a sensor or RFID, a symptom from CPU, an email confirmation from a company, a financial finding, a state change in FMS or databases, an airline state like landing or take-off etc. [73]. A complex event can be defined as a sequence of events with temporal or spatial constraints. For example an event X is followed by Y and then Z with a temporal constraint such as "in 10 seconds". In another words an A event is followed by B then C within 10 seconds. Moreover, a complex event can have a spatial constraint like distance between events.

Complex Event Processing (CEP) is used to detect complex events. It can be conducted by a language or a technology. There are lots of event processing applications that are used widely in industry. Some of them are: Fault management in telecommunications, database management systems, and RFID systems; fraud detection of Internet banking, casino, and on-line shopping. It can also be used for patient monitoring in health sectors. The CEP methods are used in such systems mostly based on traditional data structures. For example; a tree-based structure is used to define and detect complex events and processing them used in [69]. On the other hand a directed graph structure as used in [122] to process complex events coming from RFID systems. Most of the CEP publications use NFA (Non-deterministic Finite Automata) to detect complex events [123], [101], [53]. Some approaches use ontology to define and detect complex events [32], [51]. A kind of Petri-Net model which is called Timed Petri-Nets is used in [129] to process and detect complex events in RFID systems. A similar method which is called Timed Net Condition Event System (TNCES), also based on Petri-Nets, is developed to model complex event processing in [1].

In [123], a query-based method which uses a tree structure is developed to process hierarchical complex events coming from distributed sensors. The authors mostly focus on probabilistic complex event processing. They use Non-Deterministic Finite Automata (NFA) and Active Instance Stack (AIS) to detect sequential events. In order to provide hierarchical complex event detection, they locate a primitive event at the leaf node and operators at the non-leaf nodes in the tree. The method detects events considering the probability of them.

[101] introduces a new method which is called INDCEP to process complex events in sensor networks that can meet the needs of real-time distributed systems. INDCEP supports disjunction, conjunction, sequence and negation operators. In order to detect complex events INDCEP uses an NFA-based CEP. The authors tested their systems

with fire detection system and contamination level monitoring system. However they do not give any information about any test results.

In [13] a complex event processing model and its engine is designed and developed based on relational algebra. The system works on ESB (Enterprise service Bus). The authors designed a static publish/subscribe system which is managed by an event engine for events. Whenever an event occurs, it is stored in an event queue and later is processed by an event processor. After the selection and combining operations which are related to the event, the result is sent to the event subscriber over event scheduling. The authors compared their system with the existing ESB systems. They claim that CEP system which is based on relational algebra performs much better than existing ESB systems. Similar solution for CEP is given also in [65].

[6] provides a backward chaining approach for logic-based and data-driven complex event detection approach. The authors use Concurrent Transaction Logic (CTR) [15] to define complex events. With the help of this method, they encode events patterns into the logic programs. They also provide a transformation method to convert a user defined complex event pattern to the event-driven backward chaining rules. Actually the authors use Prolog type rules which are suitable for backward chaining. The developed approach has been tested with a rule structure which is composed of six atomic events and four complex events. They called atomic events like e1, e2, e3, ... and complex events like ce1, ce2, ce3. A complex event can be composed of atomic events or a combination of atomic events and complex events. According to test results the designed method can process between 12600 and 13500 events in a second. Moreover it also process 10000 events in 4438 milliseconds.

An event-driven complex event processing approach is introduced in [32]. It can drive Jess and Esper rule languages. Events, constraints and relationship between them which is gathered from sensors are defined by using OWL type ontology language. A generic event hierarchy structure is defined for sensor networks. The authors use temporal operator like after, before etc. Esper engine and EQL query language which belongs to this engine are used for rules.

In [53], the authors claim that they developed the fastest complex event processing system which can process an enormous amount of events on FPGA. Instead of traditional SQL-based approaches, they developed a C-based event language which supports regular expressions on basic functions. In order to define each expression on regular expressions, they use a function which is written in C-language. The developed approach is tested on real-time stock systems.

[51] introduced complex event processing engine which is called ORFID-CEP to use tracking the agricultural products. The temporal relationship between events can be defined by using ORFID-CEP system. The authors introduce two types of events: Scheduling events and non-scheduling events. While scheduling events can be defined with temporal operators such as a sequence (e1,e2), within, temporal sequence (e1,e2,t1,t2), and sequence+(;4) meaning that two or more events can occur in order. Non-scheduling events can be defines with non-temporal events like OR, AND, NOT, COUNT (an event occurs n times). The developed engine is compared with Esper complex event processing engine.

A fuzzy ontology based context aware complex event processing is introduced in [124] in order to support uncertainty in RFID queries.

2.5 Expert Systems

In practice, expert systems are used widely in many domains such as fault detection and repair, tracking, analysis, translation, consultation, planning, design, forwarding, clarification, training and definition. There is also a fuzzy expert system study on fault management [68]. Definition of expert systems is given by Feigenbaum et al. as below [19]

"Expert system, is an intelligent software that can solve problems which require expert knowledge and skill, by using information and logical inference. The information and logical inference mechanism used by this software are required to model expert's knowledge and logical inference."

Expert systems are developed as an production system and generally consists of three parts:

1. Production groups that are held in production memory.

2. Assertions that are stored in working memory- blackboard/data memory/current state/knowledge base
3. Inference engine.

There are two types of production systems:

- Forward Chaining (Data-Driven)
- Backward Chaining (Goal-Directed)

Data-Driven production system [71] determines which production to be executed using the information in knowledge base. Production is a state-effect structure: State corresponds to left part and effect corresponds to right part. This way, with the help of knowledge base, processes can infer their new states by using their current state. Briefly, when a rule set is applied, a new situation is occurred. At the end, either valid or invalid results are obtained.

In goal-driven production systems, first it is assumed that our aim is correct and system tries to reach the data using a backward method. If all the information obtained by this procedure exist in the system, the correctness of the goal is approved.

Another component of the system is Working Memory. Working Memory is a collection of data objects that shows the current state of the system. Memory of a production system is like a blackboard of a class [71]. Consequently, expert systems are capable of inclusion of new information and consistently developing systems in this respect [48].

A production system interpreter runs the following processes in a loop successively:

- **Matching:** For every rule, left part of the rule compared to the working memory. Every subset of the element of the working memory that confirms the left part of the rule is called an exemplification. Each exemplification is numbered to shape the conflict set.
- **Selection:** A subset of the exemplification in the conflict set is chosen according to some criteria. In practice, a single most recent exemplification is chosen.

Table 2.1: Comparison of shells and programming languages

Properties	Shells	Programming Languages
Speed and Ease of Development	High Level	Low Level
Knowledge base structure and reasoning	Limited	If necessary, development
Knowledge base management	Very easy	Hard
Interface	Available	Must develop
Efficiency and Performance	Slow	Fast
Explanation	Limited with tool	Must developed

- **Action:** Right parts of the rules that show chosen exemplifications are executed.

There are two different tools for development of expert systems.

1. **Programming Languages:** An expert system can be developed using General Purpose Programming Languages such as LISP and Prolog that can easily process symbolic data.
2. **Shell:** A shell involves an inference engine that helps developer to create her own knowledge-base and an editor or a graphical user interface.

A comparison of shells and programming languages is given in Table 2.1.

Expert systems are not recently emerged. Commonly known first successful examples are DENDRAL [18], MYCIN [104] and PROSPECTOR [31]. DENDRAL is and application used for solution of organic chemistry problems, MYCIN for medical diagnosis, and PROSPECTOR for mining exploration problems. Normally, expert systems are systems that would requires expertise and makes decisions that require human factor. They can be used at different complexity levels. An expert system should be developed for a narrow and well-defined domain. It is very hard to develop a system that can provide expertise in several domains [5].

Some common expert systems tools: CLIPS (C Language Integrated Production System) is an expert system tool that is implemented by NASA to design and developed object oriented expert systems. It has several important properties such as: Portability, diversity and integrity of the knowledge representation, to be expanded, verification and validation [98]. There are three different knowledge representations in CLIPS:

- Rule based
- Object oriented
- Procedural

Since CLIPS was written in C programming language, portability has been acquired without sacrificing speed. It can work many operating systems and computer architecture without requiring code changes. It has some mechanisms providing the accuracy and approval of written rules semantically. Moreover, these accuracy and approval mechanisms have ability to control arguments of functions.

PPS (Parallel Production System) is a data-driven parallel production system independent of the domain. The system give ability to users to create modular expert system can work in single or multi-processor architectures. The modules are defined as expert objects which are called communication information resources [71].

PPS has two powerful properties:

1. The ability to hide the actual physical architecture of the computer from users. Briefly an application which is written by using PPS can be automatically adapted to the physical architecture of the computer by PPS system administrator. Naturally it has two parallelisms: Exert Object and Rule Set Level.
2. User can easily create any type blackboard structure with help of power of PPS semantic. User can draw a graph to show how expert objects communicate with each other and the can easily define its structure with PPS. Some noticeable applications that are developed with PPS are: Circuit Simulation, Silicon Compilers, Expert Systems, Systolic Arrays, and Switching Networks [71].

OPS5 (Official Production System, Version 5) is a production language that is commonly used in expert systems [37]. It has common three properties that are already in standard rule based systems:

Table 2.2: Differences between OPS5 and Poplog

Poplog Terminology	OPS5 Terminology
Database	Working memory
Fact	Working Memory Element
Rulebase	Production Memory
Rule	Production
Conditions	LHS
Actions	RHS
Non-Repeating	Refractoriness

- A set of rules
- The facts that are working with these rules
- Inference engine determining the correct application of these rules

The biggest difference of the production languages such as OPS5 from other programming languages is that they do not have a code that is run sequentially. The rules are fired according to the conditions that are covered by the working memory. OPS5 has two types of data type: Simple and Composite data types. While simple data types are integers, floating point numbers, and strings, composite data types are data that are defined by the users. The data are stored in the working memory in the OPS5. On the other hand, “If-then” rules are stored in the production memory. The rules are completely independent in OPS5 and can take place in the production memory in any order. Rules and facts are used for knowledge representation in OPS5. Since it is written by using Lips, OPS5 is a fast and efficient rule based programming. Terminology of OPS5 is slightly different from Poplog terminology which is a reflective, incrementally compiled software development environment for the programming languages. These differences are reflected in Table 2.2.

OPS-2000 is a commercial and very popular rule based language used in 1980s [86]. Now it is free and open source, and can do forward and backward changing. It is a knowledge reasoning system based on information and supports following elements: Rules, a set of rules, expert objects, multitasking, parallelism, communication between objects, forward and backward changing, control library for inference engine, and there type patterns; class, free-form and relation.

RETE algorithm is a fast and efficient pattern matching algorithm that is mostly used in rule based systems [38]. RETE has form the basis many expert systems such as CLIPS, Drools, BizTalk, Rules Engine, and Soar. The main purpose of RETE is to increase the speed in forward chaining rule systems. In order to do this it limits the effort which is necessary for recalculation of the conflict set after firing a rule. While this advantage for RETE, require an excessive amount of memory is also a disadvantage for it.

A traditional expert system compare every rule to facts in the knowledge base, if it is necessary operating the rule and switch to the next rule. When this operation is finish repeat cycle and return to the first rule. The problem here is that in case of thousands of rules this cycle takes a lot of time and reducing efficiency of the system even if in the moderate number of rules.

RETE create a net which is composed of nodes. Every node except root node corresponds to pattern on the left side of a rule in the net. Hence the path from root node to the bottom node (leaf node) defines a left side rule. Every node has a facts memory that corresponds this pattern. In fact, this structure is generalized text tree which is called Trie or Radix tree [27]. Briefly a text can be coded on the tree and only one deterministic path that gives this text can be traced. The main advantage of Trie is requiring process as much as the text size when searching a text in the tree as opposed to binary search trees. Trie also use the memory efficiently since the deepest point on the tree is the longest text on tree.

In an expert system, when new facts are added or facts are changed, these are propagated throughout the network. If the fact is matched with a pattern, the node that causes it interpreted and an explanation is added to the node (annotation). When a fact or combination of facts corresponds to all patterns of a rule, leaf node is reached and the rule which is related to this node is fired.

RETE has the following characteristics:

- In every change in the working memory there is no need to re-evaluate to all facts from start to finish. Instead of this only facts which are subject to changes in working memory are evaluated.

- When the facts withdrew from the working memory, it allows effectively removal of memory elements.

  The algorithm continuously keeps the information associated with the nodes. When a fact is changed, the old fact is deleted and new one is replaced with it. The production rules can be acquired and defined by analysts and developers.

Another matching algorithm for artificial intelligence systems is Treat [77]. It developed a new storage method which is called conflict set support using in production system interpreter. The performance of Treat was compared with RETE and it was proved that Treat showed %50 better performance.

In [82], a programming language that is called XC is developed over C++. They also create a hybrid expert system tool which is called ET. ET combines the production rules, semantic network, and procedural codes written in Lisp languages. In ET, a little modified and faster version of RETE algorithm which is called Simple-Minded Production System algorithm is utilized. The minor changes which are done in the algorithm do not include about the partial matching that is stored. Authors claim that RETE causes some problems in embedded real time control applications. According to authors, the reason for the speed of RETE algorithm is very few elements is changed in the working memory for each cycle. Therefore the algorithm does not iterate on working memory or a set of production rules. In order to do this, RETE uses a discrimination network which is acquired compilation of left side of rules. The network indexes the rules with elements which are in the working memory. While some matched left sides are stored in discrimination network, fully matched are stored in conflict set. RETE algorithm operation speed is preferred to both static and dynamic memory. The amount of dynamic memory used at runtime is varies too much. Therefore it is difficult to predict the amount of memory that will be necessary. This property is not preferable for an application that will be used in real time systems. They also argue that the change will be done in the working memory is very costly.

RETE algorithm is extended to support temporal operators in [121]. The authors introduced a new concept which is called Complex Event Processing (CEP). They claim that CEP is an important technology for event-driven systems such as supply chain management for RFID, system monitoring, market analysis to news services.

The main purpose of this is not only to define single occurring events which are not connected to other events but also to define event patterns having logical, temporal or causal relationships. These types of events are called composite events by the authors. They also developed a matching algorithm which is used to detection of interval based events with sliding windows.

A rule based expert system is developed by using en Event Control Action Domain Specific Language (ECA-DL) in [106]. The system is composed of monitoring, detection, diagnosis and recovery components using detection and recovery of network faults.

Jess is rule based engine and scripting language based on Java. Despite it is built on RETE, it support both forward and backward chaining. Jess also added a query algorithm into the working memory. The performance of Jess is depend on the number of the partial matches generated by rules rather than number of rules. It has a XML rule language which is called JessML. Jess can be used both a rule based language and general purpose language [43].

A lazy approach to evaluation for production systems which is called LEAPS is developed in [11]. In fact, it is used as a compiler for OPS5's rule sets. LEAPS puts all coming evaluated facts to the main stack in the order of working memory. All facts are examined one to one and tries to find a matching rule. Whenever match is found the system remember the current iteration position and it stops the iteration to continue from where it left off then fire the matching set of rules. After completion of the operation of the rules, system is to try to continue either examining a fact at the top of process stack or from where previously stopped.

Drools uses forward chaining system and developed based on RETE by using JAVA. It is an object oriented rule based system. It has an inference engine containing forward chaining rule engine which is called JBoss Rules [94]. Drools provides several powerful properties such as: Engine (full RETE implementation, dynamic rule bases, asynchronous operations, rule agent etc.), Propositional Logic (inline evaluation, lateral and variable restrictions, First Order Logic quantifiers (or, and, exist, not, from, forall, collect, accumulate etc.), Execution control, temporal rules, event model, and etc.

Another usage of rules based systems are Identification Tree (IT). ITs are used to build rule based systems. With IT, the usage domain of rule based systems can be extended much more [43].

Some Expert System Shells: Some of the shells given below are free and some of them are considered cheap expert system shells [58].

KAPPA-PC works under Windows operation system [59]. It differs from the other expert systems by having elements that defines the system besides the rules. It supports backward and forwarding chaining. It has developed by using KAL programming language that is similar to C. In KAPPA-PC system components are defined hierarchically as class or instance. In this hierarchical structure, the features and information that are defined for upper classes pass through classes and instances that are derived from these classes. The duty of expert system elements that are defined as class and instance are taught as methods to the expert system. KAPPA-PC has the ability to run all the programs that involves methods in "EXE" format. It presents a graphical environment for rules, objects and code debug.

BABYLON: It is a development environment for expert systems. It involves skeleton structures, restrictions, Prolog type logic interpreter and definition language for diagnosis applications. Babylon is developed in Common Lisp and adopted to many hardware platforms [28].

Mobal is a system that is used for application domains that uses first order logic notation and development of operational models. It involves surveillance/monitoring and manual information extraction, an extraction motor, machine learning methods for automatic information extraction, and an information browsing tool and it unifies all these components. It is possible to develop a model for a specific domain by using Mobal's information extraction environment with an incremental approach in the sense of logical phenomena and rules. It is also possible to review the information entered through textual or graphical interface and edit it or insert new information [107].

ES is an expert system that supports fuzzy set relations and descriptions and utilizes backward and forward chaining systems. It can work on standalone in personnel computers [112].

WindExS (Windows Expert System): It is an Windows based, fully functional forward chaining expert system. Since it has a modular architecture, its capability can be increased by adding new modules. WindExS has a natural language processor, extraction motor, file manager, user interface, message manager and knowledge-base module. It offers graphical knowledge-base support.

MIKE(Micro Interpreter for Knowledge Engineering) is an educational software environment that is developed for English Open University, that is portable, free and fully functional. It involves forward and backward chaining systems, user-definable conflict resolution strategies, a frame representation language and inheritance strategies that can be adjusted by the user. Fine and coarse grained rule monitoring and rule execution history are supported in a new "rule graphic" screen. MIKE's version 2.03 has RETE algorithm. In this way, it provides a fast chaining system, a real maintenance system and uncertainty management [33].

RT-Expert: It is a shareware expert system that allows C programmers to integrate their own expert system rules. RT-Expert involves a compiler that can compile rules to C codes and rule engine library.

CLIPS based Expert System Shells:

DYNACLIPS (DYNAamic CLIPS Utilities): For versions 5.1 and 6.0, CLIPS consists of a blackboard set, dynamic information extraction and agent tools. CLIPS 5.1 and 6.0 is developed as a set of libraries to be connected. Agents can communicate with other intelligent agents via these blackboards. Each intelligent agent can send and receive events, rules and commands. Rules and events can be added or removed while agents are working. The information can be transferred temporally or permanently [21].

FuzzyCLIPS: It a version of rule based expert system CLIPS that can serve rules and events and modify them expertly. Besides of the CLIPS functionality, it has designed to allow the usage of expert system rules and events which is a mixture of exact, fuzzy (or uncertain) and combined reasoning, fuzzy and normal terms. This system uses two main ideas: Fuzziness and Uncertainty. It is a freeware software that can be used under Unix, Mac and PC [87].

AGENT_CLIPS: Actually, it is a multi-agent tool that has designed for Macintosh machines. Several copies of CLIPS can work on Mac simultaneously. Each agent can send CLIPS commands to other active agents. Agent_CLIPS manages incoming commands automatically. In this context, command transfer is actually interchangeability of commands between agents during execution. This is a free tool of information transfer between intelligent agents. FuzzyCLIPS applications can connect to AGENT_CLIPS as a library.

wxCLIPS: It provides a simple graphical user interface for CLIPS versions 5.1, 6.0 and Fuzzy 6.0. Actually it is a modified version of CLIPS for event-driven programming styles [105].

Some Expert System Shells for Commercial Usage: s ACQUIRE: Acquire is a knowledge acquisition and an expert system shell. It is a development environment for developing and managing knowledge-based applications. It provides field experts to configure the information and code the information directly by providing a step-by-step procedure for knowledge engineering. Direct participation of the field expert improves quality, integrity and accuracy in knowledge acquisition. It increases control in shaping software application while decreasing re-development and maintenance costs. Structural approach in knowledge acquisition, knowledge acquisition model on pattern recognition, representation of knowledge as objects, management of uncertainty, non-numeric techniques, sophisticated reporting tools are some of other features.

ART*Enterprise: It is a rule-based object-oriented development environment that also covers connective rules for applications covering whole operation. It also supports database access and team development. It has been claimed that backward-chaining can be developed while supporting forward-chaining engine.

COMDALE/C, COMDALE/X, and ProcessVision: COMDALE/C allows requests for accuracy of the proposal, results and control actions without interrupting decision-making processes. It has designed to cope with data uncertainty. COMDALE/C has the ability of time-based reasoning and open architecture. Fully object-oriented structuring, fully network capability, alarm processing, interrupt driver controller, time adjusted events, a synchronous database are among the features of this system.

COMDALE/X is an offline recommendation expert system. For real-time expert systems it includes COMDALE/C as a development environment. This shell can also evaluate hypertext documents.

ProcessVision is a real-time process tracking and software control package. It has a modular and open architecture. With its graphical interface, it provides institutional object-oriented display configuration for global process instruments, smart alerts, sensor evaluations, hot standby and limitless connectivity.

C-PRS is a Procedural Reasoning System in C [52]. It aims execution and demonstration of operating procedures. User is allowed to demonstrate and explain conditional series of complicated processes. In embedded application environments, the system runs in real-time. C-PRS is very useful for process control and audit applications. PRS Technology is applied to very diverse real-time constrained domains including, tracking systems of NASA space shuttle subsystems, control and inspection of communication networks (Telekom Australia), control systems of mobile robots, control systems of observation planes and management of air traffic. Initially, PRS is developed by Artificial Intelligence Center of the Stanford Research Institute (Menlo Park, California).

ECLIPSE is a Rule-based image matching expert system compatible with CLIPS. RETE algorithm is employed and very efficient compared to systems that do not use RETE. Notable features are data-driven pattern matching, forward and backward-chaining, accuracy management, numerous target support, relational and object-oriented demonstration and "Dbase" integration. Also, it is possible to load multiple rule-sets separately in order to test different knowledge-bases.

FLEX: It is a hybrid expert system that offers procedures, rules and frameworks integrated into a logical programming environment [9]. It can work on many different hardware platforms. Alternately working forward and backward-chaining, multiple inheritance and automatic question& answer system. Rules, frameworks and questions are defined using a Knowledge Specification Language (KSL) similar to English. With KSL, easily readable and easily manageable knowledge-bases can be defined. FLEX is developed with Prolog and has been used on many commercial expert systems.

OPS83 is developed by OPS developers. Written in C and rules can be embedded to C programs. OPS utilizes Generalized Forward-chaining (GFC): GFC is a control system which defines rules more semantically. A GFC rule can be replaced by several traditional flat rules. Current version uses licensed RETEII algorithm in order to manage much bigger and complicated rule sets efficiently. OPS83 provides a multi-windowed and click-based development platform called OPS83 Workbench [40].

RAL (Rule-Extended Algorithmic Language) is an expert system which has high performance and is based on C programming language [39]. "It allows to integration of rules and objects to C programs seamlessly. RAL is superset of C. Rules of RAL work directly on data types which is used by C codes on our applications. RAL recognizes definition of C types, function prototypes, define constants etc. C statements, declarations, macros, function calls etc. can be embedded into rules directly. RAL has open architecture that can be used with any GUI builder, database libraries or other libraries that are used with C. RAL rules are compiled to C codes for productivity. RETE II is developed by Charles Forgy for RAL. It has added extra features to be used in real-time expert systems to the language called RALRT which is a specialized version of the language. It also presents multiple windowed and click based development environment called RALRT Workbench.

Rete++ supports not only forward chaining but also backward chaining. Programmers can develop and change object hierarchies and access it using via C++, C and standard rule based syntax. Rete++ creates C++ class classifications automatically. C++ components of applications of Rete++ uses this created classes either directly or advanced sub classes if necessary. Rete++ inference engine considers examples of created class(or its subclasses) to match rule conditions automatically. C++ data types, provided by Rete++, which does not use more flexible representation over function calls but uses standard C++ syntax, allow reasoning. In Rete++, changing of C++ objects does not require obviously function calls by programmers, but monitors automatically. Rete++ is provided as C++ class libraries. Rate++ with graphical development platform or without can be embedded into code completely. The reasoning modules which are integrated to database for developing based on actual cases, come with the applications of Rete++. The multiple rule sets or agendas are suitable for modular performing and they support working with expert system. Rete++ has advanced browser for rule based programming. Windowed development platform of Rete++ monitors knowledge base which is updated in real time with multiple perspectives, allows error debugging, scan, arrange and monitor separation points and monitor rules, targets and events [34].

YAPS: It uses rule based representation in LIPS and is tool that used to develop expert systems and programs. YAPS library, provides class which is called CLOS and proper procedure, allow addition of created own classes by programmers or allow to use directly. An example about events and rules can be associated with this example. Instead of a huge knowledge base which more difficult to fix and management with a lot of rules, programmers can create more modular and more effective with small knowledge bases [2].

2.6 Domain-Specific Languages

Domain-Specific Languages (DSL) are designed specifically for predetermined problems, either as programming or specification languages. They can be used for representing specific problems of solutions [125]. They provide ease-of use as compared to general purpose programming languages, which is beneficial in numerous areas and applications. Developing a DSL is a non-trivial task, since it requires both domain knowledge, and programming language development experience, which rarely co-occur. Most DSL-development attempts cannot surpass API (Application Programming Language)-level development. Furthermore, in spite of a large body of literature in specific DSLs, there are limited resources over DSL development methods in general. This leaves open the questions of how and when a DSL should be developed [76]. In addition, it is tedious to debug programs written in a DSL, and using more than one DSL in a single organization can cause communications problems, a.k.a. 'The Babel Effect' [89].

Since DSLs are designed specifically, they cannot address problems out of their designated domain. In contrast, General-Purpose Programming Languages (GPPL) have been developed to solve problems in generic areas, which can also be a business domain.

A DSL stands somewhere between a tiny programming language and a scripting language, and often times can be used in a manner that resembles APIs. Therefore, the borders between these concepts are quite blurry. Mostly, their relationship is like the connection between scripting languages and general-purpose languages. Applying a DSL approach to software engineering brings both risks and advantages. A well-designed DSL should find a good compromise between the two. In contrast to general-purpose languages, DSLs boost specific design targets.

- DSLs are more comprehensive.
- DSLs are more influential in their own domains.
- Regarding the subjective description below, DSLs contain minimum redundancy.

The redundancy of a program is defined as the number of insertions, deletions, and replacements necessary to implement a stand-alone change in the requirements. A DSL can visual, such as the Generic Eclipse Modelling System (GEMS) [White et. al, 2007], or text-based (examples include CSS, regular expressions, make, rake, ant, SQL, HQL). The most important two advantages of DSLs is increasing the productivity of developers, as well as the communication between domain experts. A carefully chosen DSL can eases readability of a complex code block, thereby improving the performance of the programmer [76]. Domain-specific languages can be examined in three categories:

- Internal DSLs
- External DSLs
- Language Workbenches

Internal DSLs: This kind of DSLs aims to behave like a specified language using certain techniques. In fact there are limited usages of general purpose languages. This approach has recently been popularized by the Ruby community. They are also known as embedded DSLs. They use certain subsets of a language for a certain application viewpoint. LISP is a good example for this class. It is possible to think of DSLs as specific APIs. Whereas APIs define the vocabulary for an abstraction, Internal DSLs contribute the grammar. Therefore, it is more feasible to imagine Internal DSLs as a collected set of sentences, rather than independent commands. The constraints in these languages are by no means the core properties of the language. Their limits are defined by the purpose and manner of our usage of them. This is, in turn, related to which subsets of the language are used. It is common practice to avoid conditionals, loops, and variables. An external DSL has its own syntax rules. A fully equipped parser is required to process them. XML, SQL and regular expressions can be counted as examples. In spite of being targeted for specific domains, external DSLs are still general-purpose languages. DSLs can be supported through interpreters or code generation. The fact that interpreting is the easiest and commonest option does not mean that code generation is never necessary. The code is commonly implemented using a high-level language, such as Java, C, or C++.

Language Workbenches are Integrated Development Environments (IDEs) that are designed to build DSLs. These tools allow describing the syntax of the languages abstractly, in addition to generators and editors specific to it. Editors provide complex programming environments for the language. Language workbenches hold the DSL programs in an abstract manner [125].

The prominent advantages of DSLs are domain-specific abstractions, formulas, and their limited power. The reasons that render DSLs more attractive for certain application types, compared to general-purpose languages, are listed below:

- **Ease of Programming:** A DSL has a high level of accord with the problem domain, as well as a high level of abstraction. Compared to an implementation that uses a general-purpose language, it is more complete and easy-to-understand both by the developers and domain experts. Typically, this means shorter developing times and less maintenance. In addition, DSLs come with high-level debugging support systems, that enable analyzing and debugging the codes in a conceptual level directly.

- **Systematic Reuse:** Reuse has always been a method of shortening the development span of new applications. In spite of the introduction of standards and domain-specific libraries, the reuse of general programming languages is mostly left to the developer. On the other hand, DSLs force reuse of existing DSL libraries. Furthermore, since they are developed for specific problem domains, they catch the domain information and reuse this information continuously.

- **Ease of Verification:** Thanks to the advances in Software Engineering, controlled verification holds a prominent role in successful development. In general-purpose programming languages, verification can at most guarantee that a certain code will run, whereas in DSLs, due to the tidy structure of the code, and specific structuring, it can guarantee that the code will also generate the expected outputs.

In an organization, the usage of same business-related concepts eases information exchange, and minimizes the risk of discrepancy between user expectations and the reality of the business.

2.7 Domain-Specific Kits

A new concept that provides a practical application environment for DSLs is the Domain-Specific Kits (DSK). These structures, as proposed by the Software factory approach [3], are frameworks that support the life span of a DSL (Figure 7).

A DSK is an environment that has been built for the design, development, and running of a DSL software. A DSK can provide both generic graphical design build, and building of domain-specific graphical representations using source codes. Modelling tools for various fields can be built through DSKs. Defining and implementing a modelling language is a relatively easy task. For instance, user interfaces, business processes, databases, or information flow models can be defined using specific languages. Afterwards, these descriptions can be implemented and run through interpreters, or code generators. Using DSKs, we can generate specific visual designs for a certain domain. For example, during the development of a state chart tool, they make it possible to define a state, its properties, and the transformation rules to other states, using domain-specific concepts and symbols.

A Domain-Specific Kit is comprised of three components, which are:

- Domain-Specific Language (DSL)
- Domain-Specific Motor (DSM)
- Domain-Specific Tool (DST)

Conclusion

In this thesis, we use the DSL approach to define specified environments for debugging and fault tracing, in various layers of the architecture. Each subject is regarded as an 'area', and for each area a DSL can be developed by defining its specific operations and conditions. These languages can as well be text-based. Thus, the operations to be handled under each condition in run-time will have been defined as practically as possible. The Domain-Specific Motor corresponding to each area will then interpret the programs written in these languages directly, with no necessary compilation phase, which will ease the development

2.8 Checkpoint Literature Survey (Checkpoint with rollback-recovery)

This section summarizes the information acquired from result of literature survey study of Checkpoint techniques. The information obtained from different sources and presented in a summary along with brief descriptions of some of the resources listed. While much resources can be found on the subject that will constitute the basis of information sources were content with the representation.

Checkpoint approach is a technique that is being used to increase fault tolerance of the systems. The checkpoint will be applied in case of a malfunction of the unit should be returned acceptable ’previous’ state without loss of any critical information. In operation unit of the current status information periodically or prior to the operation of a critical section of code is stored. Although the checkpoint is a very costly method even for embedded systems design approaches that can be implemented with acceptable cost is available. For example, approaches such as minimizing the critical situations and memory exclusion [93] can be used as allowed. Because frequency of state storage costs too much, that’s why statically (during installation) or dynamically (change during runtime) [85] calculation of the frequency may be considered [97]. When added checkpoint to a system, the entire system’s current state is stored in non-volatile storage: The checkpoint mechanism takes a snapshot of state of the system and of the data which used and stored this information some storage tools. The cost can vary according to bandwidth of the media and amount of condition to be stored. In case of any failure in the system, it is restored from the last stored internal state. This process is done with restarting either the task that caused the error or all system with some parameters. The process can take a few seconds depend on complexity of the task will be restarted and the bandwidth of the storage unit. However, this simple method can only provide good protection for temporary faults. If the fault is a design fault that caused the failure, this system will continue to malfunction the recovery process will be repeated continuously.

