Adaptive Root Cause Analysis and Diagnosis

by

Qin Zhu
B.Sc., Nanjing University of Aeronautics and Astronautics, 1989
M.Sc., University of Victoria, 2002

A Dissertation Submitted in Partial Fulfillment
of the Requirements for the Degree of

Doctor of Philosophy

in the Department of Computer Science

Supervisory Committee

Adaptive Root Cause Analysis and Diagnosis

by

Qin Zhu
B.Sc., Nanjing University of Aeronautics and Astronautics, 1989
M.Sc., University of Victoria, 2002

Supervisory Committee

Dr. Hausi A. Müller, Department of Computer Science, University of Victoria
Supervisor

Dr. William W. Wadge, Department of Computer Science, University of Victoria
Departmental Member

Dr. Jens H. Weber, Department of Computer Science, University of Victoria
Departmental Member

Dr. Issa Traoré, Department of Electrical and Computer Engineering,
University of Victoria
Outside Member

Abstract

Supervisory Committee
Dr. Hausi A. Müller, Department of Computer Science, University of Victoria
Supervisor

Dr. William W. Wadge, Department of Computer Science, University of Victoria
Departmental Member

Dr. Jens H. Weber, Department of Computer Science, University of Victoria
Departmental Member

Dr. Issa Traoré, Department of Electrical and Computer Engineering,
University of Victoria
Outside Member

In this dissertation we describe the *event processing autonomic computing reference architecture* (EPACRA), an innovative reference architecture that solves many important problems related to adaptive *root cause analysis and diagnosis* (RCAD). Along with the research progress for defining EPACRA, we also identified a set of autonomic computing architecture patterns and proposed a new information seeking model called net-casting model.

EPACRA is important because today, root cause analysis and diagnosis (RCAD) in enterprise systems is still largely performed manually by experienced system administrators. The goal of this research is to characterize, simplify, improve, and automate RCAD processes to ease selected tasks for system administrators and end-users. Research on RCAD processes involves three domains: (1) autonomic computing architecture patterns, (2) information seeking models, and (3) complex event processing (CEP) technologies. These domains as well as existing technologies and standards contribute to the synthesized knowledge of this dissertation.

To minimize human involvement in RCAD, we investigated architecture patterns to be utilized in RCAD processes. We identified a set of autonomic computing architecture

patterns and analyzed the interactions among the feedback loops in these individual architecture patterns and how the autonomic elements interact with each other. By illustrating the architecture patterns, we recognized ambiguity in the aggregator-escalator-peer pattern. This problem has been solved by adding a new architecture pattern, namely the chain-of-monitors pattern, to the lattice of autonomic computing architecture patterns.

To facilitate the autonomic information seeking process, we developed the net-casting information seeking model. After identifying the commonalities among three traditional information seeking models, we defined the net-casting model as a five stage process and then tailored it to describe our automated RCAD process.

One of the main contributions of this dissertation is an innovative autonomic computing reference architecture called *event processing autonomic computing reference architecture* (EPACRA). This reference architecture is based on (1) complex event processing (CEP) concepts, (2) autonomic computing architecture patterns, (3) real use-case workflows, and (4) our net-casting information seeking model. This reference architecture can be leveraged to relieve the system administrator's burden of routinely performing RCAD tasks in a heterogeneous environment. EPACRA can be viewed as a variant of the IBM ACRA model—extended with CEP to deal with large event clouds in real-time environments. In the middle layer of the reference model, EPACRA introduces an innovative design referred to as *use-case-unit*—a use case is the scenario of an RCAD process initiated by a symptom—*event processing network* (EPN) for RCAD. Each use-case-unit EPN reflects our automation approach, including identification of events from the use cases and classifying those events into event types. Apart from defining individual *event processing agents* (EPAs) to process the different types of events, dynamically constructing use-case unit EPNs is also an innovative approach which may lead to fully autonomic RCAD systems in the future.