A checkpoint application can be developed with the following three phases:

1. Design Time: To define at what point and under what conditions of the system, checkpoint mechanisms should be run.
2. Runtime: According to some specification such as location, frequency / time, condition, etc. made at design time, system status information stored (backed up).
3. Recovery Time: Restoring the saved status information and adjustments related management of the process. Reloading of the state, restarting of a process and, starting a different copy at a different site should be evaluated together.

A snapshot of the entire program is taken periodically as long as the program runs. In general, the running process is stopped and the status of the processes will be saved and copied to a non-volatile storage unit.

In order to reduce checkpoint operation cost, only circumstances' changes are saved during snapshot. This approach is called incremental checkpoint [93]. The purpose of this method is to decrease the cost of checkpoint operation in terms of both time and space (storage unit). An incremental backup of a disk may be considered as an example for this operation.

Moreover, with the help of memory exclusion technique, an application can inform the checkpoint algorithm about which memory area states to be used whether critical or not. This technique is used to store only the most critical information for program state. Thus, the program designers do not include the large number of working groups (set array), string constants, and other similar memory area to the control point. When using this technique combined checkpoint cost is reduced by 3-4 times. There are some techniques which aim minimizing the emerging overload during state saving [91]. These are:

1. Latency Hiding Technique: It is a technique which hides or reduces disk writings.
2. Size Reduction Technique: The purpose is minimizing data amount which will be stored in each checkpoint.The most important point of size reduction technique is memory exclusion method.

In the memory exclusion method, memory space of the process is either read-only (these values have not changed since the previous checkpoint) or dead (the value memory space of the process, it is not necessary to successfully completed of the program). The difficulty of the memory exclusion is definition of read-only and dead zones with detecting minimum load.

The methods are generalized as follows:

Dead memory: If memory region is dead while taking checkpoint, these values are not included in the checkpoint. The value in the dead memory region does not read by subsequent checkpoint. "b" array in the above example is an example of the dead memory.

Read-only memory: If a region on the memory is never changed since the most recent checkpoint, does not have to be taken from the current checkpoint. "a" array in the C2 checkpoint is an example of read-only memory.

Memory exclusion technique has two important factors that complicate the checkpoint: |

1. Definition of memory region that will be excluded
2. Maximize the memory region to be excluded per checkpoint.

In order to apply memory exclusion technique two transparent technique is developed for checkpoint system:

1. Exclude code section and benefit from the stack pointer Since code section once installed cannot be changed during execution of the program. Therefore, during the lifetime of the calculation is read-only mode and there is no need to be stored in any checkpoint. On the other hand, many checkpoint system do not save memory addresses which are at the bottom of the stack pointer region since the current value at that region will be never used.
2. Incremental checkpoint Incremental checkpoint approach significantly decrease the checkpoint size and load if easy to determine the exchange locality on application program. However, the checkpoint approach to other techniques such as fault tolerance requires additional resources. Successful operation is highly dependent on the design of the target application.

Reloading of working state: In case of any failure, recovery mechanism restores the system from the last checkpoint. State of the system whether is completely restored from the last saved snapshot or reconstructing by using incremental checkpoint technique with variation information and the last snapshot. If the source of error cannot be prevented, the system which is reinstalled can give this fault repeatedly. In such cases, the error may be hidden in the system over the period of several checkpoints. And in the end the same fault is repeated and the system will restart from this checkpoint that contains the error and failure is repeated. Therefore, the system designer should consider any of the control point might fail. This type of recurring faults can be removed with multi-level rollback techniques and algorithmic diversity [10].

Process migration is a technique for transporting of the state of a process from one machine to another one. In order to communicate and manage some of references sent to the process which is migrated should be organized. To success this operation various methods have been suggested: Some of them as follows: Sprite method only save current location in the home node [30]. In the VKernel method, new address of the process which is migrated is searched in the communication protocol level [115]. On the other hand in the Charlotte method, messages are send to all nodes already visited [7]. When a sufficient amount of state is transferred and added, the newly created process is run as a normal process. After all situations are transferred, the original process is deleted from the source node.

Existing Tools, Techniques and Measurements There are free tools that are available in a wide range for checkpoint approach. These tools, compiler-assisted static and dynamic checkpoint placement options up to the user-level checkpoint library are at different ability. Some tools are introduced in this section.

Tools: Libckpt: Libckpt library have been developed at the University of Tennessee in Knoxville [93] and has also a rich library containing incremental checkpoint and memory exclusion. Checkpoint mechanism that can be installed with an initialization file even can executable without compiling. Some properties of Libckpt are as follows:

- Active / not active
- Parallel chekpoint
- Incremental checkpoint
- The most and the least time between checkpoints
- Complete status storage between maximum number of incremental storage.
- Checkpoint compression

libFT: It is provided by AT& T research laboratories and it supports user-level checkpoint capability and watchdog mechanisms. Even if it does not provide advanced and comprehensive features as Libckpt, in terms of providing users-level checkpoint to the process libFT is more powerful than Libckpt. As a result of including the watchdog process, in case of interruption of abnormal process offers easy detection and correction capability. In addition, it has monitoring and correction capability to many processes in a system

Condor Project: It is developed by University of Wisconsin. Basically the process migration and load balancing objectives were pursued. They get rid of the difficulties of process migration with their new study which called Process Hijacking.

Porch: It is a portable checkpoint compiler developed by MIT. It has a compiler that can read C type programs. One can add checkpoint and recovery related codes into the Porch. The compiler is still beta version and open for use [111].

CATCH (compiler assisted techniques for checkpointing) is developed in University of Illinois at Urbana-Champaign. This technology diminishes the cost of checkpoint by reducing the data size and requirements [67].

Especially many new and old checkpoint tools that are used with Linux distributions can be examined from http://www.checkpointing.org/

Measurements

Although there are no ideal measurements can be used for comparing the checkpoint systems, achievement and performance of these systems can be measured with the following critical parameters [93]:

- How much time that is required to saving the state and reading back
- For this process how much time the system operation should be interrupted

The results not only strongly depend on the application and platform but also how checkpoint mechanism is applied.

Generally the following criteria is important for measurement:

- Snapshot time

- Determination of the snapshot will be taken and copy time of all the necessary state of program
- Commit time
- Copy time of snapshot to the non-volatile storage unit.
- Recovery Time
- The time that is spent for reloading of faulty process's state

All these criteria depend on snapshot of size of the state to be received and stored and performance of the system.

Current Articles

In this section referred to advanced approaches on fault management mechanism mostly. Within the framework of this thesis study, by checking of size and storage frequency of state information, which will be essential to be stored, can be concluded that these techniques are not required.

In [36], authors mention about recording only state changes instead of recording the whole state information. After the recording of checkpoint, state changes are started recording to memory incrementally. It is said to provide high performance. Because, creation of checkpoint and recording the state requires constant time only. Restoring the situation is requires linear time bounded to situation changes. This is also emerging independently from application and detailed checkpoint techniques. General-purpose applications think memories they can access as their own states, because this memory includes objects, which operated by these applications. For instance a document processing application stores documents that processed by users in memory. For this type of applications it will be sufficient to save history of memory writings on checkpoints. In this article authors are not endeavouring with external device conditions and developing a productive checkpoint approach according to above definitions. They resolve memory accesses in the source code by using a program analyzer. In cases which status information changes during processing, a checkpoint added automatically. In fact they are benefited from aspect orientation in this stage. Program has written in Java, however within some limitations it is said that it could be apply on some languages

In [84], CC (Cooperative Chekpointing)theory is designed to increase the system performance and reliability. This study has made a comparison between periodical state storage and CC. By using CC both run time system and application programmer can decide how and when checkpointing could be done. Generally programmer adds checkpointing to application code in areas which it could be productive freely and abundantly. Application requests a checkpoint during process time. System allows or refuses that (disc or network usage and reliability information) on the basis of various heuristics.

Constant and prepositioned checkpoint frequencies cause compromising from time and source based quality features. The article [23], authors present a behavior-based framework structure. They specify that, checkpoint consumes time and energy, that's why checkpointing at very frequent intervals causes time and energy restrictions especially at embedded systems.

A checkpoint system for rollback recovery intended and has been evaluated in [10] for use in embedded systems. A pre-processor is designed for evaluation of the intended system. Approximately 3000 tests has been carried for each checkpoint. Rollback techniques are grouped under three main headings:

- Global reset
- Rollback recovery
- Roll-forward recovery

In the first two techniques fault detection mechanism returns the system to its starting point or a different point just after the detection of a presence of a fault. Both two techniques are cost-effective because there is not a necessity for any additional resource. However there could be wasting of time because of system's returning to its starting point or a different point. At the roll-forward technique, system has to have two copies of a program which are running simultaneously and one of them has to work properly. A separate processor has to be used for checking this. This technique increases costs while it annihilates waste of time of the other two techniques. Choosing one of these techniques is depending the trade-off between success and cost. The intended technique in this study has designed based on control flow error. The program is divided into simple blocks at compile time. These blocks consist of a set of commands without jump commands or jump target addresses in them. Afterwards error control mechanisms are added in these blocks. Checkpoint is also applied as an interrupt. When a control flow error is detected, error control mechanism saves the system by reloading it from its last correct saved state by informing error recovery process. The memory, which stores checkpoint has to be fault tolerated.

The study in [83] firstly a checkpoint modelled by using a half Markov decision process and afterwards the most appropriate checkpoint policy estimated statistically by applying reinforcement learning algorithm (Q-learning). Checkpoint is placed on processing time on different time points like T, 2T, ... Each of this time points are called as decision epoch. Decisions made on whether or not checkpoint at each time point. This plan is considered as quite logical especially when checkpoint is located sequentially. If its decided on checkpoint at any point, it is forced to divide two to two forks as parent and child processes. The child process saves the state of parent process. On the other hand parent process continues to run. Either processes continue to run in parallel state. This time period which child process saves state in it, is called as latency. Overhead is described as the increase at the runtime of the process. Fault management intervention consists of two phases: correction and retry. After completing the roll-back, the process will be retried and intervention will be completed. It is assumed that there won't be any checkpoint during these two processes. Dynamic checkpoint mechanism is developed on the basis of Q-learning which is an algorithm of a reinforcement learning.

The incremental checkpoint process approach is intended for minimize the additional load at the checkpoint by saving only the changed pages of a process in [49]. The problem in this method is many updated values can be saved to the same page so this increases the total size of the incremental checkpoint as time progresses. The authors developed a page-level incremental checkpoint method in this article which is called Pickpt. The method minimizes the usage of disk space. Test results showed that Pickpt has decreased the disk space usage significantly according to existing incremental checkpoint approach. Various methods developed to decrease the load of checkpoint. These methods are classified into two main categories:

- Methods of decreasing latency time: Example:
  - Checkpoint without disk
  - Forked checkpoint
  - Checkpoint with compression
- Size reduction method: Example:
  - Memory exclusion method
  - Incremental checkpoint

Both of these methods try to decrease the data amount which stored in each checkpoint (i.e. per each checkpoint). With size reduction method, unmodified memory pages or large amounts of memory that read-only are determined and all of them are deprived from checkpoint. In practical applications incremental checkpoint is one of the most used methods. The method determines "the dirty pages" which changed since the most recent checkpoint by using page fault mechanism. The biggest problem of such conventional checkpoint is the usage of an unpractical checkpoint file. To do a full checkpoint for recovery, only retention of the last checkpoint file will be sufficient. However, the old checkpoints cannot be ignored at incremental checkpoint approach, because memory pages of the process are expanded on many checkpoints. That's why performance of the storage unit decreases

Tool Comparison

In this section, the current comparisons, and also a new comparison targeted to the parameters that will be related to this doctoral study are given. Different comparisons were not included in the same set of tools. Nevertheless, different comparisons constitute a very informative source about the most of the tools.

User Level or Kernel level Checkpoint

Generally, a checkpoint application is implemented either kernel level or user level and sometimes combination of these. There several advantages and disadvantages of these approaches. In user-level checkpoint applications, there is no need to make changes to the operating system. Therefore the operating system remains unaware both checkpoint and recovery operations. In this case, in order to store and reload state of a program, the user-level checkpoint/restart application must examine a wide range of system functions. Because a checkpoint application must reach the much information (memory mapped region, file open operation etc..) which is required watching the state of the program in both when backup and reload. Another difficulty in user-level checkpoint application is it cannot advantages of SMP (Symmetric Multi-Processors) of threads. Because kernel does not know these threads in the user-level. On the other hand in the kernel-level checkpoint application, kernel can adjust the timing of thread and keeps track of their status. Moreover fault management application can easily access the desired information in the kernel-level checkpoint applications. And some operating system also provide a checkpoint/restart mechanism in the kernel level, application programs does not need to do any preparations for the checkpoint. However user-level checkpoint applications are more portable than kernel-level. Briefly user-level checkpoint does not require to do any changes in the kernel-level (writing to code, add, change etc.)

Comparison Tables

The first informative comparison was done by Eric Roman in 2002 [99] Table 2.3. The result of this study is depicted in Table. Then in 2005, an outstanding study was done by Kim [64] is shown in Table 2.4. Finally comparison study was done for this study is summarized in Table 2.5.

Conclusion

Checkpoint mechanism is effective for fault management as well as being a complex mechanism. Many parameters can be applied as a result of different requirements with different values. There are many example information sources, tools, and application available for this doctoral study that will be developed with different requirements. Nevertheless, enough design effort should set forth for each new application. Fault management mechanisms is an application that information cost such as location and time can be quite a lot. That's why in every adaptation, options should be used properly by optimization in accordance with the requirements.

Table 2.3: Comparison was made by Eric Roman in 2002 [99]

	libckp	**libckpt**	**Condor**	**libtckpt**	**CRAK**	**BPRoc**	**Score**	**CoCheck**
Type	library	library	library	library	library	library	library	library
Scope	process	process	process	thread	sub process	process	parallel	parallel
File data		good						
Resource usage						good		
Reputation					good	weak		
CP handlers			weak	good	weak			
Signals			full	full	full	full	full	full
File descriptors	good	good	good	good	good		good	good
Address space	good	good	good	good	good	good	good	good
Registers	full	full	full	full	full	full	full	full

Table 2.4: Comparison was done by Byoung-Jip Kim in 2005[64]

	Crak	Blcr	Epckpt	Ckpt	CryoPID	Duation	Zap	MCR
Version	Linux 2.4	Linux 2.6	Linux 2.4	Linux 2.6	Linux 2.6	Linux 2.4	Linux 2.4	Linux 2.6
Application	Kernel+application	Kernel+application	Kernel patch	Library + utilities			Kernel unit + library	Kernel patch + application
Year	2001	2005	2001	2005	2005	2005	2002	2005
Single thread process	OK	OK	OK	OK	OK	OK	OK	OK
Multi thread process	X	OK	X	X	planning	OK	OK	OK
Process groups	OK	planning	OK		planning	OK	OK	OK
Sessions		planning						
Signal state / handler	OK	OK		OK	OK		OK	OK
Resource constraint	X	planning		X				
Timer			planning	X				
Current working directory	OK		planning	X				
SYS V IPC	X	X	SEH SHM	X			OK	OK
Normal Files	OK	planning	OK	X	OK	OK	OK	OK
stdin/stdout/stderr	OK	OK	OK	X		OK	OK	OK
Mapped files	OK	OK	OK	X	OK		OK	OK
X			OK	OK				
TCP/UDP sockets	OK	X	X	X	OK		OK	OK
Tool files	X		X	X			OK	limited
MPI programs	X	OK	X	X				OK

Table 2.5: Checkpoint tools comparison table (2009)

	Transparent*	Incremental	Parallel	Memory Exclusion	Compression	Synchronous	Single/Multi Processes	Process Migration	Watchdog	Level
Libckpt	OK	OK	OK	OK	OK	OK	Multi	OK	OK	User
libFT	OK	X	X	X	OK		?	OK	OK	User
Condor	OK	OK***	OK	OK	OK	OK	Multi	OK	OK	User
NT-SwiFT**	OK	OK						OK	OK	User
EPCKPT	OK	X	OK	X	OK	OK	Multi	OK		Kernel
CATCH**	OK	X	OK	NO	OK	OK	Multi	OK	OK	User
CRAK	OK	X	OK	X	OK	OK	Multi	OK		Kernel
CryoPID	OK			X	OK	OK	Single (planning)	OK		

*Transparent: modifying the user program is not required. Kernel-level easy, very difficult at the user level **CATCH ve NT-SwiFT mostly developed for network applications. *** With a little modification.

CHAPTER 3

FAULT MANAGEMENT REFERENCE ARCHITECTURE

3.1 Introduction

This section defines the reference software architecture developed under this doctoral study. This reference architecture will enable development of software architectures which will form a basis for fault management infrastructure software. The structure proposed here forms a basis for fast development of various software applications that may satisfy fault management requirements which may vary from one project to another. The product, being defined by this document is under development and has the opportunity to diversify according to actual needs. Targeted use, after finalization of diversification-related decisions on actual needs, may enable:

1. fast development of application-specific fault management architectures, and

2. fast conversion to desired fault management software since it is considered that key units of such reference architecture will be coded in time. Customized architectures would be able to develop through selecting and combining those coded units.

Applications expected from which to employ fault management infrastructure should be developed together with some additional features. The proposed architecture and supportive units will be facilitative to application systems. The applications should enable instillation of features necessary for demanding such support in their early stage of development (such as requirement notification, design stage).

3.1.1 Architectural Views

In this study, several models that describe various aspects appropriate for architectural views. This representation includes many aspects while not being designed primarily for architectural aspects. Architectures are asked to provide their aspects through various models. Today, UML-based 4+1 perspective is widely accepted. Such perspective covers use case-based design, process, implementation and deployment aspects. The design aspect, indeed, correspond to logical aspect in Krutchen's 4+1 perspective. Therefore, the UML class diagram-based design aspect was accompanied by conceptual modelling. Since the report did not organized to have these aspects with their topic especially designed for them it would be facilitative to explain which parts support these aspects. Use case view: This view is provided in details together with use case diagrams under relevant requirements and sequence diagrams that describe the use cases.

Design view: The architecture's conceptual units and relationship between them are modelled by supporting this aspect in addition to class diagrams that correspond to architectural units. Classes included in the class diagrams, in addition, are used to define internal structures of several components in the models for implementation aspect.

Implementation view: Describes components that are defined as the main units of the architecture in various parts isolated under similar names. Such components placed in different logical layers and relationship between them form the frame of the reference architecture. In the first part of this section several models that reflect also the implementation aspect are included.

Deployment view: Different possible distributions are described and two important deployment drawings are provided in the first section of the part on implementation.

Process view: This view refers to the control structure among the tasks. A serious complexity on this matter was not identified in the resulting architecture; which means no detailed model is needed.

3.2 Studies Conducted

The literature on Fault Management was reviewed. Targeted features were determined after assessing the determinative features of the domain according to the objectives of this doctoral study. In order to develop an architecture which supports such features, Software Architecture Design methods which would support also the concept of Software Product Line were examined. The reference software architecture was designed by use of such methods.

During the architecture design, firstly the scope was determined and the requirements were modelled in addition to development of a feature model. Requirement modelling was structured by use cases and supportive interaction models (sequence diagrams). A layered structure was arranged, relational model of the concepts defined in the requirements was developed by use of such information; and corresponding software unit abstractions were located in the layered structure in a structural manner. Thus, models that correspond to implementation and deployment views of the architecture were developed.

3.2.1 Feature Model

The feature model study is the preliminary stage of the requirements study. Main characteristics of the domain were determined after the literature review and in the feature model various features that would be necessary for different fault management infrastructures were provided in a graphical model. Requirements of the infrastructure to be developed to enable such features were provided in the form of another model. The feature model was employed as a guide for transition to the requirement model. There is no organic link between the feature model and the requirement model. Figure 3.1 shows the general concepts of the feature model.

3.2.2 Requirements

A general study on requirements was already conducted under this thesis work. In this work, existing report was considered and improved to enable it to support a reference architecture by proposing some changes. The requirement study was supported with graphical models; for which UML compatible environments were employed. In Figure 3.2 a use case diagram that deals with main features for the targeted fault management infrastructure is provided. In the following parts of this thesis descriptions for each use case included in several scenarios are presented.

Since the repair function shown in the use case model is an off-line function then its content is not modelled in this study. However, system decision on the start-up of repair and notification on the end of repair will be carried out through reporting.

The communication functions, not shown in this use case but observed during the literature review, are assessed in lower level details. In addition, this function is included in the model through another indirect way by distributing it among levels of detail reflection relating to the communication layer.

Similarly the state management function should be discussed here which seems irrelevant to operating mode that may be called main cycle but which will be needed at the restart of the units. State back-ups should be made against any potential fault before such faults occur within the necessary units. Such backed-up cases will be used during restart procedures.

Typical procedure flow is from the monitoring to detection and then to diagnosis. Recovery is the first option for any diagnosed fault. For any fault that cannot be recovered is referred to repair function. Recovery is composed of, depending on the type of the fault and the unit concerned, isolation of the unit from the application, activating its substitute when necessary, and sub-operations of case loading and restart which is the last step of the recovery process. In the remaining part of this section use cases are discussed in details through several models.

3.2.3 Use Case Descriptions

In this part the use cases are explained by use of sequence diagrams. For some of the use cases multiple scenarios are necessary to be assessed; which means multiple sequence diagrams for those use cases are provided. The use cases are taken form 3.2 and explained and discusses one by one:

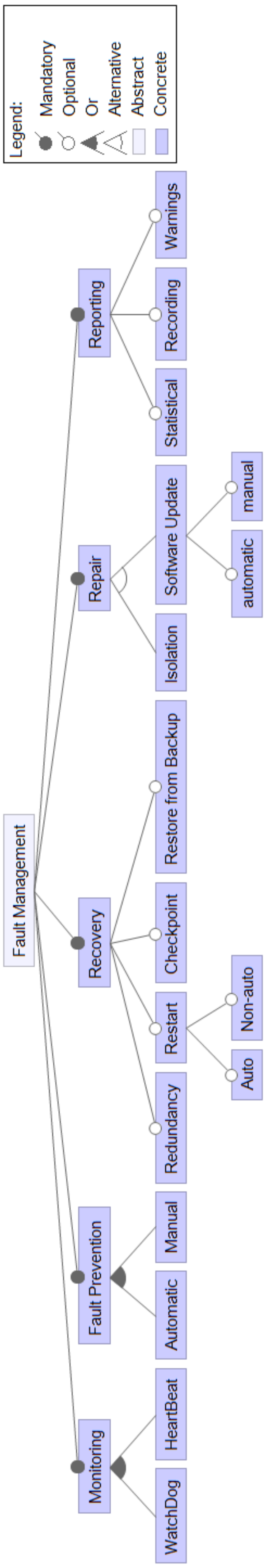


Figure 3.1: Detailed feature model of the Fault Management

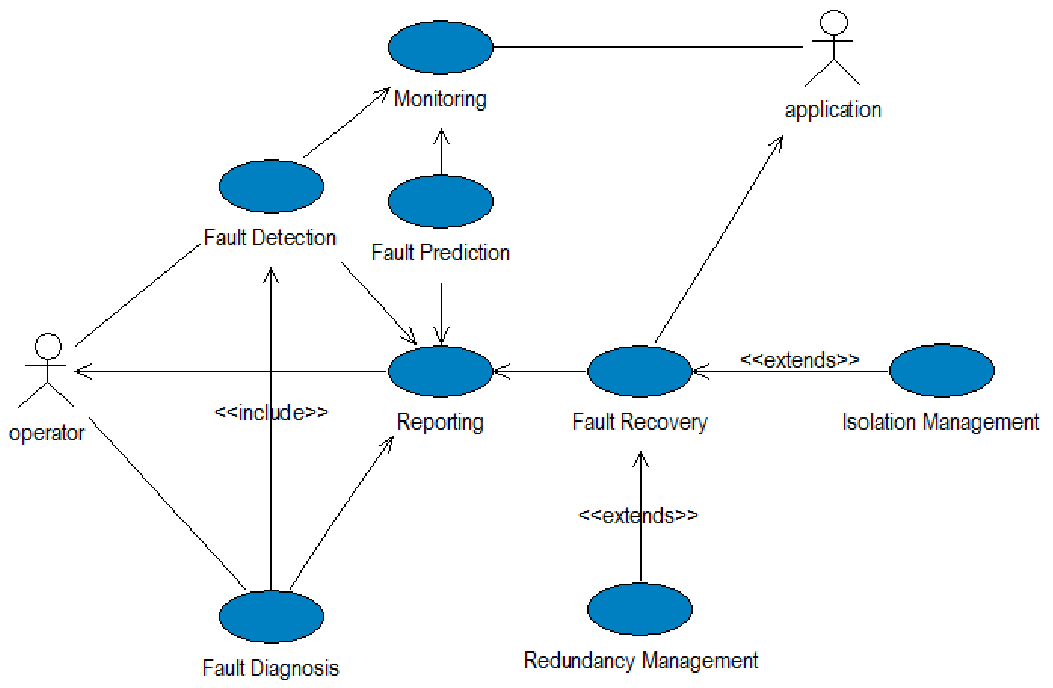


Figure 3.2: Use Cases

3.2.3.1 Monitoring

Monitors any event that may indicate any fault.

Concepts relating to the Use Case Monitored: Memory, processor use, tasks, data from units (temperature, voltage, etc.) notification data (traffic, time-out, data accuracy, etc.) the message "I am alive (heart beat)", etc.

Monitoring Methods: Operating system functions, symptom monitoring functions

Monitoring Point: Connection points from hardware, software or operating system to the fault management system that are used for monitoring a certain fault symptom. Connection points will be used only for data traffic.

Monitoring:

- Monitoring will be started and stopped on demand.
- Monitoring start/end criteria will be definable (ex; start/stop time, monitoring duration).

- Monitoring frequency will be definable.
- For any certain symptom type, historical data will be stored on a rules basis for a defined amount/period (history cache).
- Symptom types should include the following:
  - Those that should be received in a certain time
  - Those that should be lower/higher than a certain threshold value
  - Those whose data should have a certain value or pattern
  - Those that will be sent to detection without any assessment process (ex; messages covering ultimate fault information like BIT (Built-in test): In this study the BIT Messages are deemed to be at direct detection level. However, some other messages that will be deemed at the symptom level, in general, and of this type sometimes contain upper-level information).
  - What to do with the data (send, write down, etc.) may be subject to rule-based determination.
- What to do with the data (send, write down, etc.) may be subject to rule-based determination.

The need to start monitoring operations later without any continuous monitoring process may arise from the following cases,

- When monitoring negative affects the operational mode (to be started possibly with the user trigger)
- When monitoring negative affect the performance (to be started with the user trigger or with the decision given by the diagnosis, depending on the decision to be given)
- When there is no need to allocate resource for such operation as no continuous need to related information is predicted even if monitoring does not have any significant negative effect on the operational mode and the performance. When up-to-dateness of the monitored data is not sufficient for the diagnosis generation of up-to-date data may be requested regardless of the monitoring period.

Sample Scenario 1:

The "I am alive (heartbeat)" message is sent from the monitoring point (through interrupt method). Monitoring monitors the delays of "I am alive" message, and informs the detection unit of the delays according to the pre-defined rule. The detection units, decides whether there is a fault or not after assessing the relation between the delays in the heartbeat message.

Sample Scenario 2:

Monitoring reads out the temperature value from the monitoring point (polling method). Monitoring, if the temperature value is higher than a predefined value in the rule, sends a message to the detection. Detection decides whether there is a fault or not after assessing the temperature value.

3.2.3.2 Detection

Receives and employ data from monitoring as input. This unit decides whether there is a fault or not after assessing the monitoring data (which may be sent from multiple monitoring points) according to a rule or set of rules. Such rules may be operated under a certain scenario. Such scenario should not extend beyond taking decisions on existence of a fault. However, in some cases detection of a fault may also mean it's diagnosis.

The detection unit will be able to store historical fault decisions for a certain number/time period. User will be able to start another more comprehensive detection operation in comparison with the continuous detection upon any detection by use of new monitoring data.

- The predefined set of rules cannot be changed by the user trigger. However, new rules that do not operate under normal conditions can be run with receipt of/demand for new data.

- Fault detection demands initialized by the user are expected to be more comprehensive and, in some cases, able to influence normal operation.

- In the comprehensive detection, monitoring of several points that cannot be continuously monitored due to influences on performance or normal operation will be enabled.

An asynchronous symptom flow is expected to occur under normal circumstances between the detection and monitoring. The detection will be able to detect any fault that occurs as well as that which disappears. It will also record any change in the fault status.

3.2.3.3 Prediction

The prediction employs monitoring data as its input. The prediction should be able to access monitoring points and have the feature to automatically activate monitoring operation. It will also give a decision on fault probability after employing a rule or set of rules on the monitoring data (current or historical data) The prediction predicts any probable fault

- by capturing predefined symptom patterns; or
- depending on the operation time.

Detection of any patter will be carried out off-line. The patterns will be sent to the prediction in the form of a set of rules (ex; if symptoms of type a, b or c occur with this order then there is a possibility of fault X to occur. Recovery method Y should be applied).

Appearance of such patterns may be traced by use of two methods:

- When a symptom is received relevant pattern is searched by use of the past symptoms the Symptom Log.
- When a symptom is received which pattern's beginning this symptom is (if the symptom is beginning of a pattern then this pattern is searched among those which are being traced) or which pattern's, started to be traced previously, next step this symptom is should be determined. Runtime may be traced by the relevant unit or the Prediction. Time-dependent triggering is another option.

Statistical data may be changed at the runtime (other than the Prediction) and some time-dependent rules may employ such statistical data in the runtime. Therefore, the Prediction is able to access to both the prediction rules and the statistical records (fault prediction will be able to be done through other statistical information such as MTBF (Mean Time Between Failure)).

3.2.3.4 Diagnosis

Diagnosis will start upon fault decision given by the Detection. A synchronous fault flow is expected to occur between the Prediction and the Detection when normal conditions prevail. The diagnosis will be able to activate a query to find the source/reason of a fault. There exist different sequence diagrams for different diagnosis scenarios that vary according to the result of detection, previous information and reason of activation such as user demand.

Statistical Records will not be employed during the Diagnosis stage. User will be able to initiate the diagnosis. The diagnosis service to be demanded by user is reasonable only when symptoms other than those that are continuously monitored will be monitored or queried. (The diagnosis may also be triggered by the user input. Diagnosis will continue after determining appropriate rules from the "Diagnosis Algorithms" according to the demand type).

The diagnosis will have access to any kind of detection and monitoring history. By use of such access source of a fault will be determined by getting the advantage of temporal relation between the faults. It is proposed that the Diagnosis triggered by Detection will carry on working when it completes its operation while the Diagnosis triggered by user will suspend when its operation ends. However, because of the complexity of the rules in some cases, the Diagnosis triggered by user firstly receives the rules necessary for diagnosis (such rules also include necessary Detection data which is not available at that time) then collects data in accordance with those rules, and finally gives a decision. In the most general case the diagnosis will be triggered by a fault or user, necessary rules will be received and necessary respond will be given in accordance with those rules (such as generating ping message), and finally diagnosis result will be generated by making queries. The diagnosis will not be able to directly initiate monitoring but demand information on fault from the Detection.

The diagnosis will have direct access to the Symptom logs. Alternatively, monitoring data which does not indicate existence of any fault practically but which has a probability to be employed in diagnosis may be stored after passing it through the detection. However, since this alternative would cause performance (processor and storage) problems it was decided to enable the diagnosis to examine such monitoring data only in case of need. This access may be removed in the future if it is decided that the need of the diagnosis to use symptom logs disappeared.

When operating diagnosis rules units which were already labelled as failed or other isolated units should be taken into consideration. If there exist any communication with faulty or isolated unit within the rules then generation of incorrect results by the rules should be prevented. If possible, alternative rules should be employed in order to generate the most appropriate results with existing information.

Example-1:

User may initiate a diagnosis demand throughout the system before starting a mission. With the user triggering the diagnosis, the Diagnosis will determine what data it needs and in what order by use of "Diagnosis Algorithms" and then send a demand for queries/activation to be started which are necessary to receive such needed data.

Example-2:

The Diagnosis which receives a fault that communication is interrupted through the detection will determine what data it needs and in what order by use of "Diagnosis Algorithms" and then send a demand for queries/activation to be started which are necessary to receive such needed data in order to find the reason of fault.

3.2.3.5 Recovery

Recovery will be activated by the results of the Diagnosis or the Prediction. Appropriate Recovery Rules will be obtained by use of information on the fault or potential fault received from the Diagnosis or the Prediction. Following methods may be employed provided that such Recovery rules are respected:

- Use of isolation, when necessary.
- Establishing checkpoints
  - In order to start from the latest state in case of restart
  - In order to start from the latest state in case of activation of redundant unit
- Restart in the absence of any redundant unit
  - Restart of hardware
  - Restart of software (task, process)
  - Cold (without saving the state) or warm (by saving the state) restart
- In case of any redundant unit activation of the redundant unit as the active unit
  - Activation at redundancy state 2N, N+1, N+M
  - Cold, warm, hot restart

The Recovery will understand whether its operation is successful or failed with the information sent by the Diagnosis while the Diagnosis will update its decision by considering the information sent by the Detection or that requested from the Detection (example: x unit failed/not failed). The Recovery, if the targeted fault did not disappear after a recovery operation, may repeat the same operation (Example: re-reset the relevant unit) or initiate a more comprehensive recovery process (Example: reset of the entire unit after it is detected that the operation conducted on the task failed during the previous recovery process is unsuccessful) or carry out no operation in order to recover such fault even if the fault did not disappear by deciding that the recovery process failed. When the Recovery receives an information that the fault is removed after a recovery operation then Recovery may, by informing the isolation of the same, request removal of the isolation which is previously activated. Cases that require repair, together with their time, which will be used by the Prediction in the future are recorded on the Statistical records by the Recovery.

3.2.3.6 Isolation

Isolation aims to protect the remaining portion of the system from effects resulting from any fault.

- Isolation may be divided into two groups: Physical isolation
  - Example: Dismantling the hardware from the system
- Logical isolation
  - Data isolation
    - Example: Using different RTPs (real time processes)
  - Fencing
  - Example: Deactivating input and output by use of the unit to be isolated
  - Example: Blocking notification by removing the failed node in the network from the routing table

3.2.3.7 State Backup

This is designed for preparation to states where state loading function to be used during restart should be used. The checkpoint capability requires storage of the state periodically of under certain circumstances. State back-up scenario carries out such state storage procedure.

3.2.3.8 Reporting

Reporting activities are those included to send the results of several processes such as recovery and diagnosis to user (operator). This can be done by displaying various visual/sound messages on screen or just recording it on a log.

3.2.3.9 Repair

Repair means removal of a fault that results in failure; include debugging and re-compilation of the software. When a unit is re-included in the system after repair the "time spend for repair" which will be used for prediction is recorded on the Statistical records by the Repair. The Repair, indeed, is an off-line process and function relating to repair is not given in this report in details. The Fault Management does not have any task relating to repair other than recording the repair start time and the repair end time.

3.2.3.10 Notification

Notification does not correspond to a different use case, and is included to enable the afore-said communication in different use cases. It is aimed to create the notification as a separate unit and thus to isolate communication method from the units between which communication is provided. Such units will be connected to each other through the Notification under normal circumstances. Appropriate communication type (such as function call, message queue, socket, CORBA, DDS) will be provided by properly structuring the Notification. In some special cases direct function call between units may be available due to need to fast communication.

3.3 Reference Architecture

In this section concepts relating to the reference architecture study and infrastructure elements such as layers are described. Architectural views and information on application-oriented models and adaptation discussed in the latter sections are also included in this section.

3.3.1 Conceptual units of the architecture and their relations

This section describes mainly the logical view (according to Krutchen) or the design view (according to UML). Conceptual architecture units and relations between them are given. Some of the units mentioned here may be used as components of the fault management infrastructure in the future while in some cases they correspond to several concepts that may provide different application options. For instance, a communication point (conceptual unit) may, after a variation resulting due to application-specific requirements, become:

- a layer interface function (API);
- a piece of code that must be embedded into an application software; or
- a ready-to-use and autonomous component which operates like an agent.

The architectural units which are closer to application are described in the section that refers to the Implementation view. The conceptual units cannot be implemented directly. However, such concepts, in general, are implemented by their architectural components that appear as their sub-units. In Figure 3.3 main conceptual units of the fault management architecture and relations between them are provided. Then related class diagrams can be driven for details.

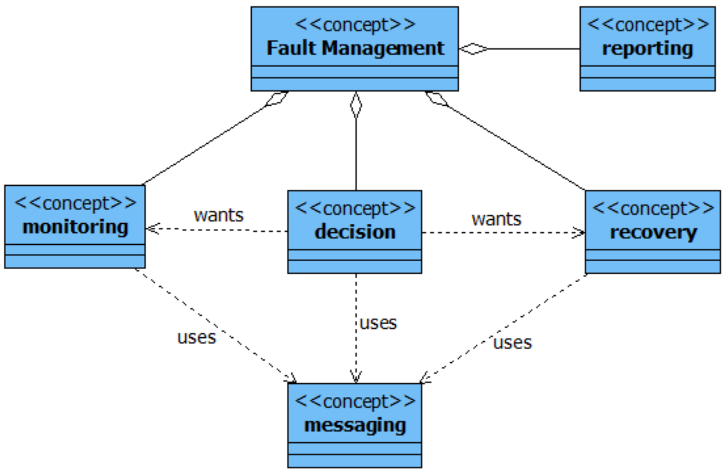


Figure 3.3: Conceptual Units

The core of the architectural concepts is composed of the monitoring, decision and intervention units; among which decision is the central unit. In case of user interactive routing such routing will be started at this point. Symptoms to be monitored is collected by the decision unit; routed, when necessary, by requests; comments are made, intervention unit is asked to respond after potential (estimated) or absolute faults. The intervention unit has key functions such as fault prediction, detection and diagnosis of the detected fault. Recovery operations that may be conducted on-line are the most important interventions. Faults that cannot be recovered on-line will be directed to repair functions which are fulfilled off-line.

The function within the conceptual structure which are the loosest relate to the repair and a little bit reporting activities. During the repair, communication relating to outer world concerning the starting and ending of such activity is targeted. Reporting will include brief information to be sent to the user by various units. Such operations are those which may be considered as less related to real-time and critical nature of the application; and are not elaborated much in the figure. For example, only connectivity of the reporting concept with other functions which may be easily predicted are shown. On the other hand, the unit which will exhibit the highest formal connectivity to other units at the application level is the messaging unit although it is not so important at the conceptual level. However, this connectivity, when considered in terms of "generality: means availability to be adapted for various project requirements" which is one of the most important criteria in this project, may be obtained through connection protocols at different levels. In other words, a strictly coded structure is not imposed or enforced.

In the following sections details of the main concepts are described in details.

3.3.1.1 Monitoring Concept

The monitoring corresponds to testing various parameters within the operating system or application with a certain procedure. The message "I am alive" are also included in this function. Information about parameters to be monitored and related methods will be provided in the system installation. In addition, a more detailed or customized monitoring methods will be able to be triggered in case of any challenging fault diagnosis operation.

Monitoring methods will be in communication with the point to be monitored; which points will be defined in the application software or in the operating system. Availability of selection between synchronous and asynchronous protocols would be useful for such communication that facilitates messages. Such selection between synchronous and asynchronous protocols may require not doing anything when waiting for the respond after making a request at the level of fault management application and storage of messages that cannot be received immediately within the message layers (such as mail box).

3.3.1.2 Decision Concept

Core features of the fault management applications relating to control are gathered in the decision unit. The units that correspond to the upper layer in a layered structure are those which relate to the decision concept.

Diagnosis is dependent of decision to be given by the detection. Under normal operation Detection is responsible for generating a decision on whether a fault does exist or not after assessing the monitoring messages. The Prediction concept undertakes a responsibility which correspond all operations to be carried out in some way by the detection and the diagnosis. However, the Prediction is responsible for faults have not happened yet. On the other hand, the Detection and Diagnosis conduct necessary operations after an actual fault.

3.3.1.3 Intervention Concept

After diagnosing or predicting a fault it comes to an effective function if it is possible to eliminate the fault on-line "recovery" functions will be employed and the system will be forced to carry on its normal operation. In practice such recovery operation will need to several operations such as getting a redundant unit into use, restart of failed unit or maybe isolation of the failed unit that may be necessary for such operations. If on-line recovery is not able to fix the problem that it will be necessary to take the relevant unit off-line and to repair it. As there is no important operation relating to the repair it is not accompanied by a dedicated drawing.

Complex operations relating to the intervention are, for example, loading case information which was backed up previously before restarting the failed unit and in addition testing the same information before re-activating a unit which was deemed to be fixed after a critical case.

3.3.1.4 Reporting and Messaging

The Reporting and Messaging concepts correspond to operations which are needed by many other concepts. The reporting concept does not have many features that is necessary to discussed in details. Information about control-related units and when necessary monitoring and intervention units should be displayed in a format compatible with the pre-designed reporting format. Such information may be compiled by use of information reports relating to actual faults or collection of historical information according to certain criteria. Information storages from logs which are deemed necessary to exist within relevant units may support the report creating studies.

The messaging concept is an expression of the feature of various units of a system to find each other and send messages according to different timing requests, and which is provided through middleware layers today. Such messages may include information or control commands. After definitions relating to conceptual definitions that correspond to the "design view" from among the views within the UML-based architecture several modelling will be provided for implementing such concepts with their different architectural units. Architectural units, components that compose the implementation view, class type architectural elements and those deployment layers will be provided.

3.3.2 Architectural Layers

The aim of the project is to provide fault management capability for embedded software, in general. In order to provide a fault management environment that is suitable for applications to be developed it should be noted that basically three software types will operate at the same time. The given three software types are:

1. Fault Management software;

2. Operating System; and

3. Application software.

Moreover, an additional middleware layer may be considered in order to provide support in messaging, for example. In some cases it may be necessary to implement such middleware layer as a component of the Fault Management. The communication layer will be used in general; together with the application software is also necessary for internal communication of the fault management. Figure 3.4 shows the related layer structure.

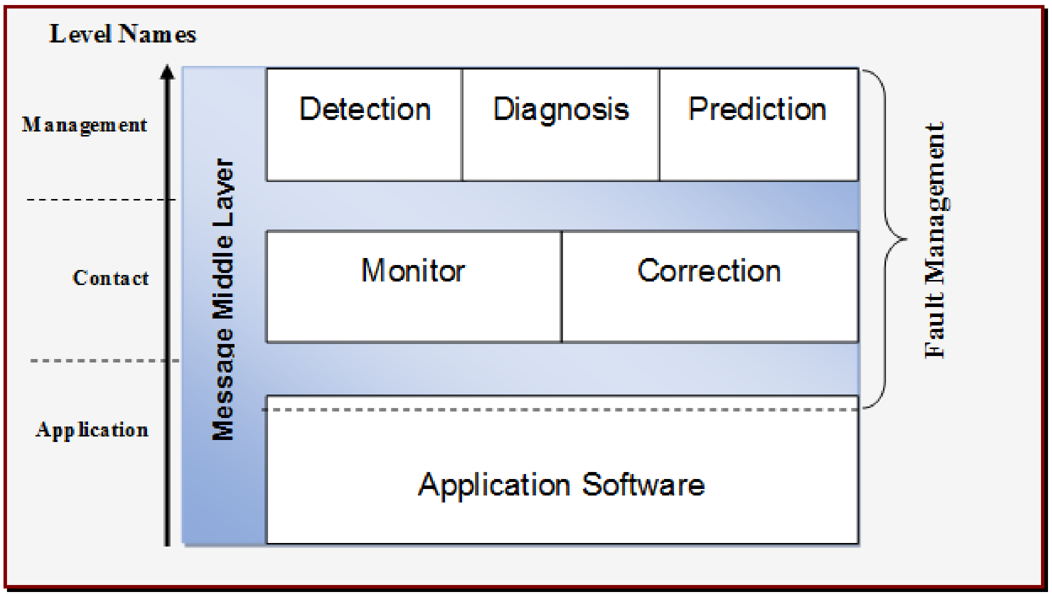


Figure 3.4: Fault management software layers

As seen in Figure 3.4 the Fault Management is composed of Management, Contact and Implementation layers; and the Message Intermediary Layer which will enable integration of and connection with the mentioned layers and the Application Software and the Operating System. The Management layer is composed of Detection, Diagnosis and Prediction components; and detailed information on these components are given in Section 3.4.1.2, 3.4.1.1, and 3.4.1.3, respectively. The Contact layer is composed of Monitoring and Recovery components and will be in closed contact with the application software and the operating system. Detailed information on these components are given in Sections 3.4.1.7 and 3.4.1.4 respectively. The Contact layer, as the name implies, is divided into sub-units relating to functions that require contact with the implementation and operating system for receipt of monitoring information as well as for responds to be given, and that are closer to implementation. The management concept is the layer where units that are rather related to management exist and that is not in direct contact with the implementation layer.

3.3.2.1 Fault Management Hierarchy

When it is considered that more than one fault management units operate at the same the need to a hierarchical decision mechanism becomes apparent. Therefore, a sub-fault management system should be connected to the upper one at detection level. In other words, a decision given by a sub-level fault management system should be delivered to the Detection component of the upper-level one, regardless of content of the decision. Within the frame of this general understanding a BIT is also considered a fault management system. The systems to send messages such as BIT will also be considered a lower-level fault management system and prevailing method will be employed. Some information that is about fault diagnosis at its own level may be considered as a component fault for "system of systems" at the upper level; and may need to be assessed together with potential faults and conditions of other components at the same level. BIT information, despite the possibility of corresponding to Fault Management messages, will be directed to the system as only detection information and, when necessary, message delivery from the first connection point to other units (Diagnosis, Recovery units) will be made. Messages are developed after tests which may, sometimes, be conducted continuously; and some of them contain a set of information. Such information which will be developed as test results may address a fault or normal operation. Basically such information are also considered a symptom in terms of Fault Management. However, there may exists which are more comprehensive or which are considered a fault message. The BIT data obtained from the detection level and which indicate a fault should be sent to the diagnosis unit. The diagnosis may need to examine other symptoms in addition to the sent one. Such additional symptoms may be requested from other units in order to examine probabilities like fault induction.

Information received which does not address any fault is an symptom for the Fault

Management. However, such information will not be those which are continuously monitored by the Fault Management. Therefore, it is not necessary to direct them back to the monitoring level after being received at the detection point. Many of them may be ignored. How to use which BIT information and also whether to ignore such information or not should be application-based programmable through a development point feature of the Fault Management. At such information input points programmable connection points may be developed which has the feature to scan incoming messages (parser) and which enables recognition of formats of incoming message packages and loading of information on how to manage the incoming messages within the Fault Management system.

3.3.2.2 Message Middleware

Message middleware have wide range of use. Such layers, being considered a system software, arise as structures in which complex software systems are increasingly used. They are facilitated as a natural extension of and even services included in the operating system. Complex software systems, as units having contact with each other, find their place in an architecture and use the given defined common services system for communicational purposes. The message middleware undertakes several services such as connection types to be required by different messaging protocols and buffering; and isolates the application from such challenges. This layer should have an overall structure that covers necessary communication methods. In other words, application developer should not engage in technical details of lower-level communication. Such work should be transferred completely to the message layer. Applications examined show that communication traffic, particularly, in embedded systems is composed of messages with very short periods. General information on the communication intermediary layers are given in Annex A.

Practical Use of Intermediary Layers, in general, synchronous communication is enabled by use of RPC-type infrastructures. On the other hand, asynchronous communication is enabled by use of MOM (Message Oriented Protocol). However, it is absolutely possible to develop a MOM-based synchronous communication. In case of need to very immediate intervention the most appropriate technique is the RPC.

MOM is the best option for asynchronous and decoupled communications. However, today many applications require very diverse messaging methods. For instance a typical company may wish to have both synchronous and asynchronous communication methods to be supported by their system. For this in addition to simple one- way FTP protocol, more complex middleware such as MOM or RPC may be used. For example pipelining-RPC supports standard **synchronous** RPC protocol in addition to **asynchronous** communication protocol.

A middleware which supports the fault management for embedded systems may vary according to application-related needs but it would be useful to employ a featured layer such as a message-oriented or object-based layer which would provide the option to call for functions between closer units; so that different messaging features and resistance to fault are provided. Fault management and application software do not show a close connectivity to the middleware but calling points of messages to be sent at the interface should be structured according to the middleware. Such method will provide units such as Detection and Monitoring with communication feature through a messaging middleware or through direct communication.

Middleware Interfaces should be defined at the communication points of the units in order to enable not only the use of different middleware, and use of function call approach when necessary but also improvement of isolation between the units within architecture. If the interfaces are structured to be two-stage such different use becomes available without any need to change the internal structure of the unit including its own interface (first stage interface). At each stage, requests that may be received by the unit (standard interface concept includes such definitions) and other requests to be made by that unit from other units should be defined. With a simplified example, the **Detection unit** in its interface should indicate that it will request monitoring information from the **Monitoring unit** and also detection information may be requested from it. Under normal circumstances, the information to be requested by the **Diagnosis unit** from the **Detection unit** is defined as an (commonly known) interface element as a method of Diagnosis. In addition, we should also know needs of the Detection to other units in order to operate properly and what it will request from who. Instead of such interface structures, called as adaptor or proxy, in the second stage a single stage interface structure may also be used. In this case, however, the units should have different interfaces whose technology and connections are previously estimated, and during the application those selected from among the considered items should be used.

3.3.3 Class Diagrams

Classes are the fundamental elements of architecture. They model data-oriented structural units necessary to implement concepts. When sufficient details are provided they also define units relating to the code to be implemented. In this section different classes which compose the design are grouped in different class diagrams. On the other hand, class relations are not given. The relations which arise in the form of a data flow-based connectivity are already modelled within the components which include those classes. The classes which form the Fault Management system are shown in Figure 3.5.

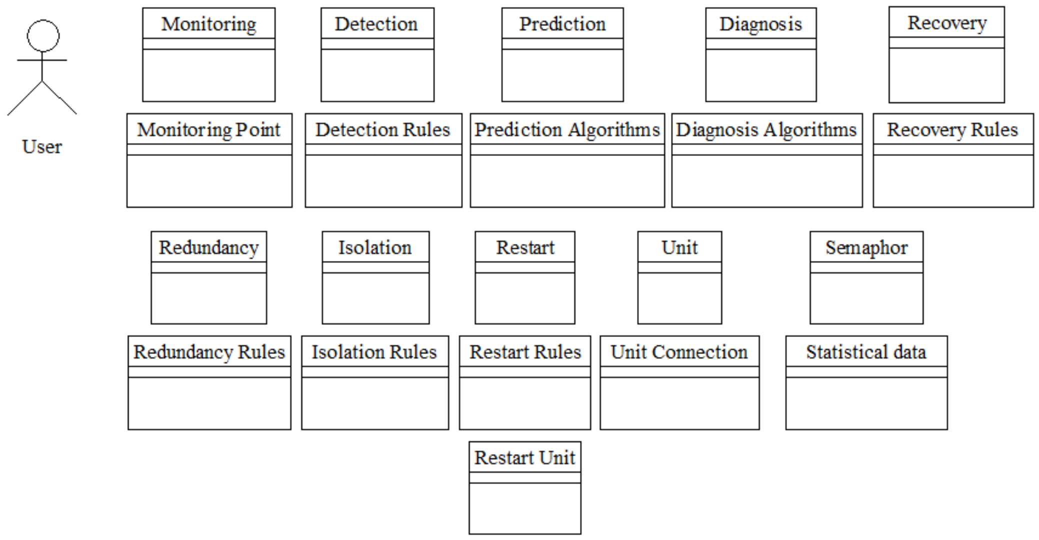


Figure 3.5: Classes relating to management

3.4 Components within the Architecture

In this section, structural components that exist within the architecture are described. Firstly, fundamental units which will undertake operations to arise from requirements (Functions of Fault Management System) are discussed. The structural units are:

- Main units which undertake control of monitoring, detection and diagnosis operations;
- Auxiliary units which will provide the above-given units with support; and
- Auxiliary units which will provide connection with other layers.

In this section, building stones of the reference architecture are provided together with their upper-level components that compose them.

3.4.1 Essential components and their distribution

Distribution of Fault Management units in the form of a three-layer structure was presented above. Units at these layers may either share the same processing hardware and thus the same network node or vary according to the network nodes among which they are distributed. Monitoring points would be the most distributed units when located in a more distributed application structure. A symptom monitoring unit should exist at each node having a monitoring point. A symptom monitoring unit will be able to monitor more than one monitoring points. Due to increasing distributedness requirements recovery points may be increased in number and distributed among different nodes like the symptom monitoring unit. In the most distributed cases, in addition to distributed detection units diagnosis units (and similarly prediction units) may also be distributed such that they form a hierarchical structure. Existence of a single diagnosis point would facilitate understanding the architecture when it is predicted that it would be valid for many systems.

A relatively more delicate distributedness example starts from lower levels is provided. Here only monitoring points and symptom monitoring units that check the monitoring points are deployed by being repeated at distributed nodes. The next concept which will run in connection with the application like the monitoring is related to the recovery. When the **symptom monitoring unit** and the **monitoring point** duo is taken into consideration it can be said that they correspond to the **recover** and **restart** duo. In this sense, a similar distribution structure arises for the recovery-related operations.

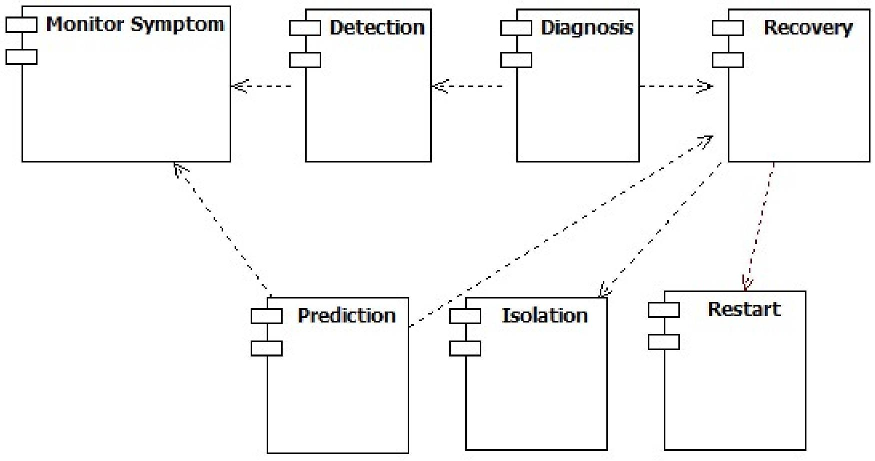


Figure 3.6: Component model for architectural implementation units

For example, recovery requests coming from the Diagnosis unit are routed to the central recovery management unit. The central recovery unit, after assessing the received recovery information, sends it to relevant Recovery Point which is already distributed and coordinates the recovery process carried out among these distributed recovery points. In this case (distributed architecture), each recovery unit should have different rules algorithms. After necessary discussions it was decided to use a Central Recovery Unit that will provide the coordination among the distributed recovery points. Fault management functions are mainly deployed at the Fault Management layer which is modelled with the Fault Management Central Node. The components mentioned here are isolated on the basis of essential system functions; and the components included the given essential functions and other units relating to their auxiliary concepts.

In the remaining part of this section internal structures of the components in the implementation view described in details.

3.4.1.1 Diagnosis Component

This component is the one that is over the control structure. It has an essential role in interaction with the user. A rule-based algorithm runs for the fault that is detected and if necessary request for a more comprehensive detection may be made; which may result in different monitoring activities. It was previously stated that the components include units that correspond to the fundamental units and auxiliary units. After diagnosis either recovery or repair operation is called. Decisions given relating to the diagnosis are recorded in a log.

3.4.1.2 Detection Component

Detection operations, like those for diagnosis, needs to be coded as rule based. These rules will be employed to enable assessment of the incoming monitoring messages. Decisions given relating to the detection are recorded in a log.

3.4.1.3 Prediction Component

Prediction component is the last one among the components which form the decision-making procedure. It is designed to provide a chance to intervene before any fault occurs. Since it will be activated before any detected fault it has direct access to monitoring messages. It receives information on how to assess such messages from its own set of rules. Decisions given are recorded in a log.

3.4.1.4 Recovery Component

This component also needs a set of rules since it operates on the rule based. Similarly decisions given by this component should be recorded in a log.

3.4.1.5 Isolation Component

This component also needs a set of rules since it operates on the rule base. Similarly decisions given by this component should be recorded in a log. Isolation operations will be routed by the recovery management unit that is a central component.

3.4.1.6 Restart Component

This component also needs a set of rules since it operates on the rule based. Similarly decisions given by this component should be recorded in a log.

The state loading operation given here, in some cases, requires a distributed mechanism. Information on state may be reloaded from a local storage, another similar (redundant) application unit or a central storage depending on the location of the state storage. It is thought that this unit being distributed but concentrated near the application (instead of loading from just one local storage) is acceptable to mitigate the communication traffic in case of large state re-loads.

3.4.1.7 Monitoring Component

The monitoring unit is also a unit that may be deployed at the application layer and that may be used after being multiplexed more than once. Each symptom monitoring unit will be able to command more than one monitoring point.

3.4.1.8 Monitoring Point Component

This component will be embedded into the application software or the operating system and, nonetheless, will be considered as a part of the fault management system. Saying nothing of intervention to the operating system, it may not be possible, in some cases, to embed an absolutely defined code part even in the application software. This component, if possible, may be carried out as a component and in other cases as any method that is able to provide the fault management with communication opportunity (for instance, a function call that will be kept open to communication by the operating system). This component, for its overall functions, will be connected to the communication middleware through an interface. Although such existence of an interface is available for many other components this interface undertakes relatively major function of the monitoring point.

This unit is a unit that will have direct access to the application software and the operating system and located by embedding it into these systems. When this component will be located in the operating system, in general, it should be deemed only as a logical expression: in such cases, there may not be any operation other than calling a defined function or any unit that will be shown. In other words, the monitoring point may be composed of just this function. If the unit which requests a service from the monitoring point through an asynchronous way then the unit should know what to do with response which may be delivered later at any time. Therefore, if this kind of communication will be used it should not be expected such issues to be resolved at the message layer level. Such requests should be made deliberately in a synchronous or asynchronous way. For synchronous requests no other operation would be carried out until the relevant response is received and the requester should wait. In general, asynchronous communication means that the unit which sends a message is able to carry out other operations without any need to wait for the relevant response; and it is agreed that the result will be delivered as a return parameter or with another message after a finite time. For a synchronous message the request sender waits for the response and cannot carry out any other operation until the waited response is received. However, both options have simplified modes. If the asynchronous message will not wait for its response then it becomes a simpler mechanism in comparison with the synchronous message. For the synchronous message the sender party should wait for at least the verification of message communication. On the other hand, for the synchronous message if the risk to lose the message then it becomes a simpler mechanism: no check will be made. In addition poll/interrupt modes should be defined depending on whether the "symptom monitoring" unit which is a more central unit requests for delivery of the monitoring data or whether the "monitoring point" which is a more distant point in comparison with the symptom monitoring attempt to send. When a typical scenario is assumed if a terminal which will periodically send data is able to send its data without central approval then this is called "interrupt mode. "poll" means sending the data as a respond to a question after a request.

Methods that should be at a monitoring point and a more central unit that will communicate with such point should have such expectations. For instance, in case of poll method a central unit (typically a "symptom monitoring" unit) will be activated by methods of the monitoring point such as "get symptom". Otherwise, in case of interrupt model since the monitoring point tries to send its information it needs methods whose name is similar to the "set symptom" definition in the symptom monitoring unit. In short, for the poll method there is need to the "get symptom" method at the end point and for the interrupt method there is a need to the "set symptom" method at the central point. This communication type addresses the features that may be used in the entire Fault Management approach and thus to the communication approaches.

3.4.2 Pattern in Component Structure

In many components described in the previous section a significant common point can be seen. It will be useful to consider this structure as a "design pattern". This structure which is repeated in many components is composed of a set of rules, a log and a control class. Moreover, interaction among these three classes exhibits several similarities which can also be found in the relational connections in these diagrams. We can call this pattern "rule-based control". Structural model of the Rule-based control pattern is given in Figure 3.7.

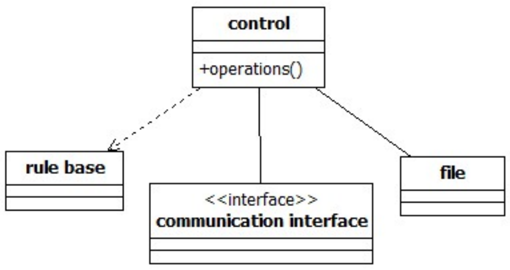


Figure 3.7: Rule-based control pattern

This design pattern models an appropriate conception in order to form a flexible structure aiming at availability in terms of various applications. The rules, here, enables expression of application-specific variability configurations. Application-specific adaptation-related information all along the entire mechanism such as how to monitor a certain symptom, how to conduct a detection may be entered into the rules without any need to programming, by use of commonly used terminology of the application site and the interface. The system should be able to reload these rules at their latest version in start. All components which were mentioned within the implementation view in the previous section may be designed according to this pattern. Already, identical similarity of the control component that correspond to the topmost layer is obvious. However, the middleware interfaces also exist even if these are not displayed at these components. In addition, the symptom monitoring and recovery components may also be designed according to such rule-based control pattern. Even the monitoring points and the user units which are the nearest to end points may be modelled in this manner. However, it may be possible to solve the rule basis issue more easily at the application stage in terms of efficiency at the monitoring points.

3.4.3 Interfaces of Components

In this section interfaces of the defined components are described in terms of services to be provided and received by them and by showing with which other components this service trade will be made. It should be noted that these interfaces may be supported with various communication methods and may be realized by using various communication middleware or through direct function calls. Published and required sections of interfaces exist.

3.5 Studies towards Implementation

The reference architecture given in this report will be adapted to software to be used for fault management after being elaborated. Such adaptation will be carried out by giving emphasis on determination of set of rules on which the component will be based. These rules will provide identification of the units and information on how to provided fault management services according to these units for instance, at the symptom monitoring level, the rules will provide information relating to how to monitor a monitoring point, identification, location of that point, which method (central management [poll] or end management [interrupt] methods) will be used, and whether the messages will be synchronous or not. In addition, between what threshold values a symptom found in a given point will be filtrated will also be determined by use of such rules.

To give example on the recovery-related rules, what kind of backups a unit will have, at which frequency and where a state will be necessary to be backed up (checkpoint) can be given. Rules relating to how to assess symptoms from a unit according to the relevant unit and how to assess information within the symptom message will be used in determining the detection and diagnosis algorithms. In case of support by a Feature Model what error management features are expected from the Feature Model may be defined. Such features may also be repeated for other units. After providing such information it will be possible for the reference architecture to become an application-specific absolute architecture. Components of this reference architecture may be coded (by versioning for different platforms) in the future so that it will cover options at the variability points to be determined by use of a feature model that will be extended. When the features are selected, the resulting fault management infrastructure will be formed as a ready-to-run sub-system. In this section several techniques relating to customization of the Reference Architecture are provided.

3.5.1 Structuring with Design Patterns

The pattern consistently resulting in the Reference architecture showed that the defined components may be considered as a "Rule-based Control Pattern". In addition, considering different details of the process it can be seen that it is able to employ common patterns defined in the literature. For instance, for the state backups, a development which is appropriate for the "memento" and "state" design patterns may be provided. For the communication-related needs the "proxy" pattern may also be taken into consideration. For the communication points of distributed components which have to use different platforms, the "adaptor" pattern that will provide compatibility with the interface may be useful. .