Finally, this dissertation presents a case study for EPACRA. As a case study we use a prototype of a Web application intrusion detection tool to demonstrate the autonomic

mechanisms of our RCAD process. Specifically, this tool recognizes two types of malicious attacks on web application systems and then takes actions to prevent intrusion attempts. This case study validates both our chain-of-monitors autonomic architecture pattern and our net-casting model. It also validates our use-case-unit EPN approach as an innovative approach to realizing RCAD workflows. Hopefully, this research platform will be beneficial for other RCAD projects and researchers with similar interests and goals.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I am especially grateful to my supervisor, Hausi Müller, for his support, guidance and patience throughout this long journey. I would like to thank all my friends and the members of the Rigi group at the University of Victoria, who contributed significantly to my appreciation and understanding of autonomic computing and other related domains of knowledge. In particular, I acknowledge the excellent work of Lei Lin, who has helped me with the case study. Finally, I would like to acknowledge the support of University of Victoria, CA Canada Inc., IBM Canada, the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Consortium for Software Engineering Research (CSER).

Dedication

For my parents

Chapter 1 Introduction

1.1 Research Overview

The goal of this research is to investigate concepts, methods, and tools for root cause analysis and diagnosis (RCAD). A *root cause* is the most basic cause (or fault) that can reasonably be identified and that management has control to fix [PaBu88]. In the context of software systems, root cause or fault is the basic cause of an error or a failure. In the software literature [Musa04] fault is defined as a bug or problem in the system (the cause), error as the deviation of the system from its expected state (possibly not observable by an operator or a monitoring process) and failure as the deviation of the system's observable behaviour from the expected or specified behaviour. RCAD in this dissertation refers to the task of identifying root causes in enterprise system management. So far, there is no single technology or tool to tackle the complexity of RCAD problems effectively. Instead, an innovative and integrated platform is needed with a combination of technologies and tools to help users perform root cause analysis and diagnostic capabilities. Thus, our goal is to develop innovative analysis methods, techniques, and tools to improve root cause analysis and diagnosis.

In the real world, enterprise applications often consist of components that were developed independently and utilize heterogeneous event logging and diagnostic techniques, as well as diverse log management and system monitoring policies. In this context, the main challenge is to devise software engineering techniques that allow analysis, re-engineering, amalgamation, and integration of heterogeneous logging, monitoring and diagnosis processes, so that such software systems may still be monitored, audited, and diagnosed in an effective and efficient manner.

One of the central problems in RCAD is the management of events. Luckham pointed out the following challenges with respect to event management in networks [Luck05].

We submit that these challenges are readily applicable to other IT components in general and RCAD in particular:

- Event logging can become very large and difficult to handle in real-time.
- Event identifying tools for sets of related events are required (especially when an event storm happens).
- Causal tracking is essential.
- Predictive monitoring is beyond the state of the art.

Filtering and processing event streams out of an event cloud in real time pose huge challenges to root cause analysis and diagnosis tools. Complex event processing (CEP) is able to meet such challenges by allowing users to specify the events that are of interest to them at any instance of time. Different kinds of events can be specified and monitored simultaneously. CEP provides techniques for defining and utilizing the relationship between events. One of the techniques lets users define their own events as patterns of events in their computing system. This is exactly the technology that we require to process large amounts of events to be able to determine and track the causality among the events.

However, CEP alone is rarely the answer for pinpointing the root cause of a problem. One typical method for RCAD is to use *cause-and-effect diagrams* (also called *fishbone diagrams*). The effects of root causes are often manifested in alerts or warning signs. A system administrator can trace the causes by checking all the endings of a fishbone diagram, as depicted in Figure 1-1. Some causes may contribute to multiple effects and thus the presence of one effect alerts the administrator to check for the presence of others. When a particular set of effects (called syndrome) occurs, the administrator can narrow the correlated cause (or set of causes) to the root cause(s).