3.6 Use of the Reference Architecture

As is seen particularly in the deployment diagrams several options such as existence of multiple copies of some of the components, and various distributions of these copies will be available. The structure of the application, fault management features which are selected according to such structure and locations of the necessary components will enable transition from the reference architecture to the desired system architecture. The architecture should be converted into fault management software customized for the application by addition of necessary details. It would be useful to be aware of the architectural structure of the application initially. Determination of different fault management requirements of the application components, having decision on different techniques on how to fulfil such requirements and then structuring fault management architecture compatible with the application architecture would be reasonable way to use. It would be useful to establish a feature model that will enable selection of different fault management features for an efficient use. A feature model was developed also in the report. However, this model was used to determined fault management requirements at the initial stage. A feature model adaptation for the product line conception may be very useful. In addition, various versions of the architecture according to different applications of its components, which are stored in a library as coded components it would be enabled to find fault management software from that library as new applications are developed. During each improvement specific needs should be met and it should be taken into consideration to improve such units in a way that enables improvement of such units to be components targeted to re-use. Such an effort may not be shown several projects if the project deadline is so strictly defined. However, such effort should be shown as much as possible. Upon completion of a project such units which are just completed should be reviewed and detailed. The points that should be taken into consideration in the course of development of a component are version options or adaptation features of the component in terms of different requirements, different communication protocols, different rates and platforms. For cases with limited performance constraints more general adaptable components may be developed. In other cases a separate version for each case may be obtained maybe by performing the adaptation in the pre-compilation environment. The rule-based approach would be significantly important for, particularly, the units which are at the management level. Functions of the components will be mainly defined by the rules. One of the most important steps towards making the architecture applicable is designing and then testing the rules.

3.6.1 An example to Rule-based Process

Sets of rules are the principal element that will show the way in which many unit will operate. To give an example, the following scenario which covers the event chain symptom-prediction-recovery is given. In this example the rules relating to prediction or fault detection to be carried out after the warning message activating at the symptom level are provided. First of all, characteristics of the symptom message arising at the monitoring point should be assessed:

- Symptom Point: CPU monitor at the operating system (temperature, voltage, frequency, load)
- Symptom Type: Temperature increase
- Reason of Warning: Violation of default threshold value

A message relating to the processor temperature is received by the Symptom monitoring unit from the monitoring point. The rules defined in this unit indicate which temperature values should require a message that should be transmitted. Table 3.1 gives the rule on generation of message for the processor temperature.

Table 3.1: Rule on generation of message for the processor temperature in the Monitoring unit

Type of Symptom	**Condition to generate message**	**Message to be sent**	**Receiver**
Processor temperature	Processor temperature >62	Processor hot (63)	Prediction
Processor temperature	Processor temperature >90	Processor hot (96)	Detection

The monitoring point detects the temperature rise in the CPU and sends such information to the Central Symptom Monitoring unit. Alternatively, information sent from the Symptom monitoring point is assessed by the Monitoring Unit and then sent to the relevant unit (Prediction or Detection) after being filtered. This information received by the Prediction Unit as a warning is compared with the table of rules. In our example, the Prediction unit firstly checks if the CPU_FAN is on or not. If the FAN is not operating the Prediction unit sends a potential fault decision to the Recovery unit together with a unit number. If the CPU_FAN is operating then its revolution speed is examined. If that speed lower than that specified in the table of rules then the same potential fault decision is sent to the Recovery unit. In both cases the Recovery unit shuts the system down and sends a warning to replacement of the FAN to all units. Table 3.2 provides the rules to be applied within the prediction unit when the said warning message on the processor temperature is received by the prediction unit.

Table 3.2: Rules to be applied for the prediction unit for processor temperature warning

Rule No	Request	Condition	Operation	NextRule /Exit
1	Fan operation	no	Recovery request	2
2	Fan speed measurement	F.S.<threshold value	Recovery request	3
3	Processor voltage	Vcpu >thresholdVcpu	Recovery request	4
4	Time frequency	Fcpuclk >Fmax	Recovery request	5
5	Processor load	Lcpu >thresholdLcpu	Recovery request	6
6	Warning message!			Exit

If the FAN is operating the rule number 3 in the table of rules relating to the relevant unit (CPU) is applied. This rule refers to whether there is an increase in the CPU voltage. If the CPU voltage value has increased then the Recovery unit is informed of the same; the Recovery unit decreases that value and then re-checks the CPU temperature. If the temperature is lower than the threshold value then it sends this information to the relevant units. The units to be informed in this case are the unit which requests for a recovery operation and other recovery units. In addition, the same information may be sent to the user as a Reporting process.

If both the CPU Fan's speed and its voltage value are within the acceptable range then the Prediction Unit applies the rule number 4. According to this rule CPU frequency is checked. If the CPU frequency value has increased the Recovery unit is informed of the same. The Recovery unit decreases the value and then re-checks the CPU temperature. If the temperature is lower than the threshold value then it sends this information to the relevant units. The 5th and the last rule refers to the load on the CPU. Load distributions of the processes that use the CPU are assessed and the number of the process which cause the extreme load on the CPU is sent to the Recovery Unit (the rule may not let process use the CPU more than a pre-defined extend). The Recovery unit kills the relevant process and all other processes that the original processes is linked with; and then re-checks the CPU temperature. If the temperature is lower than the threshold value then it send this information to the relevant units. A rule-based operation may be even more complex. While establishing such sub-mechanisms structures that enable perfect programming, and that are easy to learn how to use it, do not include complex implementations and that may operate faster may be designed. For instance, interference through parsing of rules may be conducted at the installation stage so that a run time environment may be developed which will not spend time for that process in the course of the run. In addition, designing displaying structures that makes perfect rule entry easier would prevent arbitrary rule entries.

3.7 Options for the Preliminary Design Stage

During the preliminary phases of the study several assessments were made on the basic architectural styles. Some of these first trends were eliminated in order to provide a more general infrastructure without being influenced by a certain pilot application trend.

3.7.1 Black Board Architecture

A black board-based architecture was assessed. While discussing this approach the facts that different information from different points will be received and that different units may draw different conclusions from the given information depending on the application criteria were taken into consideration. However, it was concluded that a more structured and strictly defined decision-making process would be more efficient when in particular the limitations were taken into consideration. The fact that the expectations from fault management are structurally obvious eliminated the need to connive at the inefficiency source such as collection of communication and local decision-making features at a certain point. The points at which various information will be gathered and assessed are more or less defined in this study like other fault management approaches. With the reasons stated above it was concluded not use employ the black board architecture.

3.7.2 Layered Architecture

Another preliminary assessment was conducted on the layered structure. Layered structures cause inefficiency although they are easily understandable and provide a traceable structure. However, it was concluded that there will not be a heavily layered mechanism required in the latter phases of the study and slightly layered mechanisms are preferable for studies of this kind. In addition, it was also considered that such layered structure naturally support a hierarchical deployment: In this study, there is a necessity to repeat the elements which are closer to the application at different points in a distributed environment. Therefore, it was agreed to employ a light and flexible layered structure which included a limited hierarchical structure and only two layers.

3.7.3 Control Flow

For the study on the architectural needs the communication with the existing systems to be provided was determined as the starting point of the study. Consequently, a way starting from the lower level concepts such as symptom monitoring up to the upper level checkpoints was selected; which means from bottom-up structuring approach was adopted. As a result of this approach the user requests are able to initiate specific monitoring operation at the lower layers. During latter assessments it was concluded that the user interaction should be limited to the upper level control units; which also conceptually reasonable. Therefore, it was concluded that the operator interventions as well as the upper level control units (units undertaking the diagnosis and prediction tasks) should have limited interaction to the topmost relevant level without ignoring any level.

For instance, if a fault has occurred and it has not been diagnosed by the system then additional checks may be required by the operator. In this case, instead of the operator initiating specific monitoring operations by getting in touch with the symptom monitoring or with the detection functions which are a layer above the monitoring initial determination of actions to be activated and initiating them from the topmost layer, which is the diagnosis-related unit, was found more reasonable. Necessary routines will be formulated at its own base of rules that correspond to level of each unit.

3.7.4 Built-in Test (BIT)

It was found necessary to assess a feature which already exists and relates to the fault management such as the Built-in Test (BIT). After BIT operations several messages covering comprehensive information will be generated. Such messages may be assessed at a lower level since these messages will be received by the fault management from outside. However, messages from a lower level, as included in the architecture, correspond to more unprocessed symptom information while the BIT messages may possible include information relating to operations such as diagnosis which is at the topmost level. Delivery of these messages to the relevant operations was also assessed. In other words, ıt was planned to transfer the information being transmitted directly to the detection or diagnosis operations at the relevant level instead of engaging the capacity of the system at the symptom messages level. Then these opinions were abstracted and concluded that the BIT information is at the most appropriate level in terms of the detection level and thus such information should be sent to the detection level. In addition, understanding where the incoming information should be delivered is another operation that should be settled at the first contact point. The BIT signals which will pass through a couple of points without causing a heavy message traffic will be able to reach the Diagnosis operation.

3.7.5 Managing Message

Features that may provide a message receiving unit that can be defined without any coding effort through an "expansion point" regarding how to send such comprehensive input info to which appropriate point of the fault management system may be developed later. As a result, the signal flow was structured to be from bottom-up and the control flow from top-down. Another point similar to the control flow is the type of information to flow. Firstly in was thought that the raw information from the monitoring points would flow continuously but then it was decided to inform only the changes. Although a continuous raw information flow means simplicity transferring only the changes would not cause a significant complexity; however, it will provide a chance to eliminate major portion of the network traffic. According to this opinion, for instance, even if there is a continuous information flow to the symptom monitoring unit such information will be sent to the detection unit only when this information points out any fault and constitute a set of information that is not known previously. Similarly, even if information that may correspond to fault continuously comes, another message will not be sent to the detection after the problematic information initially monitored but when the incoming information starts pointing out that the relevant fault disappeared then necessary information will be given to the upper level units. This approach has been footed on two basic reasons:

1. Reduction of message traffic; and
2. Reduction of operation intensity.

Although the operation seems to the same in general it would be reasonable to reduce such intensity on the upper points (for example, over the monitoring point several other points may exist such as the symptom monitoring, detection and diagnosis, respectively) which will have to process many other information. It would be useful to employ such filtration actions closer to the relevant source, as much as possible: operation intensity on this point as less and more accurate and efficient filtration may be conducted at the point the filtration is fully engaged.

3.7.6 Intelligence at Monitoring Point

The monitoring points are first contact points of the data to be delivered to the fault management through the application software. The monitoring point, in order to receive this information, have to get in touch with the application. At this point the information is either delivered as raw information (the only function of the monitoring point is transmission) or processed by more intelligent structures. Works to be done here may be increased and reach at an intelligent agents capability. The consensus on this point is to provide simplicity of the monitoring points and that the filtration process may be facilitative but symptom monitoring units should be used to do this. The Symptom Monitoring units will know under what circumstances and what values relating to the data they are monitoring they will send messages to the fault management system, through a rule-based operation. Filtration process should not take place at higher level the mentioned one. In particular, it would be useful to determine simple monitoring points for the monitoring point and the symptom monitoring units that will operate at the same node. In order to enable the proposed reference architecture to be used by different systems the connections to the application system should be easy to learn and easy to erect. Therefore, the monitoring points should be kept simpler.

3.7.7 After-intervention Notification

A fault that has been diagnosed is waited to be recovered. However, there exist two principal approaches on this matter: 1- Diagnose, request for recovery, forget: the monitoring and detection information will flow upwards. If there is no new fault information received then the recovery is deemed to be successful; if such information received then recovery should be repeated. 2- wait for confirmation of the recovery success after requesting for recovery: This approach is compatible with the approach "delivery of only the change information" which will be expected also from the messages from the lower levels. In addition, it may be necessary to carry out the recovery action on certain critical units by a unit which is isolated from the system and to re-connect it back to the system after the recovery is confirmed. In such cases, confirmation of recovery is needed. Such critical components correspond to cases in which fault cannot be tolerated but other measures may be taken if it known that it is not operating and when the system can operate without the given component or that the task should be quitted.

3.7.8 Separation of Diagnosis and Intervention operations

In an example on the rule-based operation given above how the prediction unit will move after a warning related to the processor temperature. The prediction unit, with the guidance of the rules, was able to make sequential tests and to determine the next step depending on the test results. However, before this point a discussion was made on how to make the prediction or diagnosis units (which have power and skills equal to each other) more effective through intervening approaches. For instance, there was a possibility to increase the measurement frequency in order to decrease the processor temperature, which means measuring the temperature value with intervals of a couple of minutes, and therefore to determine if the prediction made will be correct or not. However, each condition which will be specified by use of rules for the prediction, diagnosis and recovery processes should be known; which means in this case there is a necessity for the operations relating to the recovery to be separable. Although in some cases gathering two types of feature at the same point would be effective it was concluded not use such structures in order not to disturb the regularity of the system as that requires specific rules. In this case necessary measurement can be made in a desired order and many problems can be diagnosed in this manner. We can, then, refer the diagnosed problem to the recovery unit just for once.

3.7.9 Central / Distributed Management

The essential components at the management layer that compose the architecture will, in general, operate with a central algorithm. Some of the unit that are closer to the application may be distributed. In this case the option to design the recovery operation in a distributed way was discussed. In case of restart of more than one application units a method that may require coordination such as use of a sequence may be employed. The mentioned coordination may be provided through a distributed algorithm. However, distributed algorithms increase the probability of error. During the discussions it was decided that the distributed processes to be experienced would not so complicated to deserve a distributed algorithm. In order to provide such coordination through a central approach the recovery operation may have a component at the management layer.

3.7.10 Hierarchical Integration of Fault Managements

Like in the discussions on where to connect the information coming from the BIT units different fault management systems may need to operate in an interconnected way. In this case a hierarchical structure was agreed upon. Each fault management system will carry out valid operations within its own structure. However, the hierarchy should be structured to find the source of the overall fault by assessing the diagnosed results at the upper fault management point by the fault management of more than one systems when considering a more general fault that cannot be understood alone by the deficiencies resulting in the operation area of a single system while many systems of this kind are operating together. In this case, like in the BIT example, diagnosis results of the lower systems should be delivered to the upper system's detection unit.

3.7.11 Overall Approach to Discussions on Options

Related decisions were taken by taking some important criteria into consideration for the discussed options. The mentioned criteria are: functionality of the fault management, network traffic, use connection to the application software and generality.

3.8 Conclusion

Here, a reference software architecture that may be used for Fault Management infrastructure software is provided. Different fault management features that will be needed by different applications may be found by customizing and elaborating this reference architecture. It would be useful to develop adaptations of the given architectural elements, tools to be used for making selections and various adapted versions of these elements.

CHAPTER 4

RULE BASED USAGE AND CHECKPOINT APPROACHES FOR FAULT MANAGEMENT INFRASTRUCTURE

4.1 Determining the Method

In this section, results of a study for determining a method on how to impart Rule Based Usage and Checkpoint abilities to this doctoral study are presented. A reference architectural design for fault management applications, which could be supportive at various applications for various institutions, was made at previous works. Some applications have been developed in the industry based on that architecture. The mentioned reference architecture foresees a structure that can be easily adapted to any required fault management. In order to achieve this adaptability, points that can be programmable on rule level have been considered. These points will be within the architecture and applications. A literature survey had been made on rule based systems and domain specific languages within the context of this project of which this report is also a content. A similar study had been made also for Checkpoint approach. This ability is not included yet in current applications and a study is needed on its application. This section presents a proposal for a method on how to achieve the ability of adaptability to different applications and, with similar intentions, checkpoint ability by using a rule based approach at fault management reference architecture.

The proposed reference architecture for fault management offers fault management islets that work with a layered structure on different levels and in a distributed style. Different rules will be used depending on the types of these elements. In order to prevent language differentiations within the system integrity, how the rules will be stated with a single language approach should be defined. However, it is possible to propose unit language definitions that could be defined as different sub-units of this rule language. In this case, for units that undertake different fault management duties (ie. surveillance or detection), different rule languages could be proposed. Very simple installation information could be enough for some fault management elements and it could be felt like there is no need for an expert system that would appear along with complicated inference expectation. However, it would be useful to apply a consistent approach within general structure of the system. Target here is a fault management that can be easily adapted: instead of developing a specific fault management system for each new application by coding new programs, it is desired to shape adaptations with a high level specification. This high-level specification can be realized with rule structures that will reflect the definitions about the domain. As a result, new systems can be realized with a method that does not require much time and will gain the ability of Software Production Line. The environment in question, which has been constituted with an understanding of language that is specific to the domain in order to run it, is shown at Figure 4.1. We can assume to have some general requirements that would be applicable to the rule based languages that are considered to be defined:

1. Rule defining languages must be easy to learn and use.
2. Consistency and similarity must be sought at language structures.
3. Languages must be supporting faultless entry of domain specific terms and cognizances and thus, supportive of swiftly forming of different fault management applications.
4. Elements like 'status control' and 'message transmission' that should be kept in hand at rules' state and action stages must be defined by using terms and mechanisms that refer to domain's expertise.

Aside from language's own properties on faultless and swift development, support of tools can be more important. Because of this, also the development of tools that would support expression process of these rules will be useful.

For a Rule based system to be executed:

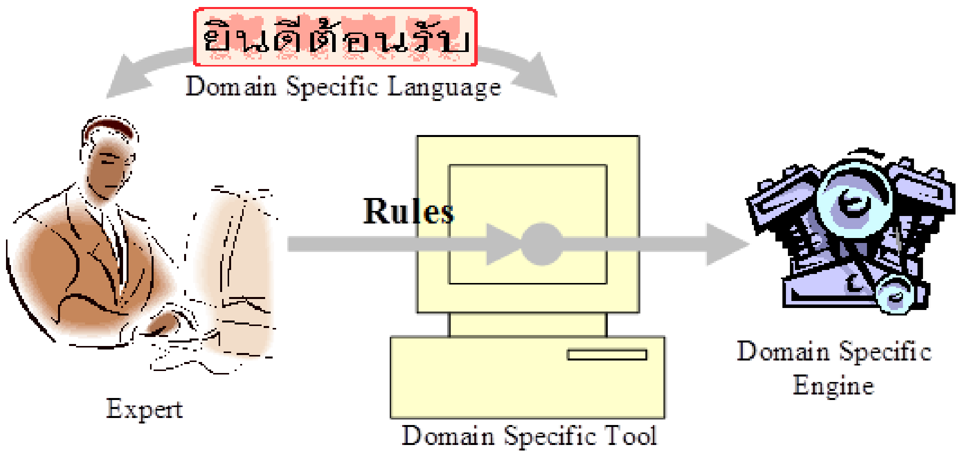


Figure 4.1: Domain specific, rule based system environment

1. Language definitions

2. Interpreters / compilers that will run the rules that will be written in these languages are needed.

Abilities That can Be Expected of the Language

In this part, some abilities that are considered to possibly be possessed by the language are presented under the light of already finished researches and continuing development activities in the industry. Issues of expectations on expression style and of what kind of temporal can be reflected to the language are dealt with. At current applications, some 'C' language alike rule structures have been found out. Studies have been made within the scope of possible fault management domain and also on temporal mechanisms that can be considered within the extent of this study in practicality. Some important ones of these mechanisms are reflected at this report.

Features that are related to the language's expression ability are presented below:

- Compatibility with the existing rule based structured syntax might be useful.
- Usage of regular expressions: This requirement has been defined especially to be able to state more than one unit with short expressions.
- High-level structure that can be useful in terms of ease of use.
- Reflection of a terminology that is suitable with fault management domain.

- Inclusion of suitable abilities and expression styles that exists at programming languages based on Event Algebra.

Abilities that the language can model:

- Ability to reflect recurrent, serial and temporal events.
- Ability to run commands with options that are either synchronous or asynchronous.
- Ability to run parallel events.

An example about synchronization options that are not deemed necessary to be developed for now because of the assumption that it is not currently needed much, is shown below. At this example, rule based language is assumed to also enable procedural blocks first and foremost for its practical benefits. In addition to that generally assumed synchronized mechanism, asynchronized functioning ability is also seen. Asynchronous mechanisms bring into force, in a way, parallel run ability. Functional blocks that will also be important in terms of efficiency can have place within rule based language. Usefulness of mentioned temporal abilities can be shown in a more meaningful way within blocks. The striking property at these structures is that more than one different rule (or rule element) can be used as a single entity. Another area of utilization for blocks is cases where various synchronization requirements will be used together. At following part, sequential usage of different rules for a correction process and asynchronous action groups that might be needed during running of these rules are shown. At this example it is shown that rules that require separate times from each other do not have tolerance for sequential running and it is preferable to start each one without waiting for the others to finish.

The language that will be developed and running/interpreting environment that should be considered along with it are seriously being considered in terms of efficiency. Examples and choices at the following parts are presented to examine the efficiency ability with regards to performance. Similarly, expression ability is given importance.

Most of the processes can be made with current grammar – for this, it is also possible to use a ’timer’. Temporal events can be coded when timer related variables also take place within cluster of facts. By registering events that are bound to time as facts , sequencing ability can also be fulfilled with standard rules. However, in these cases, a lot of compromise might have been given from understandability of expression and from running efficiency.

Example: A command block that includes synchronization

This example includes compound rules that should be examined together. Sequential rules, which are trying to make an inference in an efficient way while examining various possible reasons of a malfunction, constitute a block as a whole. Basically in this example, a sequence that is arranged with helps of 'if' and 'else' words and thus, a sequential process cluster is being represented. Only after the condition part of previous rule is evaluated then the next rule can be run. This functional block also includes interferences to the system. For example, by increasing fan speed we can observe a decrease in central processor's temperature. This block deals with different interference and observance processes that need to be examined together in order to fix a possible malfunctioning related to over heating of central processor. This block, which can be considered to have correction purpose, should be thought to have been activated as a result of a diagnosis on CPU temperature:

1. if fan is immobile then start fan
2. if fan is immobile then pass to another processor hardware, alert for repair
3. {Else} wait for 1 minute
4. if CPU heat is reduced, exit
5. {Else} if CPU frequency $>$ critical value then reduce CPU frequency, wait for 1 minute
6. if heat is reduced then, exit
7. {Else} if CPU voltage $>$ normal value then reduce CPU voltage, wait for 1 minute
8. if CPU heat is reduced, exit
9. {Else} if CPU load $>$ 90 % then stop some tasks, wait for 2 minutes

10. if CPU heat is reduced, exit
11. {Else} if CPU heat > Critical Value then unknown error, log, exit.

Sequential processes take place at above example. At an alternative approach to that, namely classical black board approach, a similar ability can be attained by setting facts against realized event sequences in a defined order. For example, if heat is not reduced within a minute after the fan is started, it is arrived to the conclusion that there is an error. If we want to test all of these events together at the condition part of a single rule, we have to be able to generate a fact that will represent fan is started and then a minute passed and heat was not reduced' events exactly in respective order. Perhaps sequential blocks are suitable to code this compound condition that also includes temporal in an easier and more efficient way. However, sequentiality in this example brings a danger with itself: worst case scenario can be that if a solution can not be reached at the first test conditions, requested intervals will overlap after each intervention and when 11th line is reached, 5 minutes will pass away. This duration will in turn cause the central processor to heat dangerously. Purpose of that example is to present modelling of temporal events to be run efficiently. It does not have an importance with regards to a real fault management scenario.

In order for above mentioned waits not to overlap with each other, interventions and the following waits can be run parallel. To achieve this, asynchronous commands can be used. At Table 4.1, for the purpose of an easier reading, command contents' different fields are arranged as columns and total waiting time has been shortened by applying previously mentioned temporal mechanism.

Asynchronous column carries the value of 'A' if command is asynchronous. These types of commands cause the next commands to start without them waiting for the previous ones' results. Otherwise commands are synchronous: they stall the next command until they are finished.

Intervention Ability

Example set at Table 4.1 also gives place to intervention events like 'speeding up fan' and 'decreasing CPU's voltage'. At the right side, which constitutes action part of rules, actions that would normally affect 'facts' can be found. Actions are usually

Table 4.1: An example of compound rule for fixing CPU overheat malfunction

Command number	Asynchronous	Condition	Condition True		Condition False	
			Action	Next stmt	Action	Next stmt
1	A	Fan_speed <THfs	Speed_up_fan	(After 1 min) 6	Message (Fan OK)	
2	A	Cpu_Voltage >V_critical	Decrease_Voltage	(After 1 min) 7	Message (Freq OK)	
3	A	Cpu_Frequency >F_critical	Decrease_Frequency	(After 1 min) 8		
4	A	Cpu_load >THld	Decrease_load	(After 2 min) 9	Message (CpuLoad OK)	10
5		Cpu_heat >84	Message (risky CPU heat)	Exit	Msg (corrected: fan speed)	Exit
6		Cpu_heat >84	Message (risky CPU heat)	Exit	Msg (corrected: Cpu Voltage)	Exit
7		Cpu_heat >84	Message (risky CPU heat)	Exit	Msg (corrected: Cpu Frequency)	Exit
8		Cpu_heat >84	Message (risky CPU heat)	Exit	Message (corrected: Cpu Load)	Exit
9		Cpu_heat >84	Message (risky CPU heat)	Exit	Message (intermittent CPU heat)	Exit

used as messages at this study. When we direct those messages to mechanisms that would affect controlled real world application, intervention fact does arise. Intervention ability that can be at the language brings to mind the need for sequential command clusters in terms of clarity. With this approach, some problems will be expressed in a more efficient and easier way for programmers. First step at adaptation of rule based systems to embedded applications is reactive systems. It is allowed to make changes mostly on facts with external inputs at reactive systems. On the other hand at interventionist systems, outputs that could affect those external inputs indirectly are produced. Although potentially a stronger and more complex model is indicated with interventionist systems notions, apart from providing an easier expression of this ability no other complexity is proposed as within the framework of this doctoral study solutions that are easy as much as possible and that would not lead to errors are sought. At Figure 4.2 the transference of standard rule based structure and its add-ons into a reactive and interventionist system is represented.

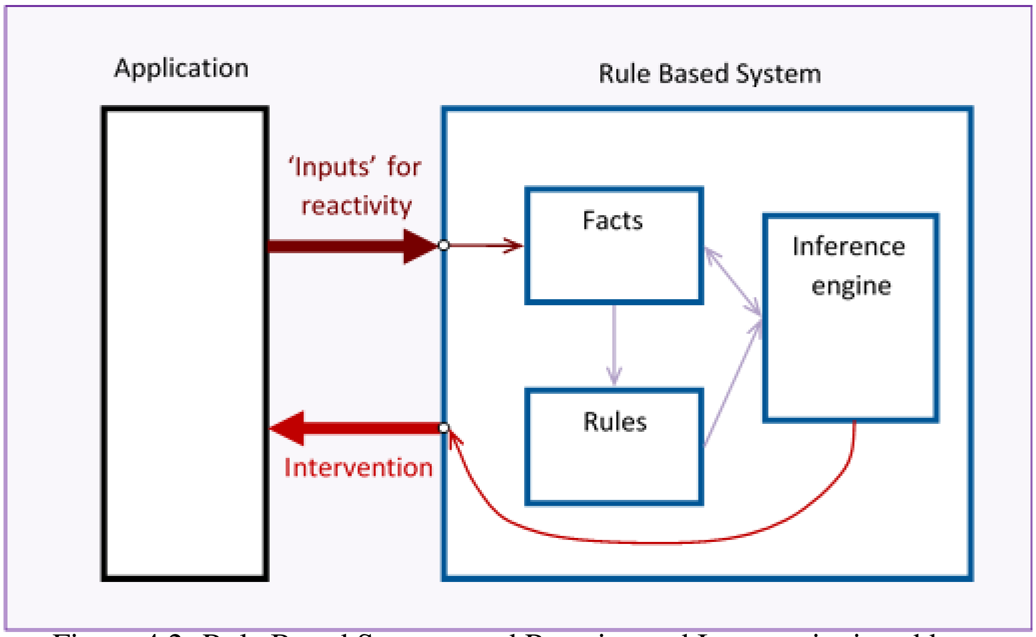


Figure 4.2: Rule Based Systems and Reactive and Interventionist add-ons

Temporal Expressions

Abilities of arrangement of events and existence of expressions about time at rules' conditions part are presented by languages based on temporal algebra. There are various language studies in literature that would meet those kind of requirements. When we take a look from the perspective of simplicity and running efficiency GEM language can be considered as close to this study's requirements. Also, we can see that many other studies take GEM as a reference. Abilities similar to planned temporal expressions take place at this language. Producing 'compound' events expressions by correlating events at GEM are summarized at Table 4.2. First two of these are not temporal.

Table 4.2: Compound event expressions at GEM language

Expression	Explanation
e1 & e2	If both events have been occurred then this 'compound' even has occurred. (AND)
e1 | e2	Development of any event is sufficient. (OR)
e + [10*sec]	After 10 seconds from e.
e1 ; e2	If both events have developed in order.
{ e1 ; e2} ! e3	If e1 ; e2 happened in order without e3 coming in between.

GEM is a language that can correlate events with their variable names, able to use delay times and exact time (ie. [10:30 27/6/95]) and it is rule based. Those kind of event expressions can be given place within rules. A rule's general structure is like: Rule rule-id [detection window] {event-expression action}

Detection windows that rules will be valid for can be defined. In this case, when it exceeds the defined time, rule is not taken into assessment. At the action part, again events can be produced and commands for existing rule's or other rules' activation or pacification can take place. At the GEM study, running environment mechanisms that would cancel the errors arising from delays in events' recognition are suggested. In this study, while benefiting from GEM language to its fullest, not implementing unnecessary abilities (for the sake of simplicity, efficiency and security) might be considered.

Examination on expression styles and implementation efficiency

In this part, coding alternatives that will be developed by different language structures will be assessed over a short part of the example that was set in Table 4.2. Assessment criteria will be easy to understand expression style and running efficiency. In summary, classical rule based approach that each and every rule condition will be assessed independently will be compared to approaches that will set restrictions on sequencing. In spite of its disadvantages like inefficiency and temporal ambiguity at running times, the classical approach is also known to have advantages like being widely used, tested and supported with tools.

Sample problem

In a scenario about corrections that will be made after CPU temperature diagnosis, if cooler fan is slow then its speed is increased, at the same time CPU clock speed is decreased and after a minute CPU temperature is measured again. The important thing here is that by doing two separate corrections in sequence, we must not spend two minutes instead of one. Solution options are coded, firstly with a standard rule based approach that does not have temporal abilities, then with a rule based approach that has a compatible temporal ability with GEM, and lastly with an approach that has sequential abilities.

1. **Standard Rule Based approach**

   We can assume that a message coming from a timer will be added as a fact to the system, in order for a one minute delay's being considered as a fact at coding that will be done with this approach. To model the sequence between two events, firstly we can discern the condition that a fact is present related with the first event but it is absent related with the second, and then we code the information that first event took place before the second. At a certain start time we must also assume that none of those facts are yet in a state that they are 'true'. Lastly, we must remember that control outputs like starting the timer or increasing speed of fan at action phase of rules can be activated. Different timers must be able to be named related to different events and also they must be accessible.

   //——— Correction pursuit for the state of fan slowness: ——————————

   Rule 1 : Condition Fan_Slow & ! (Time1_over) , Action fact : Fan_accelerated; Time1_start (1 minute)

   Rule 2 : Condition Fan_accelerated & Time1_over & Temperature_High , Action message 'CPU temperature high'; stop_system

Rule 3 : Condition Fan_accelerated & Time1_over & Temperature_Low , Action message 'CPU temperature is decreased by accelerating fan speed'

//——— similar approach is repeated for CPU clock: ——————————
– Rule 4 : Condition CPU_fast & ! (Time2_over) , Action fact : Clock_decelerated; Time2_start (1 minute)

Rule 5 : Condition Clock_decelerated & Time2_over & Temperature_High , Action message 'CPU temperature high'; stop_system

Rule 6 : Condition Clock_decelerated & Time2_over & Temperature_Low , Action message 'CPU temperature is decreased by decelerating CPU clock'

At the first rule at this coding, in order to be able to recognize the state that fan slowness event took place prior to clock timer event, the truth that first event happened but the second has not happened yet is added to facts list as a fact. By using that fact at the second or third rules, condition that the sequence is finished is looked upon. 'Resetting' of facts is an issue to be dealt with. Some language rules provides erasure of all 'events' that are used at when action part is fired. In this example, that kind of details are not integrated. Moreover, in order to keep the example simple, condition elements like Temperature_High and Temperature_Low are supposed to be measured anew when the rule is being run and they are equivalent of their values at that time. On the other hand, an example that would produce a measurement request when it is required will be more realistic.

Another interesting issue that can be seen at this example is that the second and the third (similarly at 5th and 6th rules) condition's first two elements are the same. Only, different actions are triggered for two states of the last element. Different rules can be triggered, in conformity also with RETE philosophy with respect to efficiency, by looking only at parameters related to temperature and creating branches for different values after happened events bring the central state model of 'states' at conditions of different rules to a joint state. At standard running environments, each rule's condition forms a separate evaluation tree and all elements have their places at each tree even if these elements are repeated. This causes inefficiency.

When these rules are run, two timers are started at the same time effectively. In fact, this problem can be solved with only one timer.

2. **Temporal Rule Based Approach**

   At the same example that is coded with a syntax compatible with GEM language, without the need for producing additional ’facts’ for sequencing but still producing repeatations at conditions, a solution is given:

   //——— Correction pursuit for the state of fan slowness: ————————— Rule G2 : Fan_Slow +[1*min] ; Temperature_High $\rightarrow$ notify ’CPU temperature high’; stop_system

   Rule G3 : Fan_Slow +[1*min] ; Temperature_Low $\rightarrow$ notify ’CPU temperature is decreased by accelerating fan speed’

   //——— similar approach is repeated for CPU clock: ———————— - Rule G5 : CPU_fast +[1*min] ; Temperature_High $\rightarrow$ notify ’CPU temperature high’; stop_system

   Rule G6 : CPU_fast +[1*min] ; Temperature_Low $\rightarrow$ notify ’CPU temperature is decreased by decelerating CPU clock’

3. **Asynchronous Sequential Command Blocks**

At above examples, rule based approaches’ constituting a condition of difficulty or inefficiency for sequential processes to occur as a result of in-sequential run structures have been examined. In this part, hybrid approaches that are oriented at coding of some critical parts with a sequential structure are covered. The first approach was shown at Table 1 previously. A short example that is used for other approaches for comparison purposes instead of example at Table 1 is coded with table approach at Table 3. This approach fulfils sequencing function by writing next command’s number. Depending on condition’s true or false results, different sequences can be provided. Thus, inefficiency of ’sub condition expressions’ separate evaluation’ that takes place at previous methods can be prevented. However, some abilities are unavailable at a complete sequential approach: At this approach a middle ground has been achieved by allowing some commands to be asynchronous.

Time delay mechanism at Table 4.3 differs from the approach used at Table 4.2. At Table 4.2, delay requests were placed at ’next command’ field. At this table, time delay is expressed as an event among other events by taking its place at action in accordance with GEM language. As events can be produced as actions (can even be compound) at GEM language, this kind of change seems appropriate.

Table 4.3: Table approach that contains asynchronous commands and is GEM compatible

Command #	Asynchronous *	Condition	Condition true		Condition false	
			Action	Next command	Action	Next command
T2	A	Pervane_slow	Fan_accelerate; [1*min]	T3		T4
T4	A	CPU_fast	Clock_decelerate; [1*min]	T5		Exit block
T3		CPU_heat_High		Exit block	Notify 'fan is accelerated'	Exit block
T5		CPU_heat_High	Notify 'CPU heat is high'; stop system	Exit block	Notify 'CPU clock is decelerated'	Exit block

At this approach, data structure complexity at a rule based system's run time has been reduced. Run model is tried to be deduced to central structure as in C alike sequential languages. However, because of inclusion of asynchronous rules, a parallel run mechanism is needed to some extent: such mechanisms are kept at minimum. When we look at the solution at Table 4.3, we can see existence of two blocks that are actually started with T2 and T4. After started asynchronously, these blocks continue sequentially within themselves. A multi tasked run is possible. In fact that problem can be coded as two parallel blocks: in order for the first two conditions to be tested, a sequence is not needed, they can be tested at the same time, and thus we get two parallel blocks. This table appears as coded in parallel blocks at the next approach.

Parallel Blocks

At this approach, as it was at Table management, it is tried to give weight to sequential mechanisms as much as possible where as multi-tasking of rule based systems is tried to be kept at minimum. Instead of sequential blocks started by asynchronous commands, we have parallel started sequential blocks. In most cases, two approaches can be used to give the same result. At below example, same problem is organized as parallel blocks.

```
ParallelBlock {
    If Fan_Slow then
Accelerate fan;
[1*min];
If CPU_temperature_high then { notify ’CPU temperature is high’ ; stop system }
    Else notify ’fan accelerated’;   }
ParallelBlock {
    If CPU_clock_fast then {
CPU_clock_decelerate;
[1*min];
If CPU_temperature_high then { notify ’CPU temperature is high’ ; stop system }
    Else notify ’CPU clock is decelerated;
```

Whether it is Table approach or parallel blocks approach, instead of using two timers, it is possible to change codes in a way that would allow us to use a single timer. Effort of reducing timer is in fact a reflection of the effort that is being made to decrease repetitive structures. A similar efficiency gain exists at the philosophy of Rete algorithm that is occasionally used for standard rule based systems’ application. At this algorithm, processes that will be utilized parallel although they correspond to same events are represented at a common state flow. Parallelism is allowed only at cases where there are differentials.

Rete algorithm requires a not so easy run environment. In general, it increases efficiency at rule based systems. However, in perspective of this study, existence of lots of functionality complexity must be examined for justify its complete application. Alternatives that have been discussed until now, in a sense, creates an optimization environment that can be interpreted as reducing complexity load and relay it to the programmer through Rete application.

At following parts, after similar examinations with regards to application are made, approaches that will be proposed for the study are being discussed.

Examination of application mechanisms

A standard rule based system can be run by building an evaluation tree for each rule's condition side and as events progress at run time, by making evaluations at relevant places within those trees. Condition side of a tree that has completely passed evaluation is considered to be fulfilled and action side can be triggered. Proposed approach is interventionist. While conditions are examined, some events, like accelerating fan, are triggered. This kind of interventions can have a place at condition evaluation trees without causing too much change basically. While evaluation is going on, some outputs are also triggered at some points. This information on trigger can either be shown on trees or it can be held separately and then associated with tree if necessary. At Figure 4.3 condition evaluation trees for Rules that are used at temporal rule based approach example are shown.

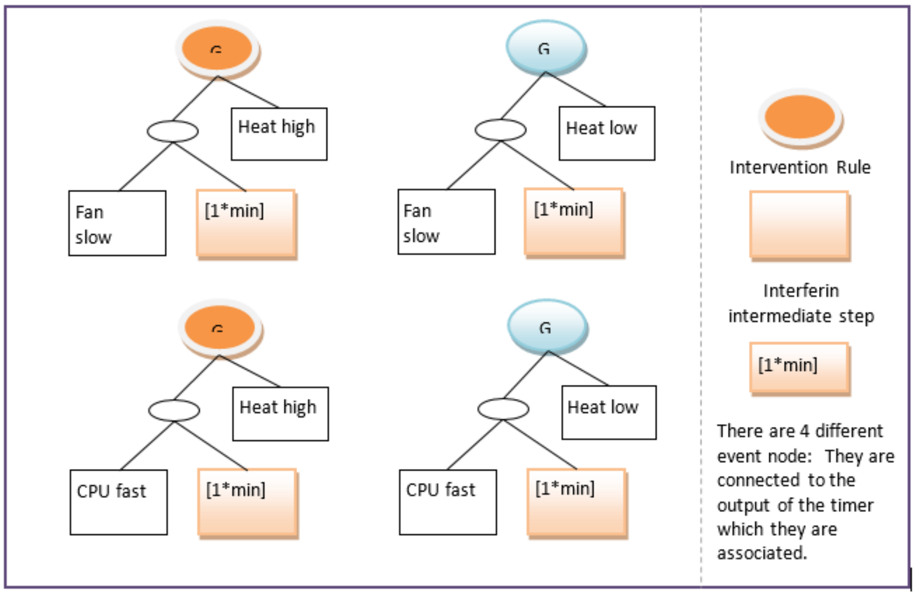

G
Heat high
Fan slow
[1*min]
G
Heat low
Fan slow
[1*min]
G
Heat high
CPU fast
[1*min]
G
Heat low
CPU fast
[1*min]
Intervention Rule
Interferin intermediate step
[1*min]
There are 4 different event node: They are connected to the output of the timer which they are associated.

Figure 4.3: Condition evaluation trees

At run time, it is necessary to evaluate the trees at Figure 4.3 from left to right. For example, if Fan is Slow fact is present, bottom left leaves of G2 and G3 trees will be realized and they will give permission for their right side neighbors to be evaluated. Later when one minute delay event happens, ([1*min]) not is realized. After that, turn will come to realization of 'Temperature is high' ('Temperature low' at G3 tree) knot. Thus, only if events that are supposed to happen sequentially do happen in that order then condition of the rule at the root (i.e. G2) will be validated and action will be enabled.

When the tree at Figure 4.3 is examined, we can see that they have a lot of similarities. A general running environment must look into every activated not at each tree at each fact change (or when a message is received) by scanning all rule trees. However, if common parts of trees are transferred to a central structure, roams over this structure will cancel the repetition complications.

Proposed approach

With a short definition, the development will be language-centered but also will be mechanism centered that will not necessitate high inference ability and performance and levels that are close to hardware will also be in consideration. It will be necessary to design both structure of language that will define rules and the environment that compounds defined with this language will be ran. At this part, preliminary design principles on firstly the language and then the running environment are given consideration.

Rule definition language

During development of rule defining language, first of all, structure of the language must be defined and then value types of this structure's elements must be classified and lastly, if possible, all of these values must be sequenced. Left side (condition) and right side (event) elements must be present at rule expressions. The state that system's history brings about must also be represented at conditions. State within framework of reference architecture will be depending on data like installation information and composition of facts that comes about as a result of received messages. It will be useful to spare separate specification fields for information classes like these. For some cases, it will be sufficient to send messages at actions whereas for other cases, actions that intervene target system will take place. It can be assumed that this kind of interventions can be made by sending messages. However, it might be useful to specify different types of processes separately with regards to a semantic angle (even if they all can be realized by sending messages). For example, it might be appropriate to event types like intervention requests, notification requests and reporting requests as different expression types.

What is proposed for this project is arrangement of action side as generally working with messaging. This approach is also coherent with its aspect at actual study. Semantic differences can be expressed by tagging messages for different kinds of actions – for example, a symbol that is related the type of action can inserted at the beginning of each message tag. Furthermore, unit that will receive message will also be informative on the type of work that is requested. It will be useful for expressions to have an element that is allocated to the receiver of message.

When application environment is considered, issue of not inhibiting performance will be crucial. Rules must evaluate some states that might be complex during conditions are formed: these conditions also include sequential, repetitive and temporal relations during events. Because each rule is evaluated independent of one another and no sequential run is considered among rules at commonly used rule based systems, this kind of conditions can be realized by indirectly forming of complex facts. It will be useful to take this kind of needs in a more practical way within scope of this study. Characteristics of language that are given below are intended for expressing sequenced, repetitive and temporal conditions in a more practical way.

Condition Definitions

For the purpose of easing the expression of complex conditions, conditions must be taggable and these tags must be able to be used in forming more complex conditions. Examples for tagging definitions:

```
CONDITION DEFINITIONS {

A: token (source (1), 4) & 0x03 != 0x00

B: token (source (1,3), 6 ) > 60

}
```

Pattern Definitions Patterns represent compound conditions. Basic structure at patterns is sequenced conditions. Repetitive conditions can be viewed as a special state of sequenced conditions. Also, temporal restrictions can be placed by a sequence. Pattern samples that contain simple sequences are shown below:

E1: A ; B ; A | C ; D

This pattern corresponds to the sequence that after condition A happens then condition B, later conditions A or C and lastly condition D happens in exact order.

E2: A ; A ; A

E2 sample represents a repetitive condition. Same condition can also be shown as A3.

Two basic mechanisms are considered for temporal expressions for now: time needed in between two conditions, and the requirement that a condition must be executed within a certain time. At those kinds of expressions, pattern can be used instead of condition. Examples for temporal expressions:

A ; 25mS ; B

At this pattern it is expressed that after condition A happens, 23 milliseconds must pass and then condition B happens. If needed, it is possible to define that time as at most: 'If condition B happened within 25 mS at most after condition A happened' conditions can be expressed as shown below:

(A ; B) ! 25mS

A ; 25mS ; B

At these expressions, syntax of GEM [75] language is used. The meaning of the first expression is: During conditions A and B are happening, 25 milliseconds timer's 'time over' message should not be seen

At this approach, exact time durations like '25 mS' are defined as receipt of a message that a timer sends when its time is up. This timer is started after when previous condition is perceived. When the second expression is interpreted in this way, the expectation that 25 milliseconds will pass after condition A and then condition B will take place.

When the second expression is interpreted as such, it is expected that 25 milliseconds will pass after condition A and then condition B will happen. The time that passes after 25 milliseconds is not important. Furthermore at GEM language 'guard' expressions can be added to conditions by using attributes of events. Even though similar specifications can also be made by using events' attributes and time stamps, these features are not considered for this study for now. GEM uses operators of |@ for starting moments, and @ for ending of events. It is possible to show the same expressions within a spotter at GEM approach as shown below, because it enables facilitation of more general time variables independent of timer analogy:

A ; B [when @B - @A < [25 * mSec]] or A ; [25 * mSec] ; B

A ; B [when @B - @A < [25 * mSec]]

In fact, a single even''s time is shown with @ operator. It possible to talk about starting and finishing moments of a compound event. If we were to call pattern A ; B as P3:

P3 : A ; B

The second expression that corresponds to total time's being less than 25 milliseconds can also be expressed in a different way:

P3 [when @P3 - |@P3 < [25 * mSec]]

After conditions are defined as simple or pattern, these conditions can be used to form left side of rules. At below examples, actions to be taken as a result of formed conditions are being expressed as error statements:

A $\rightarrow$ error (source (1), 2)

P2 $\rightarrow$ error (source (2), 3)

Expression method: Conditions that patterns involve time constraints are shown above. Because a high level abstraction ability is targeted, comprehensibility of temporal expressions also must be sought. Basically, two different usages have been observed (infix and postfix) depending on choice whether time expression will be expressed at the interim or at the and of other events. Temporal constraint is shown in a more visual way at infix expressions (ie. A; 25mS; as in pattern B). On the other hand, a time constraint that is needed for a whole pattern can be expressed more comprehensive at postfix expressions (ie. (A ; A ; A) < 25 mS). In fact, it is possible to use a single expression style but it is needed more to use the suitable style at complex patterns. For example, if only the postfix style will be used, expression of sequenced events becomes more complex. We can look at infix and postfix styles of the same expression to have an idea – at this example, the sequence that is formed by conditions A, B and C is expected to be with 1 and 2 mS delays in between:

Infix: A ; < 1 mS ; B ; 2mS ; C

Postfix: (((A ; B) < 1 mS) ; C) 2mS

It seems that postfix is a more suitable expression for temporal constraint conditions that target entirety of a sequence where as infix is suited best for all other remaining sequential operations. It can be said that also at GEM language this approach is being used: because a temporal constraint for entirety of expression is assumes a duty like spotter expressions, it is written after the expression as it is happens at spotters (ie. (A;B)!C). Decision of using either both of expression styles (infix and postfix) or choosing one of them can be made by a choice depending on expression clarity and optimization of parameters on difficulty of processing the language.

Language design procedure:

1. Rule structure must be designed: different information types that can be organized as sub units of condition and action parts must be defined.

2. Actual grammars must be expanded with specifications that are related with temporal expressions and regular expressions.

3. It must be decided that how the status info will be managed: rules look whether necessary condition for the event is realized or not usually by establishing relations on facts that are kept on a blackboard. Also, actions can make changes at status info. At Rete approach, state model develops as facts occur and action triggering ability can be acquired without the need of making much more evaluations at reached 'state'. It is suggested to use Rete logic as much as possible at the actual project. Reason for this is to have calculations that are easier to time

and work at a shorter span. In return, development cost will increase. Also, even though memory need will increase, it is not expected to pose a problem as not much state space have been observed at problems, yet.

4. Interfaces that rule based sub system will be in iteration must be defined (ie. console, and different message layers).

5. Possible values for each sub unit of condition and action parts must be defined in full as much as possible. Sub units and values that they can take can be expressed as additional constraints at syntax. However, it must be kept in mind that these can change through time.

6. Meaningful sub clusters must be presented by grouping values if necessary and tags/key words must be found that would correspond to these meanings.

7. If there are values that can not be used together with other values, they should be identified and documented. This information will be used for generating error message or directing user in case of a faulty rule entry.

It will be useful to create effective data bases would be used at all of these steps that realistic information could be gathered from this study and used at decisions to be taken.

Running Environment:

1. By keeping in mind that at sub units, especially like spotters, messages will be received frequently, it can be expected that need for a lot of inference performance that this kind of inputs will trigger. In order to increase efficiency at inferences, structures like document trees must be considered. It can be expected to design structures that will provide this kind of state model to be incremental as constants. Also if possible, developing the ability for these structures to adapt themselves automatically must be considered. Within the scope of this project, constant document trees must be sought as much as possible.

2. By taking into consideration different protocols, definitions of addressing mechanisms at messages must be given. For example, it might be necessary to route messages to target unit via TCP/IP or 1553 protocols.

3. Choices must be made by comparing 'state management' choices while taking rule clusters into consideration:

    - Facts are kept in a black board structure at a standard rule based orientation and each inference is capable of using all of the facts. Rule that is trying to find out whether required condition is fulfilled or not by using inference necessitates an inference that can also be explained as a search for information that it is interested in by looking at history and finding out whether the desired state has been reached or not. As a result, work load per inference is high.
    - If the fact space is small, states that fact cluster will imply with a state machine model can be modelled. With each entry, system makes a constrained state change under the light of information that is formed by rule cluster at hand. On the other hand, action part is activated when this state machine triggers an output. At this option, process that is required for this inference is reduced to a proportion that can be neglected.
    - Another state based presentation is formed by document tree model. At novel rule based approaches (i.e.. at Rete algorithm usage) this model is valid. Again inference is pretty much simple and development between facts (rule based history based) is kept at a tree structure. Difference between document tree and state machine is that a state can be reached by different ways or a single one. For example, if a certain amount of money has been dropped into a vending machine, information of in which order those coins were received and the total amount was achieved is irrelevant. On the other hand, words' entry order is important at a word crawler. Document tree presents a state model that is dependent on events' order of occurrence.

4. Running expressions at rule language faster by compiling them and with an option of machine language must be considered when it is required for performance. Normally, proposed method depends on the principle of rules' being interpreted instantly. At current state, a need for compilation is not observed.

5. In order to prevent contradictory rules, some validation ability would be useful. The most important contradiction occurs when they are at the same state, more

than one rule's conditions are triggered. Start of contradicting action parts of rules that will be initiated simultaneously is the kind of situation that is wanted to be prevented. Even though it can be observed with naked eye what the right sides of simultaneously triggered rules are going to do, still it would be useful to automatically check whether differently expressed left sides have the same meanings. In order to gain this ability, it is planned to transform conditions into a normal form. This way, similarity check can be made between normal forms.

6. A simple way must be followed as much as possible for conflict resolution. Standard rule based inference engines take into consideration possible conflicts between rules when they decide to choose which one of them for triggering. At this study, it will reduce run time ambiguities to create the environment which would cause the least conflicts as much as possible. It will be useful to take precautions to prevent conflict resolution if possible.
7. User interface design: preparation of related screen designs for conditions where user interactivity is required might be considered.
8. A tool with visual support will be useful at development of rule base. Development of such a tool that could be used at entry of rules might be considered.

4.2 Checkpoint Ability Development Approach

There will be different installations and requirements at Fault Management applications. Checkpoint (CP) ability, on the other hand, necessitates a capacity at process power and transmission channels. or an efficient and secure functioning, it will require the utmost care at optimization of design on that ability. Foremost parameters are outlined below:

1. At which points and in what frequency application necessitates a state save.
2. Where will the states be stored: like at the related unit, on the outside, or at specific backup point. Factors like proximity will be affective. Also, it will be important whether physical memory that state is going to be saved will be temporary or permanent.

3. Usage of precautions like incremental state save, saving of compressed data, and narrowing the memory down.
4. Recovery cost and management.
5. Fault management units
6. Evaluation of checkpoint abilities for also fault management units' themselves.

Suggestions on work to be done

1. First of all, units that checkpoint is going to be applied should be identified.
   (a) Installation of checkpoint approach must be designed for identified units.
   (b) Relationship between redundancy management and checkpoint approach must be presented.
2. These processes must be supported by tools if possible.
3. State storage policies must be defined and suggestions on their application must be presented.
4. As it is in rule based approach, checkpoint policies must be applicable without much special programs by a high level method.

While checkpoint approaches are being developed, their convenience with other components of fault management architecture must be sought.

General life cycle expectancies on checkpoint

1. As stated previously, to determine for checkpoint backup at which points and how often it should be.
2. To decide on recovery and rollback policies on the whole system or different policies on different units if possible.
   (a) Coordinated / Uncoordinated / Message-driven method choosing: Uncoordinated method should be avoided in spite of the forecast that there won't be so much state information emerging within this project. Because

saving the whole history would cause memory usage which will be hard to estimate the amounts, and also the complication of rollback algorithms and real time executions will spend times which will be hard to estimate as well.

(b) Determination of separate sub-policies for the Backup and Restore procedures

Above mentioned support tools can be used to prevent errors during rule typing and making it easier to type without the need of syntax knowledge. For example, typing of rule can be made easier by making the fields that the rule requires table based and instructive for user and by providing possible choices for necessary data. Required fields and thus the structure of table will change depending on the type of rule.

Evaluation of Method Options While deciding which methods are going to be used, it is crucial to clearly specify not only constraints of environment in which check point and recovery from error mechanisms will be used but also the requirements. If check point approach is going to be used as a recovery from error method like it was mentioned above in detail, the choice of which one of the methods below will be used must be based on requirements.

- Synchronous
- Asynchronous and
- Communication-driven (Hybrid method)

The most important constraint for the method that is going to be chosen is that it must be acceptable for a real time system. Advantages and disadvantages of synchronous and asynchronous methods must be evaluated and an appropriate method must be chosen depending on actual and possible constraints. These constraints and requirements will be used at specification of mechanisms that are going to be used at overcoming difficulties like memory and domain management, process migration, domino effect and orphan message that occurs at checkpoint applications along with the specification of method that will be used.

When literature survey is examined, it is observed that checkpoint and recovery applications might be problematic for real time systems. Basic causes for this are costs, difficulties for processes to fulfil their delivery times and the effect of this delay on time sensitive data that will be sent to outside world. Approaches for this kind of real time systems have been usually concentrated on Virtual Machine notions. Details are isolated from user code by developing Checkpoint and Recovery from Error applications at Virtual Machine level.

Real time systems bring about absolute constraints because of their time-critical nature. These constraints are:

- Consumption of resources
- Overload and
- Delay

When these constraints are considered, checkpoint process must be finished within a specified limited time. A research that was taken from literature has been able to make processes reach their delivery times by using below methods [25].

Checkpoint process must not come in between threads that are called as mutators and facilitate user codes. Checkpointing applications that have been designed for real time systems generally use copy-on-write method while recording instantaneous image of active memory. The simplest method that is used at checkpoint's general applications is to stop all working mutators, save necessary state data (a memory area) and start mutators again. This approach is not really suitable for real time systems. The most important reason for that is the fact that copying time from memory to memory is longer than anticipated. This, in turn, affects delivery times of processes and causes delays. One of the methods that is used to decrease copying time is to do this process simultaneously.

Copy-on-Write: When checkpoint request is received, related (copied) memory area is protected by using Memory Management Unit (MMU). Working mutators are stopped immediately and a thread that is working in the background saves those write protected pages. The worst case scenario for this process is as shown below.

Copy-on-write approach is considered to be an appropriate method for real time checkpoint applications. The basis that supporters of this view show as proof is the fact that mutators are stopped only for the duration required to copy memory area magnitude (a few kilobytes decided by MMU). Whereas this duration might be acceptable for some applications, it is not really acceptable for real time systems. It can affect validity of system.

Two main methods are being used at real time embedded systems:

- Off-line
- On-line

At off-line checkpoint applications, check point acquisition period for a process is predefined. The most common checkpoint applications at literature for real time systems belong to that category. The most crucial disadvantage of these methods is that they are not adaptive to faults that might occur during run time of checkpoint acquisition.

On the other hand, on-line check point applications are more suitable for the task because they are more adaptive to errors that can occur during check point acquisition period.

Conclusion Remarks

As a result, in the light of research done and observations, we can define Checkpoint design approach that will be used at this study as shown below: A Hybrid Checkpoint approach that takes communications between RTPs instead of tasks seems to be more viable and useful for our purpose. Thus, advantages of both synchronous and asynchronous checkpoint applications will be benefited from.

Systems that are designed in a ’hybrid’ structure will be supported with both user level and core level verifications. However, we gave more weight to core-level since fault management can easily access the required information and also it takes advantage of checkpoint applications that are already available in the system. We know that this approach can cause loss of a little modularity but will bring maximum autonomy and manageability to the fault management.

Checkpoint and fault recovery applications are designed off-line since almost all of the processes are known beforehand and static. Thus, all processes' checkpoint acquisition periods is decided by using dependency relations with each other still when processes were being defined. However, for more dynamic systems, coordinator process might help each task at instant checkpoint acquisition, depending on status info that will be sent by other tasks.

At checkpoint approach, there would be no need for a process migration mechanism but critical issues like memory and domain management must also be kept in mind. As abundance of received checkpoints will increase the size of stored data, size reduction and memory narrowing methods can be used. Incremental checkpoint shows page-based read-only memory narrowing by using virtual memory protection hardware. While the program is running, check point mechanism sends a list of all pages that were modified since the last check point. If operating system supports, list can be managed with virtual memory primitives at user level. For example at Unix systems, all pages are set to read-only state beginning with following check point by using mprotect() function and then it adds a page to the page that was modified to catch SEGV signal. At the same time page is set to read-write. When time for check point arrives, only pages at the list are stored because remaining pages consist of only read-only variables. If spotting the modification locales at application program is easy, incremental checkpoint approach can result at a considerable reduction at size and load of checkpoint. In cases where there is no scarcity of storage area, it might be more appropriate not to use size reduction and memory narrowing techniques by taking performance and speed factors into consideration.

Technique of hiding delay might be used to ensure Checkpoint optimization (speed and performance). Hiding the delay techniques are 'copy-on-write' and 'main memory check point'. In cases where there is no memory problem, Forked-Diskless checkpoint approach [92] stands out as the most applicable method. At this method, either process state is written over the copy or copy assumes role of process application. It is considered to be more appropriate for copy to assume the role of application. In other words, applications (processes) whose checkpoints will be saved will copy themselves at designated checkpoint save periods.

At the designed checkpoint application, decision of in which order restarting will be made might be taken into consideration by preparing a dependency graph of task or RTPs. Along with this, as there is a priority based process structure at VxWorks or other operating systems, it might be necessary to take this priority structure into consideration during preparation of dependency graph.

At this stage, the problems of how RTPs or tasks status infos will be stored and how these infos will be restored during recovery have been examined by looking at VxWorks system calls and command help pages (VxWorks man1 and man2 pages), going through lower levels. Commands and system calls that can be related with these are prepared. On the other hand, an evaluation process on how theses commands and system calls can be used at design and development periods is still ongoing. Ways and methods to store RTP's state with actually ready commands are being examined. As said before, because status info of any RTP process is going to be copied (forked-diskless) as a block (copying the process), work is being done on commands and methods that will either copy RTP directly or RTP will copy itself.

As the Hybrid (synchronous and asynchronous) method is going to be adapted at designed checkpoint method, development of a new RTP process, which will be only used at management of check point, as a coordinator process seems to be applicable. By this, coordinator process will have the capability to manage the process with help of methods and algorithms whose details were given above. Other processes will keep informations of in which periods and by using which algorithms they will be getting information within themselves. Along with this, restarting process will happen in same direction and coordinator RTP will manage this by using recovery algorithm that has been presented above.

Observance points of fault management architecture are considered to be a part of fault management system although they might be within the application. Similarly, code parts that will be within the application might be needed also for check point application. For example, 'state transmission' services that will transmit only needed status info to each unit when they send a request might be needed to be developed.

4.3 Checkpoint Approach for Fault Management Infrastructure: Design

At this section, design decisions and design models that have been derived from results of study of method on how to act to give this study Rule Based Usage and Checkpoint abilities are presented.

4.3.1 Checkpoint Design Decisions

Depending on the results of checkpoint literature survey and method setting reports, need to adapt a design approach that can be applied within scope of this project and that will also be generic. As the application environment will be VxWorks operating system based, a design that will also support internal structure of this system is needed. VxWorks features Real Time Process (RTP) structures and under these, tasks at more detailed levels. Even though researched made concludes that an RTP based checkpoint is not very feasible for this study, to construct a design model that will support both RTP and task based checkpoint approach is decided. Accordingly, design has been directed by taking into consideration the checkpoint approaches that were attained at previous studies and during method assessment stage.

Checkpoint and recovery application that will be developed are designed as off-line because they are considered to be pretty much static applications in which almost all processes are known and because they will be ran on real time embedded systems. In this way, checkpoint acquisition periods of all process could be specified while they are being defined and making use of their dependency relations with other processes. Because, this structure seems to be more appropriate if tasks that are developed with relation to the study and RTPs (all processes that are running on system) that are created by these tasks are known before-hand. However, a design approach that covers both on-line and off-line functions can be adopted as a generalizing design approach is being considered. This means inclusion of abilities of either predefinition or dynamically specification during run time of checkpoint acquisition period.

Requirement study has been supported with graphic models and for this, UML compatible environments have been preferred. A use case diagram that explains main abilities have been developed for targeted check point infrastructure. This use case diagram has been formed during requirement analysis studies and it is also used to give direction to these studies. Figure 4.4 shows this use case diagram. Scenarios that are again supported by graphic methods and explanations for necessary use cases at this diagram are presented later on.

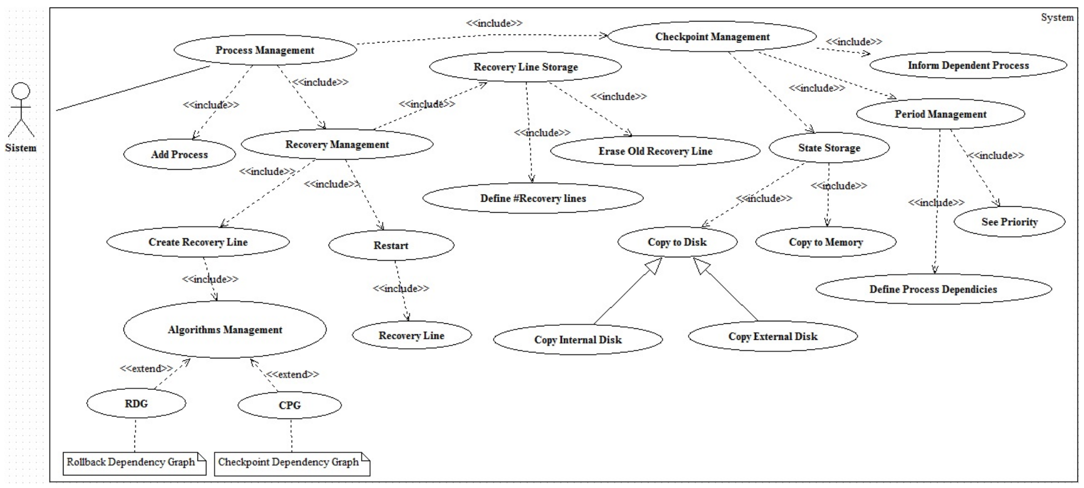


Figure 4.4: Combining the events at assessment trees

Typical event flow at Reference Architecture for this study is from sign tracing to detection and then to diagnosis. Because recovery process is considered as the first option for diagnosed faults, recovery process will be initiated in line with information received from diagnosis unit. For errors that cannot be recovered, again, recovery request will be made as defined at guiding architecture. Source of the fault and fault information that are specified by diagnosis unit will be sent to Recovery unit at check point. Recovery process will be achieved by restoration of faulty unit from appropriate check point (calculated from recovery line) and by sub process of restarting which is the last step of recovery process, depending on type of the fault and related unit. At the remaining part of this part, state uses' details are explained in detail.

4.3.2 Use Case Explanations

At this part, use states are explained with the help of sequence diagrams. For some use states, more than one scenario must be examined and thus more than one sequence diagram have been used. Use states are taken from Figure 4.4 and they have been explained one by one.