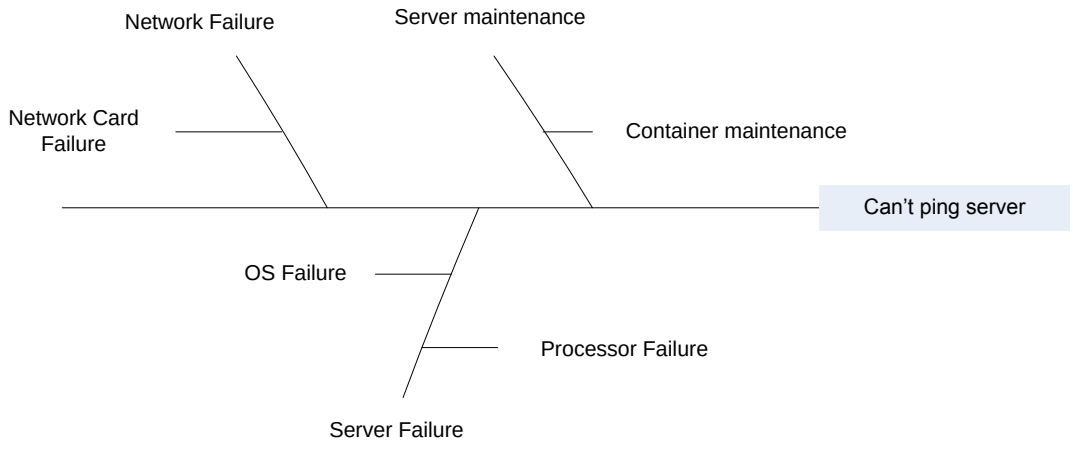


Figure 1-1: A cause-and-effect diagram

Besides a fishbone diagram, another method to solve this problem is following the RCAD process described by a scenario or workflow as depicted in Figure 6-2. The scenario describes the kind of information required for diagnosis when a certain symptom manifests itself. Frequently, the administrator cannot exactly diagnose the problem with the available information. He/she needs to acquire more information from potential root cause sources at different times and locations. This inquiry-based approach, which collects more information when the need arises, is different from the collecting-filtering-analysing approach which is typical for CEP.

Currently, both RCAD fishbone diagrams and scenarios are interpreted and executed by human operators. The goal of this research is to leverage existing technologies and standards to automate the RCAD processes. In terms of existing technologies, we are investigating CEP technologies and applying a CEP engine, Esper, to construct our event processing network (EPN). With respect to existing standards, we strive to make the automation processes more extendable and adaptable.

One key goal of this dissertation is to investigate a variety of models, architectures, and workflows for diagnosis, root cause analysis, and self-management. Models and architectures for RCAD originate in a wide range of fields including robotics, control

engineering, and software engineering. They can be applied to different scales of robotic systems, control systems and software systems, with different levels of constraints and flexibilities. Our attention focuses on three seminal three-layer reference architectures for self-management that particularly influenced our work: (1) Gat's robotics-inspired Atlantis architecture (1997) [Gat97], (2) IBM's ACRA: autonomic computing reference architecture (2006) [IBM06], and (3) the software engineering-inspired Kramer & Magee architecture (2007) [KrMa07]. Based on the three-layer architectures, we investigated and designed a novel reference architecture for RCAD called *event processing autonomic computing reference architecture* (EPACRA).

Another research approach is to automate RCAD using feedback loops. Usually, an alert or a warning sign detected by the anomaly detection mechanism could indicate a possible fault in a system. However, that system could be completely "normal" with respect to functionality (e.g., in a use case provided by CA: SystemX, which is perceived to be "slow", which usually indicates some abnormality, is in fact "normal" because such slowness is caused by a midday payroll job that is running on SystemX, and therefore no treatment is required). Under circumstances such as these, further investigation, guided by the scientific method (i.e., hypothesis, experiment and observation), is often necessary before any kind of treatment can be performed. A complete process of the scientific method forms a natural feedback loop. Instead, of the treatment being performed by human operators, the mechanism of the autonomic computing (defined by EPACRA) enables the automation of the root cause analysis and diagnosis processes that follow the scientific method. The design of EPACRA includes layered feedback loops which are consistent with the ACRA model. It consists of multiple loops which reflects the inference process of the scientific method and provides more flexibility, dynamicity and robustness to the system.

As a result, EPACRA allows system administrator to develop adaptive mechanisms to evolve diagnosis, root cause analysis, and self-management capabilities. For instance, the top level *use-case manager* (cf. Chapter 10), either a human or a machine, can prioritize the jobs of *use-case-unit EPNs* (cf. Chapter 9). The key to adaptability of self-managing

systems is their embedded control loops. Feedback loops can operate independently, form a coherent hierarchy, or interact collaboratively towards a common goal. Moreover, the control loops of a self-adaptive system evolve over the different stages of the system's life cycle. For example, control loops are introduced or evolved during the system's requirements and design phase, at its acceptance time (e.g., to satisfy validation and verification requirements), or during its long-term operation to meet new or evolving requirements.