4.3.2.1 Process Management

Process management center includes a couple of functions that will be carried out by a process. Terms Related to Use Case: Process management consists of two functions:

- Checkpoint Management
- Recovery

Checkpoint Management: Checkpoint management consists of 'Save State' and 'Define Period' sub units. Processes will have an autonomous structure that will make them capable of getting their own checkpoints when they are most available. However, it is necessary to define dependency of processes (message exchanges) to each other in order to achieve synchronization between processes and to prevent negative effects like domino effect during recovery. For this purpose, Define Period unit will both define dependency between processes and it will define the most suitable save period for each process by looking at their priorities, if there is any. Thus, when a save state period for a process is defined, it will be necessary to look at both process priority and process dependency. Process priority is expressed between 0 and 255 at VxWorks operating system and while 0 is corresponding to highest priority, 255 correspond to process with lowest priority. If priority among processes will be used at applications, this priority must be taken into consideration while defining checkpoint acquisition periods. As dependency between processes is going to change from application to application, a process priority graph will be prepared for each application and checkpoint acquisition periods will be defined by depending on these graphs. If process priority will not be used at applications, only dependency info of processes will be enough in defining the period.

A process that will acquire a checkpoint to save its state will get a checkpoint while taking this dependency and priority info into consideration and after each checkpoint acquisition operation, it will send this info as a header info at application message to processes that it has dependency. Thus, process that has received this info will acquire its own checkpoint before communicating with other processes and then it will send a message. So, because there will be no orphan messages during recovery, it will be successful.

At Figure 4.5, sequence diagram for the operation of notification of dependent processes by process that has taken checkpoint and at Figure 4.6, sequence diagram of checkpoint acquisition process scenario are given.

Checkpoint acquisition procedure will be carried out in conformity with below order. However, a revision of this approach is currently under debate: The answer to the question of whether a process will be in charge of calculating its own checkpoint acquisition period or will it be handled by central process is postponed for now for assessment of other alternatives. This responsibility is given to the central process at the actual design.

1. Each process calls checkpoint building function to save its own state at a t time when it is available. This function sends the request to get a CP to central process (checkpoint manager).
2. Central process, after receiving the request, parallel calculates both process dependency and process priority in order to determine suitable checkpoint period for the process.

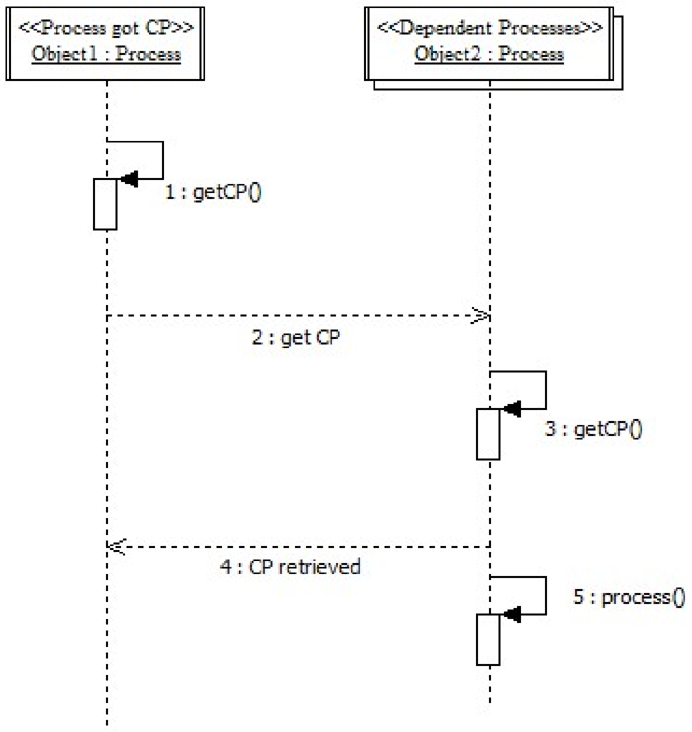


Figure 4.5: Notification of dependent processes by process that has taken CP

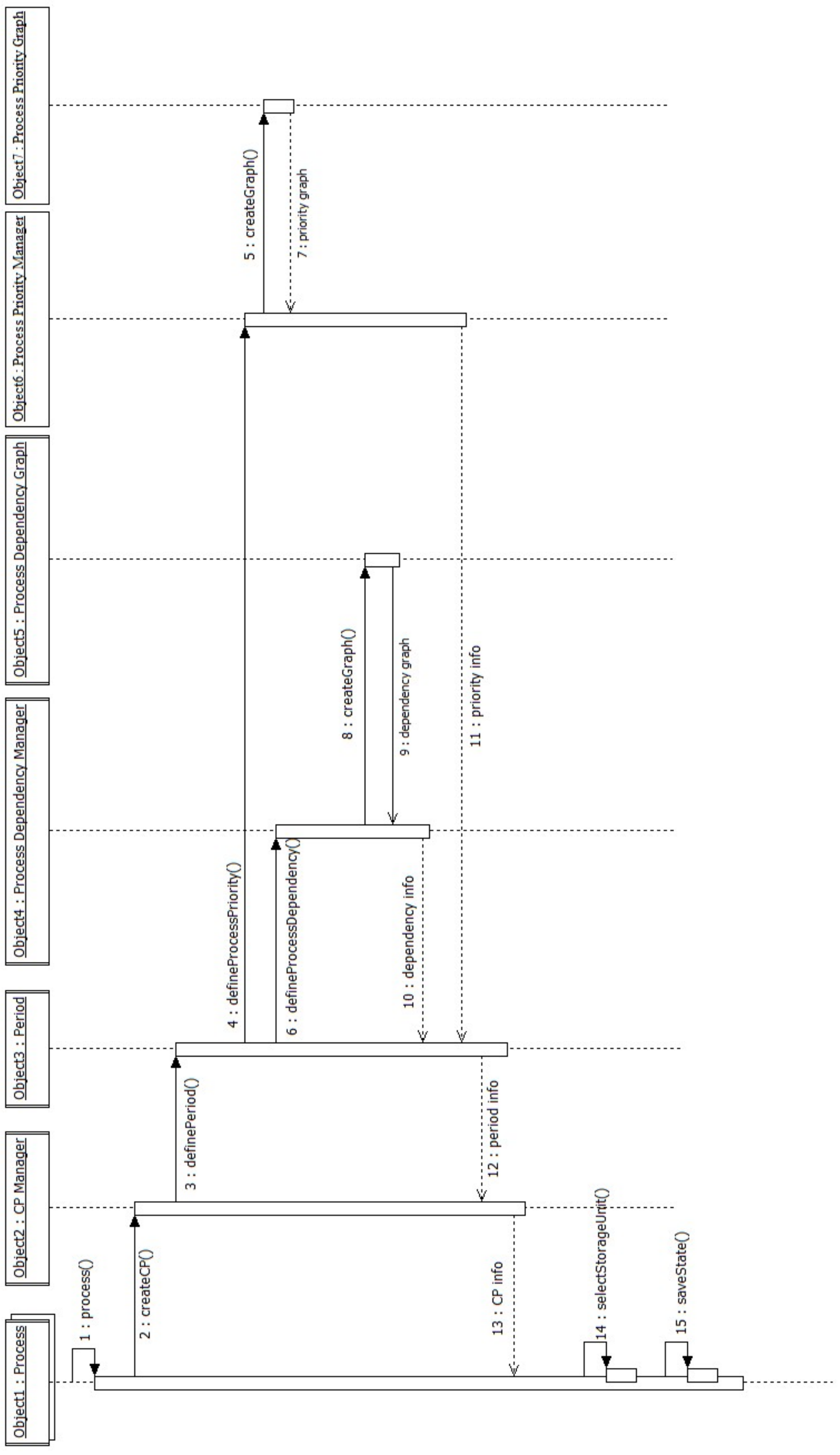


Figure 4.6: Checkpoint acquisition process

3. Checkpoint manager determines the most suitable checkpoint acquisition period in accordance with those two pieces of information and sends the result to related process.

4. Process, firstly chooses the unit (memory or disk: external, internal) that its state will be saved.

5. Saves its state info at the time of t by calling state save function. State saving procedure's sequencing diagram is shown at Figure 4.7. At state saving procedure, each process saves its state information when it is most suitable for the task.

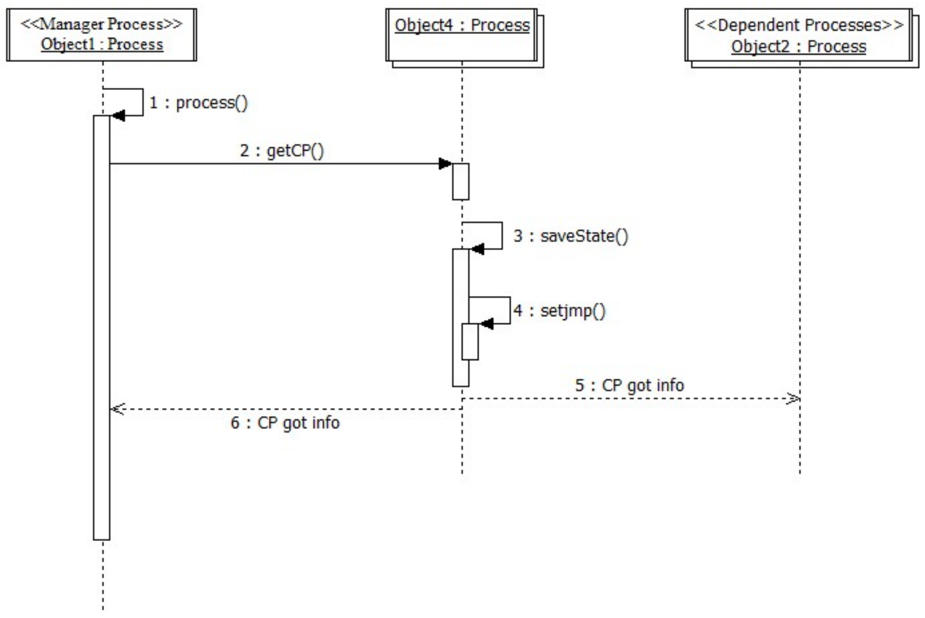


Figure 4.7: Process state saving

Faults at the program's code or at the data can cause hardware exceptions like illegal codes, bus or address exceptions, division to zero or other exceptions. VxWorks exception handling package deals with those kinds of exceptions. Default exception handler stops the task that causes exception and stores the task's state when exception occurs. In this case core and other tasks continue to function without any interruptions. Tasks are able to add their own debuggers for known hardware errors. If a task provides a signal handler for an error, default debugger does not intervene. Instead, new signal handler steps in. A signal handler that is defined by user is pretty much useful for overcoming catastrophic events.

setjmp() and longjmp() functions are being used for this purpose at VxWorks operating system. A process that wants to store its state uses setjmp() function. setjmp() stores the called state into 'jmp_buf' argument for its later use by longjump(). In short, setjmp() routine stores a process's state while longjmp() is used to recall the process from that stored state. Actual state of the working program completely depends on memory's (code, global variants, heap and stack) and registers' contents. Registers are composed of stack pointer (sp), frame pointer(fp) and program counter (pc). longjmp() uses those registers' contents that are stored by setjmp() for later installation. This way, after setjmp is recalled, longjmp() restores the program's saved state.

Example: int setjmp(jmp_buf env);

This function records current state of recorders in 'env'.

Recovery: Managing process will start the 'recovery' process by the error info coming from diagnosis unit. As the error info is going to include which process(es) is giving error, central process will send the information to all processes that recovery process has begun depending either on its previous or actual calculations of recovery line and along with this information it will also send the information about at which 't' time each process will return to checkpoint. Processes will be re-installed along with the state information at that t time according to the queue defined by recovery line. Each process will keep the information of where checkpoints are stored by itself. However, in case related processes malfunction, central process will also send the information of from where checkpoints will be restored along with to which checkpoints they will return to other processes. During recovery, only related processes will be restarted but processes that are not related to crashed or not crashed processes will not be restarted. For this reason, processes' relations to each other will carry importance.

Recovery line is the most important pillar of recovery process. While overcoming problems like process dependency, domino effect and garbage collection, Recovery Line also enables the processes to restart from the most convenient checkpoints. While recovery line is being calculated, the same two algorithms that give the same recovery line will be helpful. Roll-back Dependency Graph (RDG) and Checkpoint

Dependency Graph (CDG): One of the algorithms will be sufficient. Recovery lines that will be constructed according to these algorithms will be used to restart the processes.

Sequence diagram that corresponds to recovery process scenario is given in Figure 4.8.

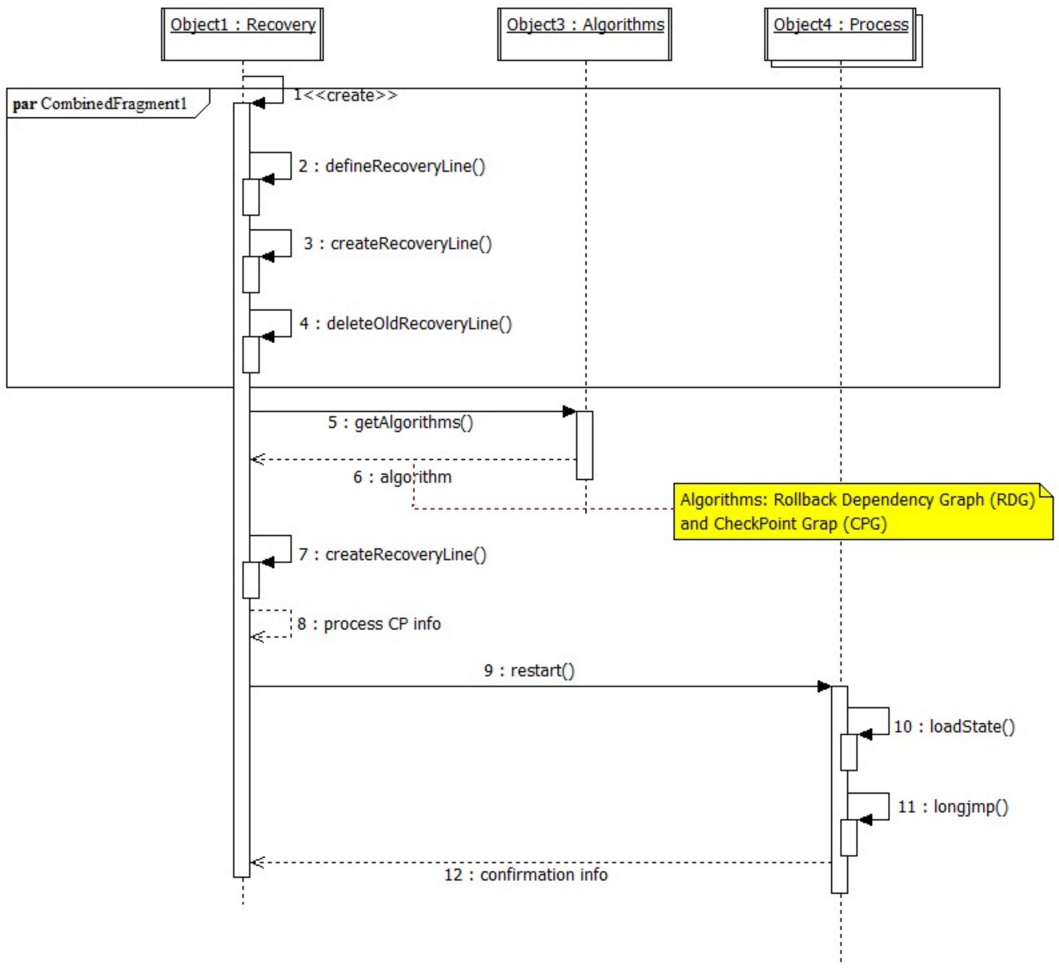


Figure 4.8: Recovery process

Recovery line is an asynchronous process that is calculated periodically and it is done independent of the recovery process. Thus determination of preserved Recovery Line number, calculation of recovery line, its creation and deletion of old recovery lines processes are all process that will be done parallel. These processes, in fact, will be executed in certain periods. As processes will be set free with regards to checkpoint acquisition, every process will send all checkpoints that they have taken to the central process. In order for the central process to be able to calculate recovery line periodically, it will be forced to wait all processes as this process will take place asynchronously and as all processes can not take checkpoint at the same time. Thus, periodical calculation of recovery line can be done only after all checkpoint information have been received from all processes. Number of recovery lines that the system holds will be defined by a configuration file and according to this, other recovery lines will be deleted for them not to occupy much space.

Processes during recovery operation are re-initiated with the order defined at recovery line from their cached state with setjmp() function by help of longjmp() function.

4.3.2.2 Class Diagrams

Class diagrams are among the basic components of architectural structure as seen from Figure 4.9. They model the data central structural units that are needed to fulfil above mentioned notions. They also define the units related to the code that will be developed when enough detail is provided. In this part, diagrams belonging to the classes that constitutes the design and class relations are being presented. Relationships that form as a dependency on data flow in general are modelled also between components that include these classes.

4.3.2.3 Architectural Components

In this part, components that are used at the architecture are defined. Units that will carry out processes which will appear during requirement analysis and method development are as in Figure 4.10. Also distributed structure of checkpoint management architectural components is depicted in Figure 4.11

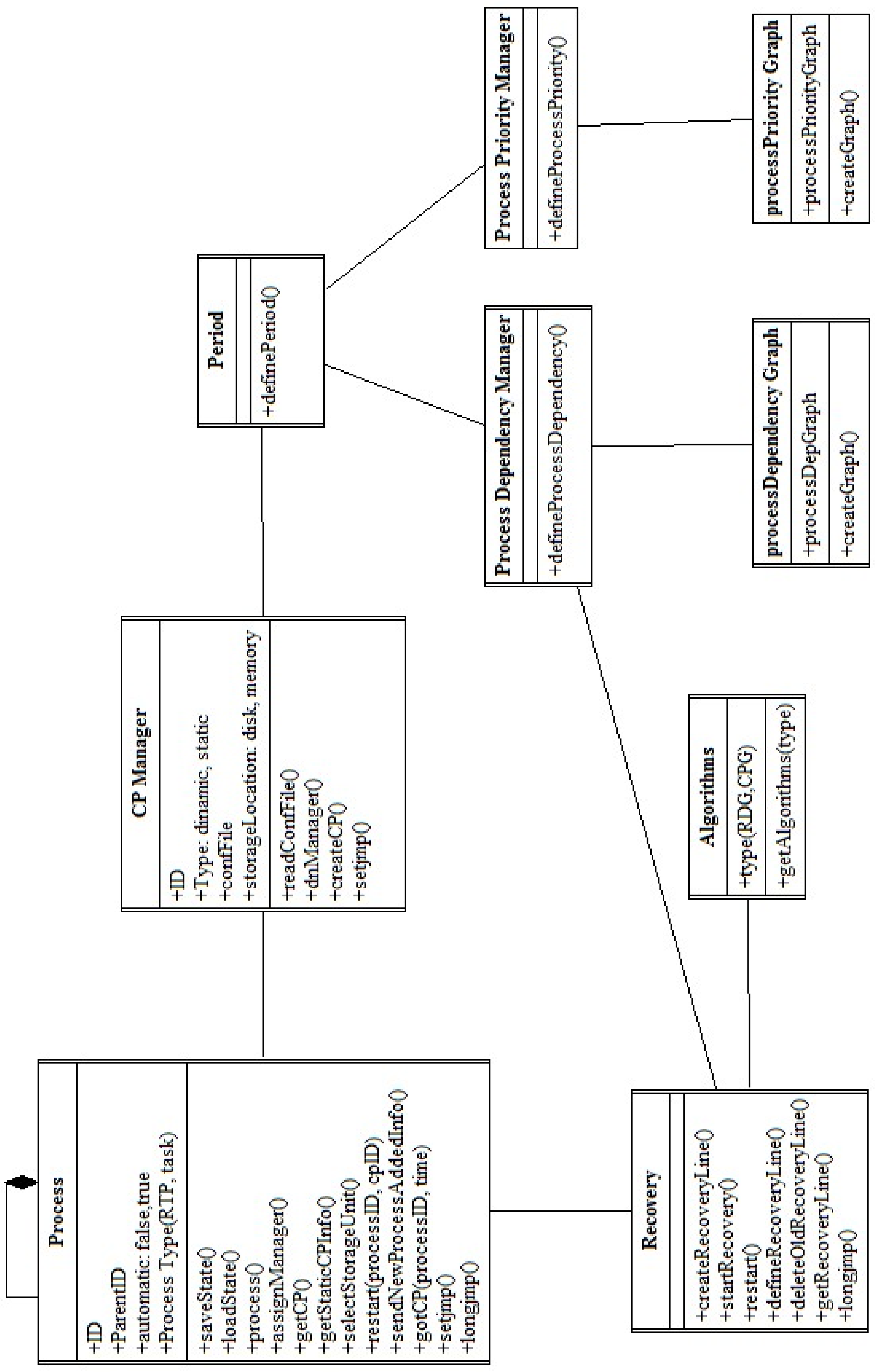


Figure 4.9: Class Diagrams

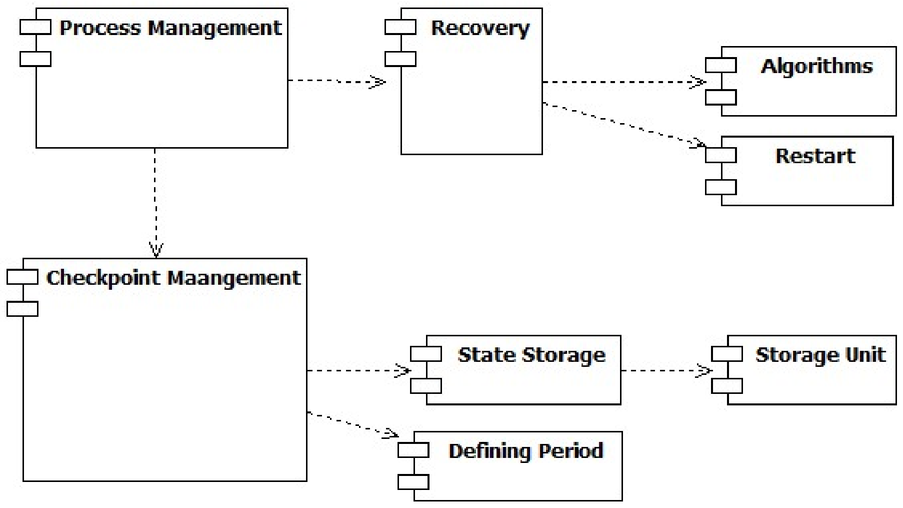


Figure 4.10: Component model for architectural implementation units

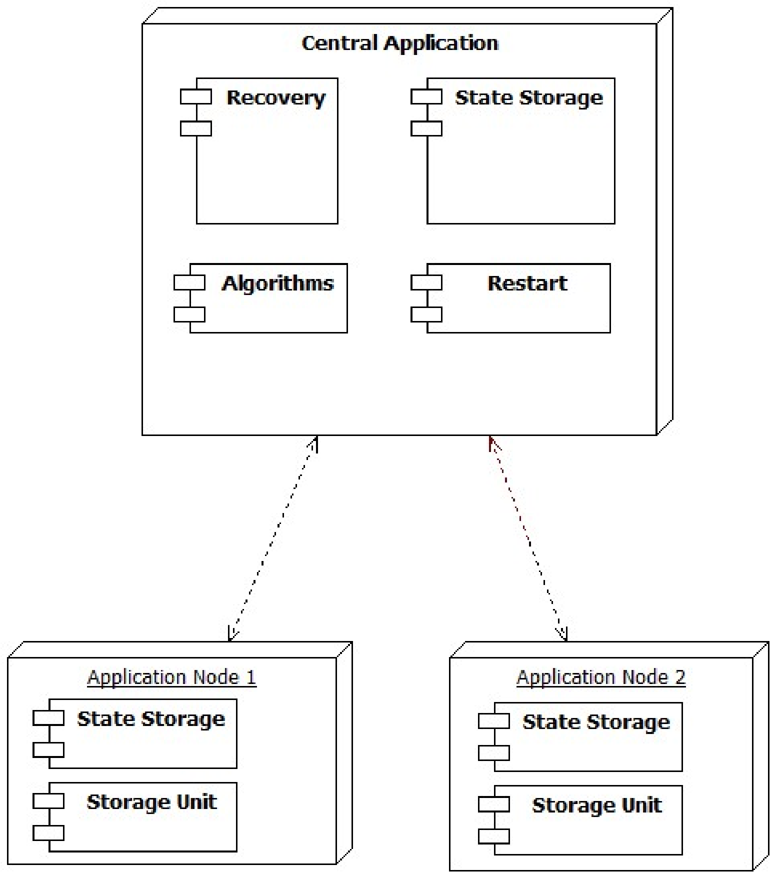


Figure 4.11: Distribution about checkpoint management architectural components

CHAPTER 5

LANGUAGE DESIGN

The design of an interpretive evaluation approach is presented for a rule-based fault monitoring language for processing events within the constraints of its application domain. A Dynamic subscription algorithm is applied to improve the time complexity toward acceptable levels. Time performance of existing language processors for rule based systems has been improved especially after RETE and similar techniques that however, could not be easily employed when events need to be processed. The approach presented in this article is demonstrated through a subset of the operations that is sufficient to represent all different cases for the developed language. A time complexity that is lower than linear is achieved, per received event.

5.1 CONSIDERATIONS IN DEFINING THE APPROACH

Fault management research and a software product-line adaptation for them have been considerably developed for mission critical applications [62] laying the foundation for this research. Low-level language syntax has been used to start experimenting with different languages for a variety of fault management needs such as monitoring and detection. Need for a higher abstraction level aroused and convenience in programming for such an established domain could be catered through a rule-based language.

There are many formalisms and languages that can be used for the event processing domain. A majority of those are very complex, difficult to learn and use, and hence, not convenient for repeated deployment of different fault management components for mission critical applications. A practical language was selected as GEM [75], due to its simplicity and ease of learning and use. Further adaptations were conducted to yield a yet simpler and more consistent language in its expressive style [63]. The earlier processing was performing a black-board metaphor and needed a speed improvement.

In general, the language consists of temporal, logical, and arithmetic operations. Some operands may be stored in a symbol table, representing the working memory or the fact base. Events on the other hand, mostly disappear after being received and instantly processed. Although there will be the capability to store values carried in by events, no such capabilities are addressed in this article where event processing is the priority. Events are also subject to logical connectives in expressions that are evaluated to Boolean values. What is interesting in this research is the temporal and logical operations on the events. Other kinds of expressions can be processed in established ways, mostly on an Abstract Syntax Tree (AST). Also, what is discussed in this article is the 'condition' part of the rules where the action part (Right Hand Side) will send a message, or set values in the variables, or create an event to be fed into the same system. The time complexity of processing the condition part is where the actual contribution is required.

Three operators are covered in this article. The ordering operator (";") is found to be demonstrative enough for the temporal operators. This operator is used to specify the case when one event has to happen after another. For example, the condition expression "a ; b" specifies the case when "b" should happen after "a", hence arriving of the "b" event after an "a" satisfies the condition. Two logical operators (AND: "&" and OR: "|") are also included that can represent the different cases in combinatorial operations.

The goal is to minimize the run-time temporal complexity. We took the liberty to spend larger times for the initialization of the data structures. The application environment can comfortably tolerate longer set-up periods. Also, given the problem constraints, memory space is not a problem within practical considerations

5.2 DATA STRUCTURES AND DESIGN PRINCIPLES

For each rule a separate AST (Abstract Syntax Tree) is provided as the heart of the mechanism to evaluate the condition expressions. A hash table is allocated in the main event loop that maintains the subscriptions to different event types: One arriving event, hashed to a specific location in the table, will start a series of notifications to the nodes of the ASTs that have subscribed for this kind of event. The key to our improvement lies with the "Activation Order Flow-graph (AOF)". This structure holds the activation order, to determine which node in an AST should start to listen to its incoming event. In other words, a node in the AST needs to know when to subscribe to an event through registering itself on the hash table. Figure 5.1 displays the main data structures and their operational connections.

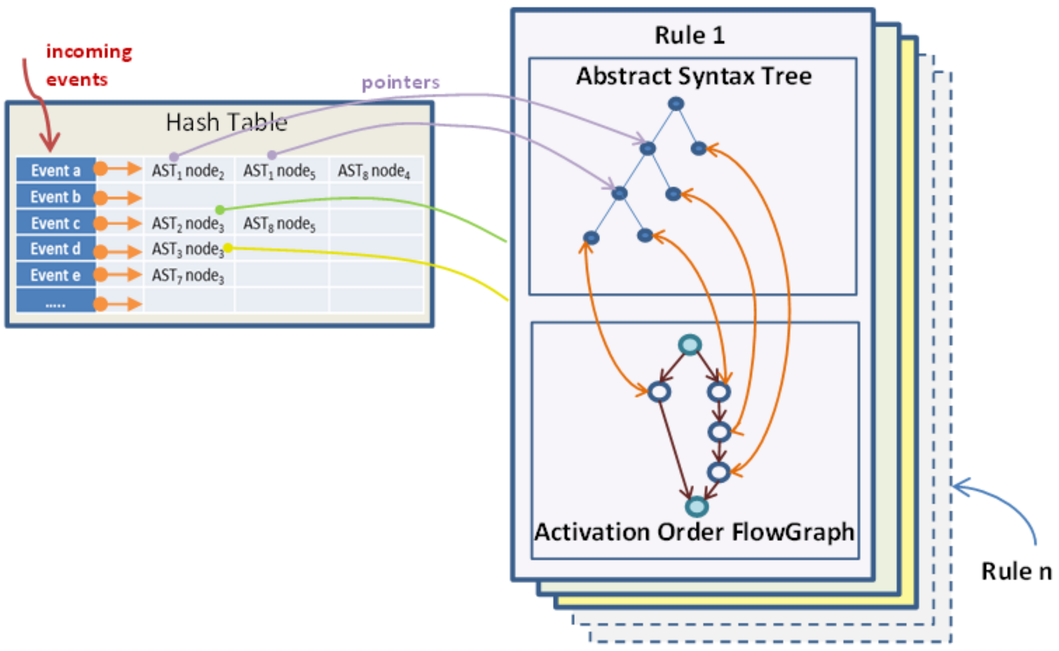


Figure 5.1: The three main data structures

To support run-time, pointers are involved where possible that eliminate the need for searches. At any given time, the hash table is only sensitive to the events whose turns have come in the ASTs. The system does not listen to the events that should not be active due to their orderings specified by the temporal operators in the condition specifications. This measure is also employed for saving computation time.

An event follows the pointers in the subscription list at the hash table to the nodes in the ASTs. Right after such notifications these AST nodes are unsubscribed unilaterally by the Hash Table. Hash Table becomes insensitive to this kind of events until at least one node from one AST requests subscription for this event type. A node receiving the event notification in an AST, attempts a partial or complete evaluation if all other necessary events have been received. After this, a new set of nodes in the tree should be ready to listen to their events – they should subscribe in the hash table. The current nodes that have completed evaluation or at least completed event notification reception let the AOF know themselves so that AOF can decide which nodes' turn it is to subscribe next.

5.3 ALGORITHMS

Among the three main data structures, algorithms concerning the AOF are presented here: the Hash Table is straight forward and the usage of the AST is quite common. Initialization concerns are different from the execution time therefore different algorithms are involved for initialization and execution times. Current implementation did not consider priorities for operators – usage of parentheses are enforced.

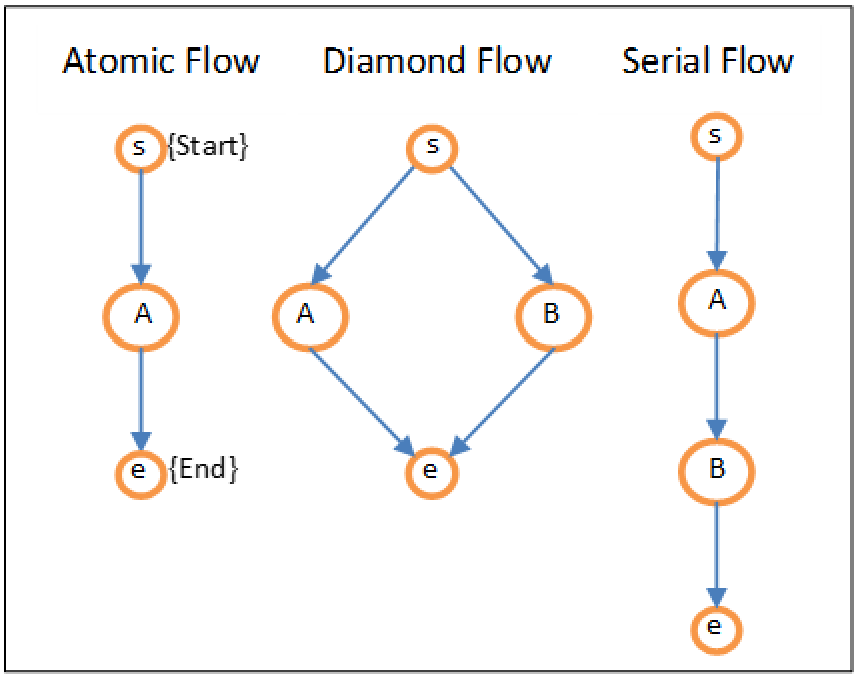


Figure 5.2: Fundamental flow types

The Activation Order Flow-graph (AOF) is constructed at the setup time, through a post-order traversal of the AST. The initial node of this graph points to the AST node(s) that need to be active first – at the beginning of the execution. One node in the AOF represents one node in the AST and this pair of nodes are connected. A node in the AOF keeps the activation order of a node in the AST. All the outflows of a node in the AOF represent nodes to be active concurrently after this one. The activation flow can thus fork where each fork needs to be met by a corresponding join in this flow-graph. Figure 5.2 depicts the three fundamental flow types that correspond to three cases: a single event, a parallel activation for 'and' or 'or' connectives, and a serial activation unit corresponding to the ';' connective. We refer to these units as atomic, diamond, and serial flows, each having a single start and a single end node.

There are pointers in both directions among the AST nodes and the AOF nodes. The links in the AOF represent the activation order. The two structures communicate within the following process:

1. An AST node notifies the corresponding node in the AOF of the fact that it has received its event or finished its evaluation

2. AOF proceeds to the next node(s)

3. Those next nodes in the AOF point back to AST telling the ASTnode(s) to be active: now they should subscribe to receive events.

The initialization of the AOF is represented with the following pseudo-code in Figure 5.3.

After the initial construction of the AOF, there needs to be a consolidation of neighbouring "fork" or "join" nodes. Alternatively, such successive fork or join nodes can be eliminated during the construction of the graph.

```
MakeFlowGraph (ASTnode  n)  // returns the start and end nodes
  [s, e] = Make New Start and End nodes;
  if   n is leaf
    Make Atomic-Flow (s, n, e)
    Return [s, e];
  Else if (n is “&”) or (n is “|”)
    [L_s, L_e] = MakeFlowGraph (n.LeftChild)
    [R_s, R_e] = MakeFlowGraph (n.RightChild)
    Make Diamond-Flow (s,e,L_s,L_e,R_s,R_e)
    Return [s, e];
  Else if  n is “;”
    [First_s,First_e]=MakeFlowGraph(n.LeftChild)
    [Second_s,Second_e]=MakeFlowGraph(n.RightChild)
    Make Serial-Flow(s,e,First_s,First_e,Second_s, Second_e)
    Return [s, e];
```

Figure 5.3: The initialization of the AOF

5.3.1 Run Time Considerations

Except for a sequence operator (“;”) no operator causes an ordering: all the event expecting nodes should be activated at the same time, if they come before in a depth first search, a sequence operator node in the AST. Forking nodes correspond to non-time-ordering combinatorial operators (&, |).

One corrective action is required in case of some situations caused by some event sequences. Such cases require the passivation (un-subscription) of some nodes without waiting for their events. Explained at the end of the next section, process can be carried out by using lists of nodes to unsubscribe held at the nodes that are responsible for the evaluation. Figure 5.4 presents a sequence diagram for demonstrating the runtime flow of actions as the main operation scenario.

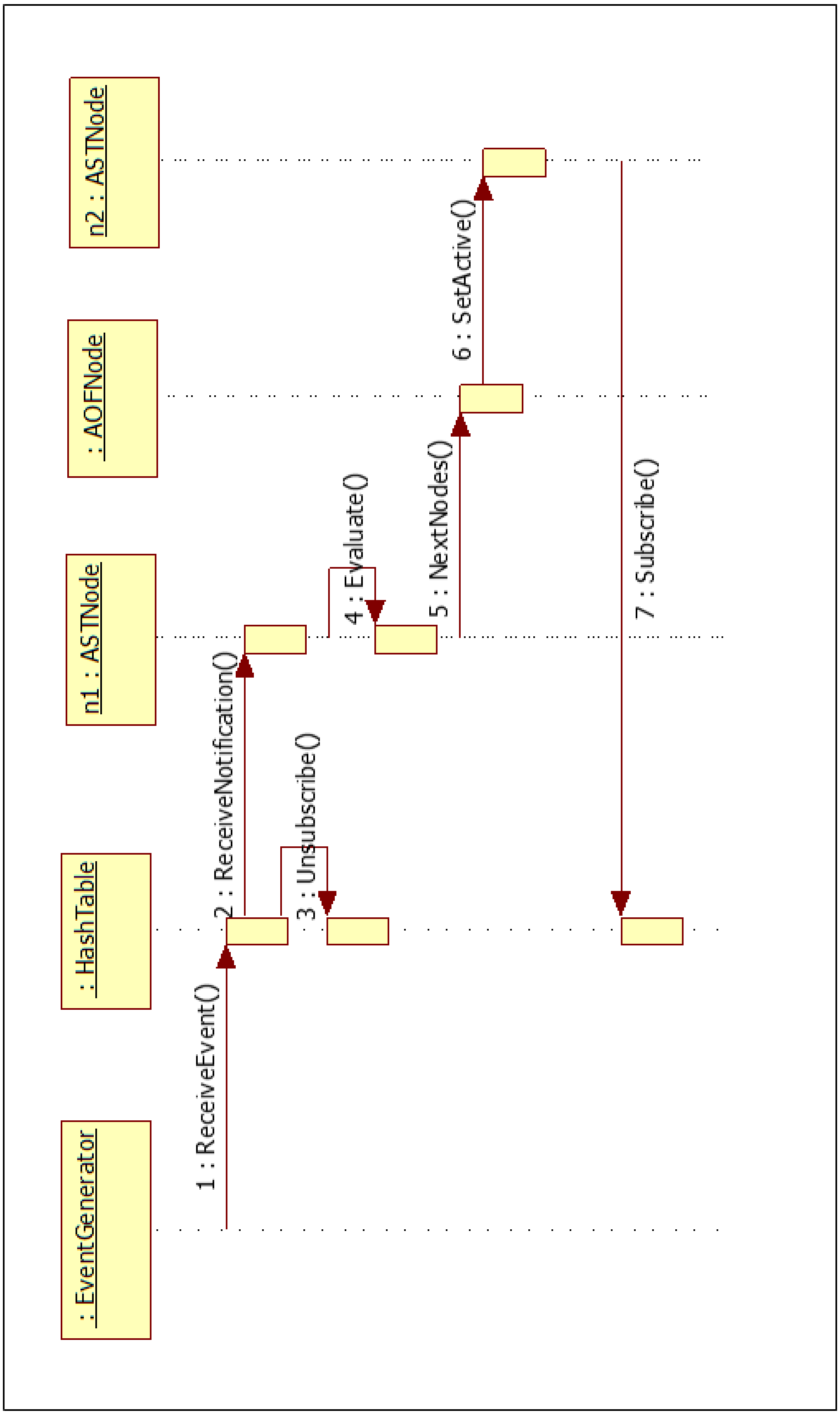


Figure 5.4: Sequence diagram for the run-time operation scenario

5.4 A TRACE THROUGH AND COMPLEXITY ANALYSIS

The goal is to conduct minimal work upon reception of an event. However, this approach assumes less than O (n) processing time per event identifier appearance in the rules. If a condition is satisfied before receiving all the events in the expression, evaluation should terminate but still, work needs to be done to unsubscribe for the events not yet received. Although not increasing, the total processing time for the rule does not also reduce, it is always about n operations of O (n) where "n" is the total number of event identifiers in the condition part of the rule. Tracing through the data structures, following the path of incoming events will help demonstrate the details of the computation time:

On the hash table, incoming event can hash to a row in the table in constant time. Then per each subscribed node, a notification will be conducted again in constant time via pointers held on the right of the row. One event may result in O (n) notifications but we are targeting a complexity that is assessed per event identifier in the rule expression. One notification costs constant time complexity.

On the AST, a node receiving the notification, will change state (from 'active' to 'mature') and ask its parent to attempt an evaluation. An event can result in an operation with two operands: current node, its parent holding the operator, and its sibling are involved and this results in constant time. However, if an evaluation was conducted, this time the parent turns into 'mature' state and asks its parent for evaluation. Depending on the states of the other siblings of the parents up on the path to the root, more evaluations may follow each other, rendering an extra complexity of log (n). This is because of conducting constant time computation per event reception, but not at the time of the reception due to the readiness of the siblings and accumulating them to be conducted later.

After AST, next stop in the trace is the AOF. It should be noted that different regions in the AST can be active (listening for events) concurrently. All such regions upon maturing, that is receiving the notifications or conducting an evaluation, should notify AOF and in return, AOF will cause the new locations to be active. AOF links correspond to "next node for activation" relation. A request comes to AOF from the

AST, carrying the current node. All the outflows for that current node are triggered and such next nodes cause their AST counterparts to subscribe in the hash table. That costs O (n) computation.

An additional computation arises due to the need to unsubscribe the nodes that did not need to wait for event notification. For example, an OR connective ("|") may correspond to a single event as one of its operands and a huge complex sub-tree as the other. If this single event arrives, there is no need to evaluate the other operand that may be an expression involving many events. All those events need to be unsubscribed. This is the worst case O (n) computation that might be required during the processing of the arrival of one event. Again, this situation is due to the fact that those event identifiers residing in the condition did not use their computations that would cost constant order per event identifier because they did not have the opportunity to receive an event. Lists of nodes to unsubscribe are prepared at initialization time, to avoid traversing the tree to find each such node, that would add to the time complexity.

5.5 AN EXAMPLE

One rule will be used to demonstrate corresponding data structures and processing operations. In real applications, there will be many rules executing at the same time. The following expression is considered as the condition part of a rule:

(A & B) ; ((C | D) & E)) (1)

The AST corresponding to (1) is provided in Figure 5.5.

The AOF for this example is depicted in Figure 5.6. The sole responsibility of this graph is to determine the next set of nodes in the AST to be active given a current one. There could be many active regions concurrently in one AST where "active" means subscribed to be notified by events.

Subscription of an AST node is done in the Hash Table which has its keys as event types. An event received by the system is searched in the Hash Table by directly checking the location through a hash function and is ignored if no key is found at the corresponding entry. Once an incoming event matches a key in the table, the list of subscribed nodes corresponding to that key is traversed for notifying each element. There are pointers from this list to the nodes in the ASTs for different rules. To keep the search time in this table to a minimum, an event type that has no current registration is not kept in the table. Likewise, potential subscribers are not kept in the lists in this table. They are inserted during subscription and removed during un-subscription.

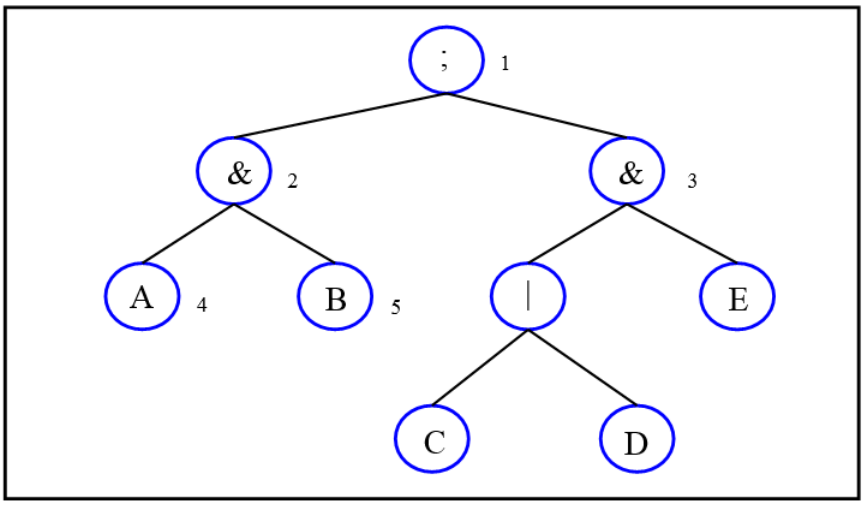


Figure 5.5: AST for the condition expression (1).

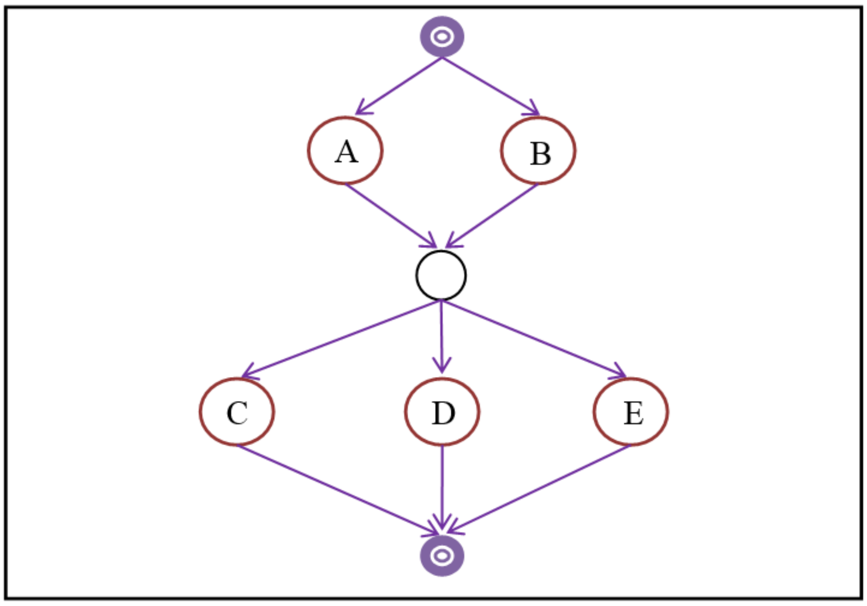
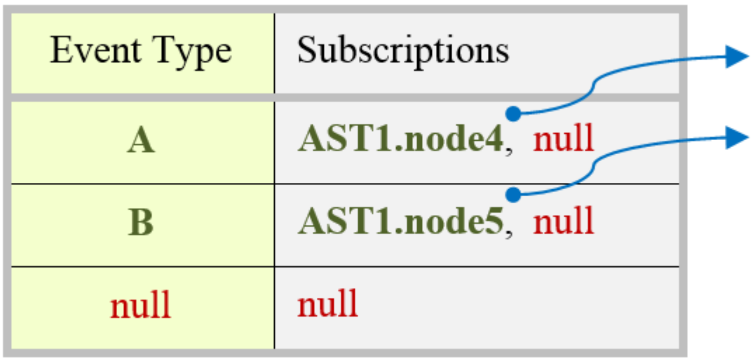


Figure 5.6: AOF for the example.

The alternative is to keep inactive keys and potential subscribers in the table accompanied with state information that carries the "passive" value. This alternative will lengthen the key search and for list traversals although for a small difference. However, a slightly costlier operation of insertion (or deletion) is necessary during subscription (or un-subscription) in this case but of constant time complexity rather than linear. The hash table after the initialization of the program is shown in Figure 5.7. Only hash entries for the event types A and B are initialized with keys, due to the fact that nodes in the AST that corresponds to other events (C, D, and E) are not active in the beginning. They can only start listening to events after the (A & B) condition will hold that is after receiving A and B events. This is due to the sequence connective “;”.



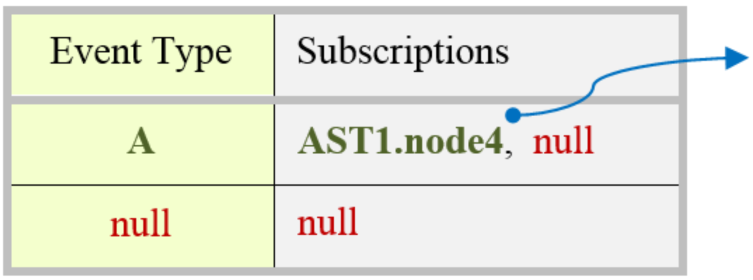
Event Type	Subscriptions
A	**AST1.node4**, null
B	**AST1.node5**, null
null	null

Figure 5.7: Hash Table after the initialization of the program

5.5.1 Execution

Now that the important components are initialized, a trace can be conducted to observe the variations on them at execution time. Any received event that is not of type A or B will be ignored by the Hash Table. An event type of B for example, will be directed to related entry and collide with the key, meaning that it is a desired event. All the nodes listed to its right need to be notified and then their pointers deleted from the Hash Table for an immediate un-subscription. Since an arriving event will be directed to all subscribers for this event, there is no need to keep it in the table: After notifications, an event type will have no subscribers left.

In the AST in Figure 5.5, node 5 labelled with "B" was the subscriber for this kind of event, and will be notified. Upon notification, it will change its state from "active" to "mature" and will request an evaluation from its parent. Since the parent needs the left operand to complete its "and" operation, with missing left operand it cannot proceed with the evaluation. This is where processing for this event stops. Now, the Hash Table and the AST have advanced to the status shown in Figure 5.8 and 5.9.



Event Type	Subscriptions
A	AST1.node4, null
null	null

Figure 5.8: Hash Table after receiving a "B".

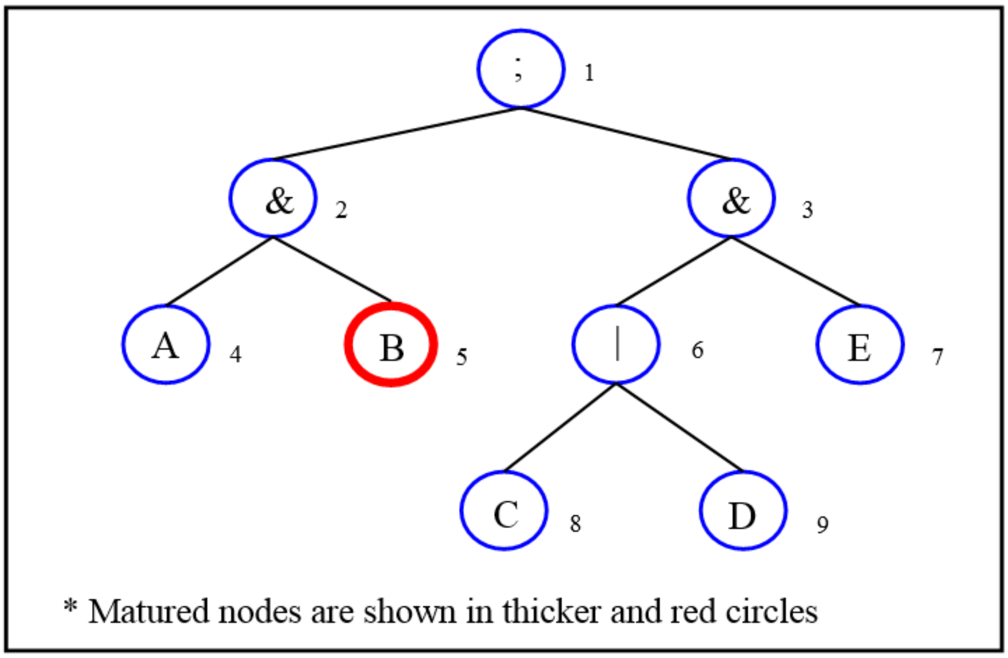
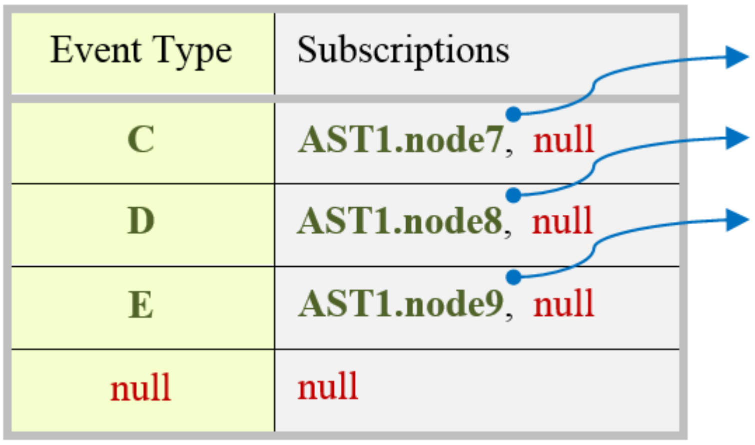


Figure 5.9: AST after receiving a "B".

Next, if an "A" arrives, Hash table, otherwise ignoring any other event, will notify the "A" labelled node in the AST. Also un-subscribing this event, the Hash Table is now completely empty. This "A" node in the AST after maturing its state, will ask its parent for evaluation. Now that all the operands are ready, the evaluation proceeds and the 2nd node ("&") now turns to mature state. Now the AST needs to check if this evaluation will trigger further evaluations; the 2nd node requests evaluation from its parent, the 1st (“;”) node. With only its left child matured, this node cannot perform evaluation. Therefore, the 2nd node (“&”) being the last one to conduct evaluation assumes the responsibility to inform the AOF in order to prepare other nodes for subscription.

The joining node in the AOF for "A" and "B" corresponds to the last evaluated node (node 2) in the AST. Through the pointer between those corresponding nodes, the node in the AST stimulates this join node that wakes the C, D, and E nodes in the AOF to trigger their counterparts back in the AST for subscription. Figure 5.10, 5.11, and 5.12 depict the states of the data structures after these subscription operations.



Event Type	Subscriptions
C	AST1.node7, null
D	AST1.node8, null
E	AST1.node9, null
null	null

Figure 5.10: Data structures after first partial evaluation

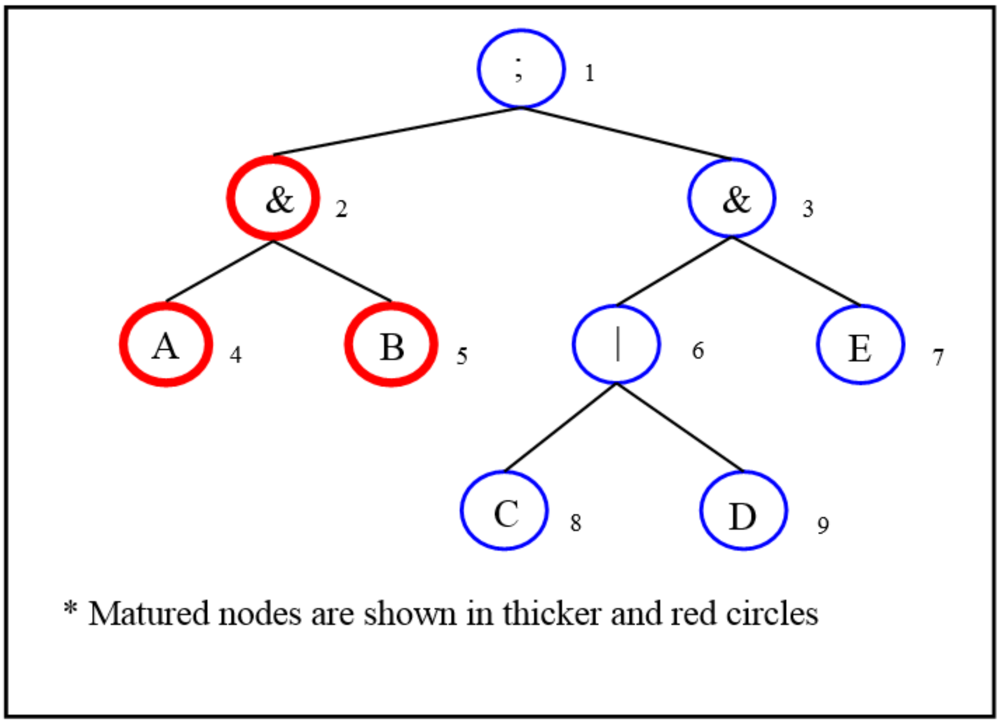


Figure 5.11: AST after the first partial evaluation

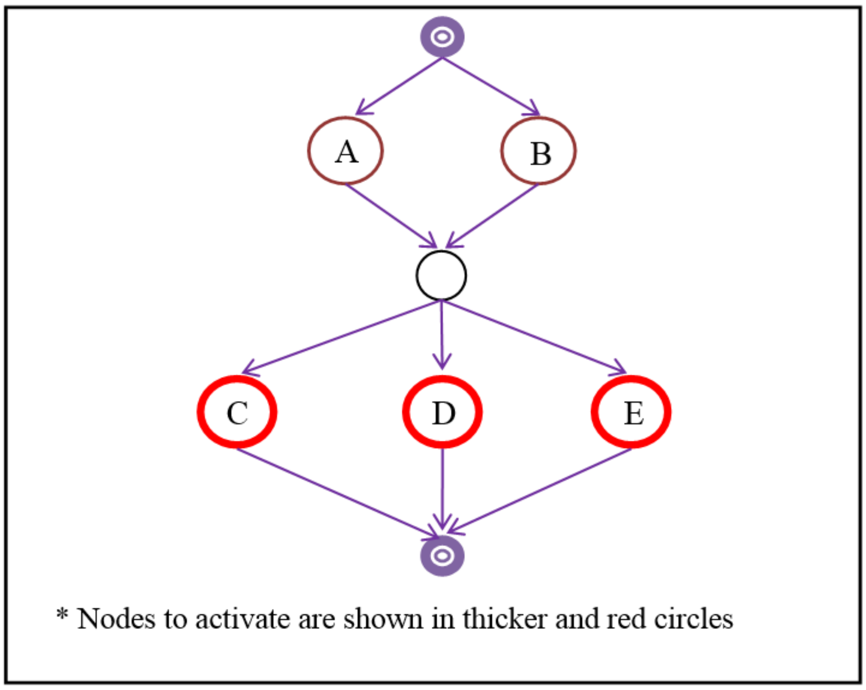


Figure 5.12: AOF after the first partial evaluation

Now a special kind of processing will be demonstrated through the reception of a C kind of event: the parent for the corresponding node in the AST holds the "or" operator (“|”). Receiving only one operand should be sufficient for an evaluation. Without waiting for a D event, the evaluation proceeds. This time however, since the node in the AST corresponding to the D event should be forcefully unsubscribed since it did not have the chance to automatically unsubscribe. Such nodes remaining subscribed may cause undesired repeated evaluations. The AST nodes that carry an or operator, also carry a complete list of the nodes in their sub-tree to traverse them in linear time for unsubscription.

5.6 IMPLEMENTATION RESULTS AND EVALUATION OF THE PERFORMANCE

To verify the algorithms, the system has been implemented using the Python language. Pointers have been emulated through the data structures provided in Python. Execution results were found satisfactory with the experimentation rules created with an anticipated resemblance to the environment of the application. Also events were generated through an event simulator developed within the same environment.

The following rules were written to test the system. Events are named with single capital letters. The random event generator generates events named 'A' through 'Z'. Not all the generated events are used by the rules. The generator was programmed to produce 10% of its produced events that are not present in the rule expressions, and 90% that are included in the rules.

Rule1 = ((((A;(B&C))&(D&E))&(F;G));(H;K))

Rule2 = ((A;(C&(E;F)));(((K;T)|(M;N));P))

Rule3 = ((A;(C|(E;F)));(((K;T)|(M;N));P))

Rule4 = ((A|(B|(K|T)));((F;M)|(P;R)))

Rule5 = ((A|(B&(K|T)));((F;M)|(P;R)))

Rule6 = ((A&(B|(K|T)));((F;M)|(P;R)))

Rule7 = ((A|(B|(K&T)));((F;M)|(P;R)))

Rule8 = ((A&(B|(K&T)));((F;M)|(P;R)))

Rule9 = ((A|(B&(K&T)));((F;M)|(P;R)))

Rule10 = ((A&(B&(K&T)));((F;M)|(P;R)))

A basic experiment was run to assess the evaluation times for the condition statements in the rules. Different numbers of events were generated and the experiment was repeated for those different cases. In case an event caused a processing in any of the rules, the computation time for that rule was recorded. The results are reported here for the case when the program was run for one million events. On the average, less than 30 microseconds were spent per rule update. Table 5.1 lists the computation times for each rule.

Table 5.1: Rules and Computation Times

Rule	Number of Updates	Average Update time (microseconds)
1	242421	32.82
2	15	26.64
3	39	23.93
4	228545	31.09
5	245012	33.21
6	236915	32.15
7	37	29.11
8	132118	30.95
9	129085	30.49
10	185	24.02
Average:	121437.2	29.44

The computational environment included Windows 7 64 bit operating system, Intel I7 CPU Q720 processor running on 1.6 GHz clock, and 8 MB System memory. Implementation language was Python version 2.7.

5.7 COMPUTATIONAL COMPLEXITY CONSIDERATIONS

Regarding a set of rules as the determining factor for the problem size, the operands in the condition parts of the rules constitute the input. Assuming an operation to be carried for each operand, the best case yields n (n = input size) operations. Spending constant time on each operand, i.e. reception of an event, therefore would be a very good accomplishment. This is not very difficult to achieve if the average case is considered. If all the events need to be received, the total execution time can be handled in "n" constant time operations. However, there will be cases where an operand will be ignored for current care, instead, time for it would be spent during the processing of another operand. This may cause accumulation of postponed computations ending up with a burst of linear complexity computation coinciding with a specific event input. Keeping the average same, processing of a particular event input could theoretically take n-1 operations. Although practical scenarios indicate such cases would not require few operations, the general complexity is in principle, O (n).

What matters in the application is the CPU load an incoming event induces. As long as there are not many events arriving at unanticipated frequencies an acceptable computation time per event reception will comply with the real-time expectations. The concern in this project is to try to achieve near constant time computations per event reception. This is mostly achieved. The less likely cases of having excessive processing however is not impossible, but luckily easy to predict, only by inspecting the rules. It may be possible to overcome these incidents of condensed computation through rearranging the rule set. For example, an expression that includes a big number of events taking part in a multi-operand ’or’ operation, can be broken into many rules that only consider a few of them at a time and update a fact to represent this intermediate result. Table 5.2 summarizes the different operations to be performed when an event is received.

Table 5.2: Computation Requirements per Event Reception

Operation	**Computation Complexity**	**Explanation**
Search for an event in Hash Table	O (1)	For checking if any subscribers exist
Notify subscribers for the event	O (n)	O (1) per notification.
Evaluating expression after event notification	O (log(n)), per notification; Worst case:n log (n)	An evaluation node asks its parent for possible evaluation, potentially, evaluations can extend up to the root. Also, simultaneous similar operations may take place at different ASTs/sub-ASTs.
Finding next nodes to subscribe	O (n)	Each node takes O (1) time; there could be many nodes to activate.
Re-initialize after one rule fires	O (n)	Subscribing ‘next’ set of nodes where next is the initial set.

n: number of operands in all rules

The biggest time in worst case, can be consumed by the evaluation operations, if they happen to be taking place at the same time: One event may trigger different locations on one AST, and in many ASTs, that attempt an evaluation. One evaluation will take place accessing two operands and conducting the calculation and storing the result. So far this atomic evaluation spends constant time. However, once completing evaluation, a node tries to activate its parent for the parents own evaluation. This kind of evaluation requests may be honored in a chain that could continue along the depth of the tree, hence causing log (n) computations. So far, an evaluation path caused by the notification of one event to one AST node was discussed. If this happens at multiple locations on one or more ASTs, the log (n) computation time will be repeated for O (n) event notifications. This case is another worst case. One possible way to deal with such a situation is through a static code analysis tool: An automatic resolution, even a methodological suggestion may not be possible for example to change the rule set in a known pattern. However, a warning can be made and the time cost for such a situation can be calculated and presented to the programmers. The analyser should search for an event that notifies many nodes, and consider the distance of each such node to its root in ASTs.

Current environment expects a constant flow of events with known frequency that is about at lower milliseconds range. For example, continuous monitoring of some environmental values through sensor readings can create such input. On the other hand, some sporadic events are also expected, such as CPU temperature. The worst case for the repeated and converged occurrence of such sporadic events in our application domain also is within the processing capability of the application. The real-time constraints of the domain therefore are possible to be satisfied. However, the worst-case "n log (n)" characteristics of the solution should be analysed not to hamper the speed requirements.

CHAPTER 6

EVALUATION

Most of the work related with event processing or improving the efficiency of rule based systems present their results with an evaluation in computation times. Although their specifics may not be published, our algorithms are open and time complexity analysis is presented. It is also possible to provide a comparison based on CPU times. For this purpose, some tools with the closest purpose to ours have been selected. To give an idea about the speed of the processing, describing the underlying system is helpful: Windows 7 64 bit operating system, Intel I7 CPU Q720 processor running on 1.6 GHz clock, and 8 MB System memory. Implementation language was Python version 2.7.

Since keeping the processing time per arriving event to a minimum was an important goal for this work, the performance evaluation is presented here in the average processing times for event arrivals. To be able to present a comparison with the existing tools, similar numbers of rules and events were generated and performance was measured. Table 6.1 lists the processing time in micro seconds, for such different sized problems. Only event types that took place in the rule expressions were generated in the experiment. A random event generator is employed to drive the experiments. Some event arrivals caused the firing of an event, without causing a significant processing difference from the others that only caused partial evaluation. Whereas some events were only searched for currently active subscriptions and not processed any further if their time was not ripe.

The other three work in the literature that provided measured times were used to present a comparison. However, a precise definition of how they were measured is

Table 6.1: Average Processing Times per Event Arrival in Microseconds

	# of Events					
# of Rules	1	10	100	2000	10000	100000
10	21.22	33.34	39.29	37.43	42.25	42.58
100	19.5	24.61	34.34	34,21	37.58	40.59
1000	25.63	28.4	43.69	32,77	50.37	n/a
2000	26.97	28.5	32.77	51.19	54.28	61.22
10000	27.03	30.28	50.37	51.19	60.61	n/a

missing. We took the most sensible interpretation for such published data in our effort to compare similar information. Out of those, more detailed but less explained measurements were provided in [128] where Xiao et al. report an improvement over Rete with no event processing mentioned. This work is also included in the comparisons due to the available data and to provide an idea in comparing event processing and non-event processing cases presented as we reflected in Table 6.2, in terms of number of rules, "data scale," and milliseconds.

Table 6.2: Processing times in Xiao et al.'s work [128] in milliseconds

	Data Set Scale			
Rule Set Scale	1000	2000	10000	100000
10	9	21	100	1040
20	12	22	115	1120
100	19	35	206	2320
1000	20	40	230	2640
1000	22	46	252	2870

In [130] the authors claim that they improve Rete in terms of temporal operators. They used after and before operators and also combination of them to define temporal constraints between events. They compare their algorithm which is called Rete' to Rete and got the following results in Table 6.3. However there is no detailed information about the number of rules, rules structure, events and complexity of the events that were used in the test.

This information can probably be interpreted as the total time spent in processing the number of events presented in the Data Size columns, for one row at a time.

Table 6.3: Processing times in Zhou et al.'s work [130] in milliseconds

	Rete	**Rete'**
Telephone system	0.4072	0.1008
Alarm system	4.8038	0.6877
First aid system	0.0776	0.02

The performance of the system is slightly reducing when a bigger rule set is run, corresponding to a row in this table. In Zhou et al.'s work [130] measurements that can only be used in their improvement are presented. Their approach as they refer to Rete' looks to have outperformed a Rete implementation by about a factor of 4. Problem specifics or the granularity of the measurement units are not mentioned. This study cannot be used in a comparison but can give an idea if the times can be interpreted as per event or per rule etc.

Finally another improvement work is published by Moreto and Endler [78] where similar to the work in [130] numeric results are more towards assessing their improvement – not as much for comparisons. Using different tree structures they have improved the time complexity from exponential to near linear. However, the small data cases corresponding to the beginnings of the two curves both correspond to big performance times. As they conservatively mention that their application can process 1000 to 2000 event instances per minute. Whereas the graph in their Figure 7 indicates a time of around 350 seconds for 2000 events, for their improved case. This is a result we can use for comparison, assuming 350 seconds for the total time spent for 2000 events.

Table 6.4 presents the problem size that can be found in both [128] and [78] for a performance comparison. Since [78] does not report rule counts, different rule count cases were averaged in [128] and in this work and used in Table 6.4. Whereas Table 6 displays other problem sizes available in [128] for a comparison with this work. An overview of the comparisons would render a rough judgement as the proposed approach being very faster than [78] whereas slower than [128]. However there are problems in the comparison with [128] simply because of lack of precision in the definition of the performance times reported by them. Also, they do not process events. This is a mere RETE improvement work. This comparison is made assuming that the times are for the total processing for all the fact notifications presented in [128] as "data scale" that we compare with event counts.

Table 6.4: Comparison with two studies

	Average Total Processing Time in milliseconds		
Approach	10K events	100K events	Comments
Proposed approach	477	4159	Averaged for different rule counts
Xiao et al. [128]	180.6	1998	Averaged for different rule counts
Moreto & Endler [78]	>40000	n/a	Estimated from curve

Table 6.5: Comparison with Xiao et al.'s work [128]'s work for performance times in milliseconds

	# of Events					
	1000		10000		100000	
# of Rules	[128]	This work	[128]	This work	[128]	This work
10	9	42.45	100	422.5	1040	4258
100	19	36.29	206	375.8	2320	4059
1000	20	49.58	230	503.7	2640	
10000	22	56.76	252	606.1	2870	

One more observation is possible when the curves are plotted for the two approaches, that display processing times as functions of event counts. Different curves are provided for different program sizes in terms of numbers of rules. When the general characteristics of the curves are compared, the proposed approach yields a more linear trend and the approach presented in [128] yields an exponential trend.

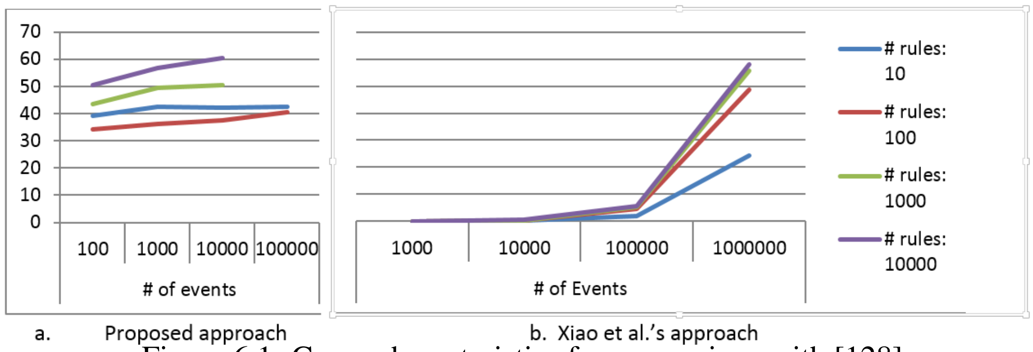


Figure 6.1: Curve characteristics for comparison with [128]

Considering the available information in the literature, there are parameters that would affect a precise comparison. Some work is reported with the number of rules and number of events. However, the complexity of the rules and events (what is included in an event) are usually not described. Within such constraints, our results provide reliable margins suggesting a preferable use in appropriate real-time applications. As an overall statement, the presented solution provides probably a lower-abstraction level environment for rules but predictable and for most cases faster processing when compared to the prominent alternatives. Table 6.6 provides further considerations in the interpretation of the comparisons.

Table 6.6: Characteristics of the alternative Approaches

	Speed	**Platform**	**Event Processing**
Proposed approach	fast	Python language	yes
Xiao et al.	faster	n / a	no
Moreto Endler	slow	Java language	yes

CHAPTER 7

CONCLUSION

The approach demonstrated itself as satisfactory. The estimated performance as a result of the complexity analysis is better than the alternatives as far as we could access their information. Our experimental assessment is based on the prototype implementation of the processing environment for the limited functionality that can be expanded to a more complete language. For example, specifying partial condition expressions and using those through their identifiers, in bigger expressions is in the immediate list for future enhancement. However, such expansion is not expected to add extra complexities due to the selection of the included capabilities in the current prototype that resemble the different cases in a complete language our domain may suggest. This also defines the future work, to include other operation types and mixing the event type of operands with variable type operands. A C++ implementation is expected to be completed where different data structures and some fine tuning on the algorithms can be experimented. One interesting study will be to access the necessary information for complexity analysis for other event processing implementations in order to conduct a more comprehensive comparison.

REFERENCES

[1] Waheed Ahmad, Andrei Lobov, and Jose L Martinez Lastra. Formal modelling of complex event processing: A generic algorithm and its application to a manufacturing line. In *Industrial Informatics (INDIN), 2012 10th IEEE International Conference on*, pages 380–385. IEEE, 2012.

[2] Elizabeth M Allen. *YAPS: Yet another production system.* University of Maryland, 1983.

[3] N. Altintas, S. Cetin, and A. Dogru. Industrializing software development: The factory automation way. *Trends in Enterprise Application Architecture*, pages 54–68, 2007.

[4] N. I Altintas, S Cetin, H Oguztuzun, and A.H. Dogru. Software asset modeling with domain specific kits. *Trends in Enterprise Application Architecture*, 2012.

[5] James L Alty and Mike J Coombs. Expert systems: concepts and examples. 1984.

[6] Darko Anicic, Paul Fodor, Roland Stuhmer, and Nenad Stojanovic. Event-driven approach for logic-based complex event processing. In *Computational Science and Engineering, 2009. CSE'09. International Conference on*, volume 1, pages 56–63. IEEE, 2009.

[7] Yeshayahu Artsy and Raphael Finkel. Designing a process migration facility: The charlotte experience. *Computer*, 22(9):47–56, 1989.

[8] Michael Salkham AS'AD. *Fault detection, isolation and recovery (FDIR) in on-board software*. PhD thesis, Chalmers University of Technology, 2005.

[9] Logic Programming Associates. Expert system shells. http:en.wikipedia.orgwikiDomain-specific_language, last visited on February 2014.

[10] Mohsen Bashiri, Seyed Ghassem Miremadi, and Mahdi Fazeli. A checkpointing technique for rollback error recovery in embedded systems. In *Microelectronics, 2006. ICM'06. International Conference on*, pages 174–177. IEEE, 2006.

[11] Don Batory. *The leaps algorithms*. Univ., Department of Computer Sciences, 1994.

[12] Bruno Berstel. Extending the rete algorithm for event management. In *Temporal Representation and Reasoning, 2002. TIME 2002. Proceedings. Ninth International Symposium on*, pages 49–51. IEEE, 2002.

[13] Deng Bo, Ding Kun, and Zhang Xiaoyi. A high performance enterprise service bus platform for complex event processing. In *Grid and Cooperative Computing, 2008. GCC'08. Seventh International Conference on*, pages 577–582. IEEE, 2008.

[14] Craig Heath Bob Blakley. Security design patterns. Technical Guide, The Open Group, April 2004.

[15] Anthony J Bonner and Michael Kifer. Concurrency and communication in transaction logic. In *JICSLP*, pages 142–156, 1996.

[16] Mr Jérôme Boyer and Hafedh Mili. Ibm websphere ilog jrules. In *Agile Business Rule Development*, pages 215–242. Springer, 2011.

[17] Tom Bracewell, Priya Narasimhan, and IDS Raytheon. A middleware for dependable distributed real-time systems. In *Joint Systems and Software Engineering Symposium, Falls Church, VA*, 2003.

[18] Bruce Buchanan, Georgia Sutherland, and Edward A Feigenbaum. *Heuristic DENDRAL: A program for generating explanatory hypotheses in organic chemistry*. Defense Technical Information Center, 1968.

[19] Bruce G Buchanan and Edward A Feigenbaum. Dendral and meta-dendral: Their applications dimension. *Artificial intelligence*, 11(1):5–24, 1978.

[20] K. Byoungjip. Comparison of the existing checkpoint systems. *Technical Report*, 5(1), 2005.

[21] Yilmaz Cengeloglu. Dynaclips (dynamic clips): A dynamic knowledge exchange tool for intelligent agents. In *Third Conference on CLIPS Proceedings (Electronic Version)*, page 353, 1994.

[22] Sharma Chakravarthy and Deepak Mishra. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering*, 14(1):1–26, 1994.

[23] Nianen Chen and Shangping Ren. Building a coordination framework to support behavior-based adaptive checkpointing for open distributed embedded systems. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 257b–257b. IEEE, 2007.

[24] Simon Courtenage. Specifying and detecting composite events in content-based publish/subscribe systems. In *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, pages 602–607. IEEE, 2002.

[25] Antonio Cunei and Jan Vitek. A new approach to real-time checkpointing. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 68–77. ACM, 2006.

[26] Olivia Das and C Murray Woodside. Evaluation of dependable layered systems with fault management architecture. 2002.

[27] Rene De La Briandais. File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 295–298. ACM, 1959.

[28] Franco Di Primio and Gerhard Brewka. Babylon—kernel system of an integrated environment for expert system development and operation. In *5th international workshop, Vol. 1 on Expert systems & their applications*, pages 573–583. Agence de l'Informatique, 1986.

[29] Senkul P. Dogru, A. H. and O. Kaya. Crtificial intelligence support for software engineering in the compositional era, in knowledge engineering for software development life cycles: Support technologies and applications (muthuramachandran, ed.). *Book Chapter*, 5(1), 2011.

[30] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software: Practice and Experience*, 21(8):757–785, 1991.

[31] Richard Duda, John Gaschnig, and Peter Hart. Model design in the prospector consultant system for mineral exploration. *Expert systems in the microelectronic age*, 1234:153–167, 1979.

[32] Jürgen Dunkel. On complex event processing for sensor networks. In *Autonomous Decentralized Systems, 2009. ISADS'09. International Symposium on*, pages 1–6. IEEE, 2009.

[33] Marc Eisenstadt and Mike Brayshaw. Build your own knowledge engineering toolkit. Technical report, Technical report, Human Cognition Research Laboratory, The Open University, UK, 1990.

[34] Haley Enterprise. Reasoning about rete++. *White paper available at: http://www. haley. com*, 1999.

[35] M Falla. Advances in safety critical systems: results and achievements from the dti/epsrc r&d programme. *Department of Trade and Industry (UK)*, 1997.

[36] Thomas Huining Feng and Edward A Lee. Incremental checkpointing with application to distributed discrete event simulation. In *Proceedings of the 38th conference on Winter simulation*, pages 1004–1011. Winter Simulation Conference, 2006.

[37] Charles L Forgy. Ops5 user's manual. Technical report, DTIC Document, 1981.

[38] Charles L Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19(1):17–37, 1982.

[39] Charles L Forgy. Rule-extended algorithmic language language guide. *Production Systems Technology*, 5001, 1991.

[40] Charles Lanny Forgy. The ops83 report. 1984.

[41] Service Availability Forum. The service availability forum solution for high availability, 2001.

[42] Service Availability Forum. The service availability forum: Hardware platform interface, 2002.

[43] Ernest Friedman-Hill et al. Jess, the rule engine for the java platform, 2003.

[44] Stella Gatziu and Klaus R Dittrich. Detecting composite events in active database systems using petri nets. In *Research Issues in Data Engineering, 1994. Active Database Systems. Proceedings Fourth International Workshop on*, pages 2–9. IEEE, 1994.

[45] Narain H Gehani, Hosagrahar V Jagadish, and Oded Shmueli. Composite event specification in active databases: Model & implementation. In *VLDB*, volume 92, pages 327–338. Citeseer, 1992.

[46] Andreas Geppert and Dimitrios Tombros. Event-based distributed workflow execution with eve. In *Middleware'98*, pages 427–442. Springer, 1998.

[47] Robert E Gruber, Balachander Krishnamurthy, and Euthimios Panagos. The architecture of the ready event notification service. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 0108–0108. IEEE Computer Society, 1999.

[48] Frederick Hayes-Roth, Donald A Waterman, and Douglas B Lenat. Building expert systems. *Teknowledge Series in Knowledge Engineering, Reading: Addison-Wesley, 1983, edited by Hayes-Roth, Frederick; Waterman, Donald A.; Lenat, Douglas B.*, 1, 1983.

[49] Junyoung Heo, Sangho Yi, Yookun Cho, Jiman Hong, and Sung Y Shin. Space-efficient page-level incremental checkpointing. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 1558–1562. ACM, 2005.

[50] Dan Hildebrand. An architectural overview of qnx. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, 1992.

[51] Zhu Huaji, Wu Huarui, and Sun Xiang. Research on the ontology-based complex event processing engine of rfid technology for agricultural products. In *Artificial Intelligence and Computational Intelligence, 2009. AICI'09. International Conference on*, volume 1, pages 328–333. IEEE, 2009.

[52] François Félix Ingrand, Raja Chatila, Rachid Alami, and Frédéric Robert. Prs: A high level supervision and control language for autonomous mobile robots. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 1, pages 43–49. IEEE, 1996.

[53] Hiroaki Inoue, Takashi Takenaka, and Masato Motomura. 20gbps c-based complex event processing. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 97–102. IEEE, 2011.

[54] Norman McWhirter Jim Darroch. Demonstrating software reliability. Embedded System Conference, April 2006.

[55] D. Kalinsky. Principles of high availability embedded systems design. *Embedded World 2006 Conference*, 2006.

[56] David Kalinsky. Design patterns for high availability-it is possible to achieve five-nines reliability with everyday commercial-quality hardware and software. the key is the way in which these components are. *Embedded Systems Programming*, 15(8):24–33, 2002.

[57] David Kalinsky. Architecture of safety-critical systems. *Embedded Systems Programming*, pages 14–25, 2005.

[58] Mark Kantrowitz. Flex toolkit hybrid expert systems. http://www.lpa.co.uk/flx.htm, last visited on December 2013.

[59] PC Kappa. Users guide. *Intellicorp Inc*, 1992.

[60] O. Kaya, L. Alkislar, R. Ergun, C. Togay, and A. Dogru. A feature based domain model for high availability reference architectures. Technical report, METU, CENG, 2007.

[61] O. Kaya, L. Alkislar, R. Ergun, C. Togay, and AH Dogru. Fault avoidance for mission critical systems. In *The Eleventh SDPS Transdisciplinary Conference on Integrated Systems, Design and Process Science (IDPT 08)*, 2008.

[62] Ozgur Kaya and Dogru Ali H. Designing of an adaptable checkpointing system for real-time applications. *The 16th International Conference on Transformative Science, Engineering, and Business Innovation*, 2011.

[63] Ozgur Kaya, Seyedsasan Hashemikhabir, Cengiz Togay, and Ali Hikmet Dogru. A rule-based domain specific language for fault management. *Journal of Integrated Design and Process Science*, 14(3):13–23, 2010.

[64] Byoung-Jip Kim. Comparison of the existing checkpoint systems. Technical report, IBM Watson Center, 2005.

[65] Ding Kun, Zhang Xiaoyi, and Deng Bo. C-nmsp: a high performance network management service platform for complex event processing. In *Future Generation Communication and Networking, 2008. FGCN'08. Second International Conference on*, volume 1, pages 7–10. IEEE, 2008.

[66] Yong H Lee and Suk I Yoo. A rete-based integration of forward and backward chaining inferences. In *Intelligent Control, 1995., Proceedings of the 1995 IEEE International Symposium on*, pages 611–616. IEEE, 1995.

[67] C-CJ Li and W Kent Fuchs. Catch-compiler-assisted techniques for checkpointing. In *Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium*, pages 74–81. IEEE, 1990.

[68] Jie LI and Shi-tuan SHEN. Research on the algorithm of avionic device fault diagnosis based on fuzzy expert system. *Chinese Journal of Aeronautics*, 20(3):223–229, 2007.

[69] Yuanping Li, Jun Wang, Ling Feng, and Wenwei Xue. Accelerating sequence event detection through condensed composition. In *Ubiquitous Information Technologies and Applications (CUTE), 2010 Proceedings of the 5th International Conference on*, pages 1–6. IEEE, 2010.

[70] Di Liu, Tao Gu, and Jiang-Ping Xue. Rule engine based on improvement rete algorithm. In *Apperceiving Computing and Intelligence Analysis (ICACIA), 2010 International Conference on*, pages 346–349. IEEE, 2010.

[71] Frank Lopez. *The parallel production system.* PhD thesis, University of Illinois, 1987.

[72] lorenzo lupini Lorenzo Fasanelli, Massimo Quagliani. Fault detection in embedded systems. Embedded System Conference, April 2008.

[73] David Luckham. *The power of events: An introduction to complex event processing in distributed enterprise systems.* Springer, 2008.

[74] M Maloof and Krys J Kochut. Modifying rete to reason temporally. In *Tools with Artificial Intelligence, 1993. TAI'93. Proceedings., Fifth International Conference on*, pages 472–473. IEEE, 1993.

[75] Masoud Mansouri-Samani and Morris Sloman. Gem: A generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96, 1997.

[76] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[77] Daniel P Miranker. *TREAT: A Better Match Algorithm for AI Production Systems; Long Version*. University of Texas at Austin, 1987.

[78] Douglas Moreto and Markus Endler. Evaluating composite events using shared trees. In *Software, IEE Proceedings-*, volume 148, pages 1–10. IET, 2001.

[79] Shiv Nagarajan. Designing systems for high availability. Embedded System Conference, April 2006.

[80] P Neira, Laurent Lefevre, and Rafael M Gasca. High availability support for the design of stateful networking equipments. In *Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on*, pages 8–pp. IEEE, 2006.

[81] Jürgen Neises. Benefit evaluation of high-availability middleware. In *Service Availability*, pages 73–85. Springer, 2005.

[82] Esko Nuutila, Juha Kuusela, Markku Tamminen, J Veilhti, and Jari Arkko. Xc-a language for embedded rule based systems. *ACM SIGPLAN Notices*, 22(9):23–32, 1987.

[83] Hiroyuki Okamura, Yuki Nishimura, and Tadashi Dohi. A dynamic checkpointing scheme based on reinforcement learning. In *Dependable Computing, 2004. Proceedings. 10th IEEE Pacific Rim International Symposium on*, pages 151–158. IEEE, 2004.

[84] Adam J Oliner, Larry Rudolph, and Ramendra K Sahoo. Cooperative checkpointing: a robust approach to large-scale systems reliability. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 14–23. ACM, 2006.

[85] Rudolph L. Sahoo R. Oliner, A. Cooperative checkpointing theory. *Cooperative checkpointing theory, 20th International Parallel and Distributed Processing Symposium*, 2006.

[86] SILICON VALLEY ONE. Ops-2000 reference manual. http://www.siliconvalleyone.com/founder/ops2000/ops_ref21.pdf, last visited on November 2013.

[87] Bob Orchard. Fuzzyclips version 6.10 d user's guide. *National Research Council of Canada*, 2004.

[88] RJ Patton. Fault detection and diagnosis in aerospace systems using analytical redundancy. *Computing & Control Engineering Journal*, 2(3):127–136, 1991.

[89] Ryan Paul. Designing and implementing a domain-specific language. *Linux Journal*, 2005(135):7, 2005.

[90] Peter R Pietzuch, Brian Shand, and Jean Bacon. Composite event detection as a generic middleware extension. *Network, IEEE*, 18(1):44–55, 2004.

[91] James S Plank, Yuqun Chen, Kai Li, Micah Beck, and Gerry Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Software-Practice and Experience*, 29(2):125–142, 1999.

[92] James S Plank, Kai Li, and Michael A Puening. Diskless checkpointing. *Parallel and Distributed Systems, IEEE Transactions on*, 9(10):972–986, 1998.

[93] J.S. Plank, M. Beck, G. Kingsley, K. Li, et al. Libckpt: Transparent checkpointing under unix. 1994.

[94] Mark Proctor, Michael Neale, Peter Lin, and Michael Frandsen. Drools documentation. *JBoss. org, Tech. Rep*, 2008.

[95] L.L. Pullum. *Software Fault Tolerance Techniques and Implementation*. Artech House computing library. Artech House, 2001.

[96] Han Qing and John Fogelin. High availability design for embedded systems. In *2001 International IC Conference & Exhibition*, pages 502–509.

[97] A. Ranganathan and S.J. Upadhyaya. Simulation analysis of a dynamic checkpointing strategy for real-time systems. In *Simulation Symposium, 1994., 27th Annual*, pages 181–187. IEEE, 1994.

[98] Gary Riley. Clips: An expert system building tool. In *NASA, Washington, Technology 2001: The Second National Technology Transfer Conference and Exposition,*, volume 2, 1991.

[99] E Roman. E.a survey of checkpoint/restart implementations. berkeley lab technical report. Technical report, LBNL, 2002.

[100] Tony Romero and Sean O'Brien. The past, present and future of high availability systems. *Performance Technologies*, 2003.

[101] Omran Saleh and Kai-Uwe Sattler. Distributed complex event processing in sensor networks. In *Mobile Data Management (MDM), 2013 IEEE 14th International Conference on*, volume 2, pages 23–26. IEEE, 2013.

[102] Kay-Uwe Schmidt, Roland Stühmer, and Ljiljana Stojanovic. Blending complex event processing with the rete algorithm. In *iCEP2008: First International Workshop on Complex Event Processing for the Future Internet–colocated with the Future Internet Symposium, Vienna (Austria)*. Citeseer, 2008.

[103] Martin L. Shooman. *Reliability of Computer Systems and Networks: Fault Tolerance,Analysis,and Design*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[104] Edward Shortliffe. *Computer-based medical consultations: MYCIN*. Elsevier, 2012.

[105] Julian Smart. wxclips, 2003.

[106] Su Myat Marlar Soe and May Paing Paing Zaw. Design and implementation of rule-based expert system for fault management. *Proceedings of World Academy of Science: Engineering & Technology*, 48, 2008.

[107] Edgar Sommer, Werner Emde, Jorg-Uwe Kietz, and Stefan Wrobel. Mobal 4.1 b9 user guide, 1996.

[108] Coskun Sonmez. Uzman sistemlerin programlanması, Nisan 2006.

[109] Adam Stevenson. Rapid and transparent failover for high-availability systems. CompactPCI Systems, May - June 2003.

[110] Neil R Storey. *Safety critical computer systems*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[111] Volker Strumpen et al. Compiler technology for portable checkpoints. *submitted for publication (http://theory. lcs. mit. edu/strumpen/porch. ps. gz)*, 477:481–484, 1998.

[112] Eric Summers. Es: A public domain expert system. *Byte*, 15(10):289–292, 1990.

[113] Hairong Sun, James J Han, and Haim Levendel. Availability requirement for a fault-management server in high-availability communication systems. *Reliability, IEEE Transactions on*, 52(2):238–244, 2003.

[114] M. Tekkalmaz, E. Gürler, and M. Dursun. Görev kritik ve gömülü sistemler için t 5 d uyumlu bir hata yönetimi altyapısı tasarımı ve gerçeklemesi, 2009.

[115] Marvin M Theimer, Keith A Lantz, and David R Cheriton. *Preemptable remote execution facilities for the V-system*, volume 19. ACM, 1985.

[116] Peter Triantafillou. High availability is not enough [distributed systems]. In *Management of Replicated Data, 1992., Second Workshop on the*, pages 40–43. IEEE, 1992.

[117] Kishor S Trivedi, Ranjith Vasireddy, D Trindade, Swami Nathan, and Rick Castro. Modeling high availability. In *Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on*, pages 154–164. IEEE, 2006.

[118] Andreas Ulbrich, Gero Mühl, Torben Weis, and Kurt Geihs. Programming abstractions for content-based publish/subscribe in object-oriented languages. In *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, pages 1538–1557. Springer, 2004.

[119] Steve Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, 1997.

[120] Karen Walzer, Tino Breddin, and Matthias Groch. Relative temporal constraints in the rete algorithm for complex event detection. In *Proceedings of the second international conference on Distributed event-based systems*, pages 147–155. ACM, 2008.

[121] Karen Walzer, Alexander Schill, and Alexander Löser. Temporal constraints for rule-based event processing. In *Proceedings of the ACM first Ph. D. workshop in CIKM*, pages 93–100. ACM, 2007.

[122] Fusheng Wang, Shaorong Liu, Peiya Liu, and Yijian Bai. Bridging physical and virtual worlds: complex event processing for rfid data streams. In *Advances in Database Technology-EDBT 2006*, pages 588–607. Springer, 2006.

[123] YH Wang, K Cao, and XM Zhang. Complex event processing over distributed probabilistic event streams. *Computers & Mathematics with Applications*, 66(10):1808–1821, 2013.

[124] Yongheng Wang and Kening Cao. Context-aware complex event processing for event cloud in internet of things. In *Wireless Communications & Signal Processing (WCSP), 2012 International Conference on*, pages 1–6. IEEE, 2012.

[125] Wikipedia. Rule-based systems and identification trees: Introduction to rule-based systems. http:en.wikipedia.orgwikiDomain-specific_language, last visited on February 2014.

[126] R Wirls-Brock. Designing for recovery [software design]. *Software, IEEE*, 23(4):11–13, 2006.

[127] Kun Wu, Jianping Wu, and Ke Xu. A novel architecture in high availability (ha) systems. In *TENCON 2004. 2004 IEEE Region 10 Conference*, pages 262–265. IEEE, 2004.

[128] Ding Xiao, Yi Tong, Haitao Yang, and Mudan Cao. The improvement for rete algorithm. In *Information Science and Engineering (ICISE), 2009 1st International Conference on*, pages 5222–5225. IEEE, 2009.

[129] Jin Xingyi, Lee Xiaodong, Kong Ning, and Yan Baoping. Efficient complex event processing over rfid data stream. In *Computer and Information Science, 2008. ICIS 08. Seventh IEEE/ACIS International Conference on*, pages 75–81. IEEE, 2008.

[130] Dongdai Zhou, Yifan Fu, Shaochun Zhong, and Ruiqing Zhao. The rete algorithm improvement and implementation. In *Proceedings of the 2008 International Conference on Information Management, Innovation Management and Industrial Engineering-Volume 01*, pages 426–429. IEEE Computer Society, 2008.

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Kaya, Özgür
Nationality: Turkish (TC)
Date and Place of Birth: 1975, Trabzon
Marital Status: Married
Phone: +90 312 210 5548
Fax: +90 312 210 5544

EDUCATION

Degree	Institution	Year of Graduation
M.S.	METU Computer Engineering	2006
B.S.	Karadeniz Technical University	2001
High School	Affan Kitapçıoğlu High School	1993

PROFESSIONAL EXPERIENCE

Year	Place	Enrollment
2002-	METU Computer Engineering	Research Assistant
2005-	METU Computer Engineering	System Administrator
2010-2013	SOSoft Information Systems	Biometric Security Device
2005-2011	METU CENG IDEA Program	Teaching Assistant
2010-2011	INVICTA	Common Criteria Eal4+ Certification
2009-2013	ASELSAN	Domain Specific Languages
2007-2008	ASELSAN	FMS for Mission Critical Systems

BOOK CHAPTER

Knowledge Engineering for Software Development Life Cycles:"Modern Approaches to Software Engineering in the Compositional Era" [with Ali Dogru & Pinar Senkul]

PUBLICATIONS

International Conference Publications

(Submitted) Kaya, O., Suloglu, S. and Dogru, A.H., "Fast Evaluation Approach for a Rule-based Fault Monitoring Language," submitted to the 18th International Conference on Software Security and Reliability (SERE), San Fransisco, California, June 30 – July 2, 2014.

Ozgur Kaya and Ali Doğru, "Further Specification of Basic Event Sequences: A Processing View", In 18th International Conference on Society for Design and Process Science (SDPS 2013), October 27-31, 2013, Sao Paulo, Brazil.

Ozgur Kaya and Ali H. Dogru, Designing of an Adaptable Checkpoint System For Real Time Applications, "The Society for Design and Process Science", SDPS 2011, June 12-16, Jeju, South Korea, 2011.

Ozgur Kaya, Seyed Sasan Hashemikhabir, Cengiz Togay and Ali H. Dogru, A Rule-Based Domain Specific Language for Fault Management, "Transformative System Conference", SDPS 2010, June 6-11, Dallas, USA, 2010.

Online Education Experiences: Information Technologies Certificate Program at METU. Yukselturk, E., Yazıcı, A., Kaya, O., and Sacan, A. (2010). ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE).

Ozgur Kaya, Levent Alkislar, Reyhan Ergun, Cengiz Togay, Ali H. Dogru., Fault Avoidance for Mission Critical Systems,"Eleventh SDPS Transdisciplinary Conference on Integrated Systems, Design and Process Science (IDPT 08)",June 1-6, 2008, Taichung, Taiwan.

National Conference Publications

Metin Tekkalmaz, Ozgur Kaya, Mustafa Dursun, Tuçe Sarı Tekkalmaz, Ali Doğru Görev Kritik ve Gömülü Sistemler için Hata Yönetimi Kılavuz Mimarisi: T5D,"2. Ulusal Yazilim Mimarisi Konferansi '08",. Izmir, Turkiye, September 2008.