We also developed a research platform for computer assisted adaptation for root cause analysis and diagnosis. Through a case study (i.e., a research prototype implementation), we demonstrate how this new architectural model EPACRA, including its CEP components, can be applied based on a *use case workflow***.**

This dissertation grew out of an industrial collaborative research project entitled "Logging, Monitoring and Diagnosis Systems for Enterprise Software Application" directed by S. Mankovskii (CA Canada, Inc.), H. A. Müller (University of Victoria), K. Kontogiannis (University of Waterloo), J. Mylopoulos (University of Toronto), and K. Wong (University of Alberta). The industrial partner in this NSERC Collaborative Research and Development project is CA Canada, Inc.

1.2 Research Motivations

There is a wide range of IT management tools for administrators available from commercial tools such as CA's Wily Introscope to open source tools such as Glassbox. They track applications on every machine in a network and provide statistics and summary data and render them into a spectrum of colourful indicators on monitoring dashboards. By interpreting those indicators, an administrator makes sense of all this information in real time when a variety of alert warning signs or alarms emerge from different layers of the enterprise system including application layer, collaboration layer,

middle layer and network layer. These indicators appear at different time intervals and different volumes.

Root cause analysis in enterprise systems is still largely performed manually by experienced system administrators. For example, the network layer monitoring tools log and record both network traffic using special kinds of instrumentation, which typically include TCP packets, warnings/alerts, and performance measurement data of network components such as routers and servers. The event logs are fed into viewing tools that provide traffic statistics and warnings of various problems. All these tools give system administrators a primitive way of keeping track of how the network layer is behaving and detecting failure at various spots all over the network. However, most tools contain little intelligence to tell administrators what the root cause is, and how to resolve the problems. Network administrators have to figure out problems from the event logs and statistical views of event traffic by applying their experience and intuition.

Thus, the diagnostic intelligence that is needed to keep the enterprise systems running resides in the system administrators' heads and not yet in the IT management tools. Hence, another key objective of this research is to help administrators of highly complex, distributed enterprise systems to process events, perform root cause analysis, and codify predictive diagnosis.

Here are the key problems motivating our research:

- **Problem I: Massive event clouds: information overload, hard real-time problems.** IT systems are widespread across large enterprises and generate many events that flow through the enterprise system layers. The events feed other applications or services which in turn generate new events. There are many event-clouds that hang around within such an enterprise and its RCAD processes.

- **Problem II: Selective event pattern sensing and causal tracking, tracing and correlation.** Because of such event-clouds, the event-flow of an enterprise IT system is not transparent and becomes difficult to understand. The simplest events

are traceable, but more complex events (which consist of multiple, unrelated simple events) are hard to keep track of. To tackle this problem and to make more use of complex events, we use CEP to view and react to complex events in real time.

- **Problem III: Situational awareness problems due to uncertainty in environment and users' need for adaptive RCAD.** With CEP it is possible to act in real time and make better use of the already available events in an enterprise. However, the systems that system administrators managed are rarely static—they are constantly evolving. Only systems designed to be adaptive are able to respond to the changes either in their own state or in their managed systems. Thus, adaptability is critical for RCAD architecture designs.

- **Problem IV: Operator tunnel vision with respect to event correlation.** In root cause analysis and diagnosis processes, human administrators are inclined to focus on one point at a time rather than have a broad set of the many possibilities in the field of investigation. Thus, we need to manage tunnel vision by expanding the correlation horizon of operators involved in RCAD processes.

- **Problem V: Latency with respect to data, insight, decision or action.** When a problem or opportunity arises it should be noticed right away, in real time, to make sure the right action can be taken as soon as possible or at the right time. Otherwise, the opportunities of revealing possible problems could vanish quickly.

- **Problem VI: Limited support for diagnosis automation.** The diagnosis (inference) is an iterative / recursive / interactive process that is accomplished by many rounds of hypothesis-experiment-observation. The process is often done manually. Potentially, many fishbone diagrams or workflow/scenarios can be processed by machines in parallel.

- **Problem VII: Limited mechanisms for adaptation and learning.** Diagnoses should not only be based upon the current situation and events, but also historical contexts and events. A knowledge base such as symptom and syndrome databases will boost the adaptation and learning capabilities of RCAD tools.

1.3 Research Methodologies

To validate research, it is essential to articulate the research methodology. A methodology refers to the rationale and the philosophical assumptions that underlie a particular study relative to the scientific method. Creswell groups research methodologies into four categories based on their philosophical stances: *positivist (quantitative), interpretive (qualitative), ideological,* and *pragmatic (mixed-method)* [Cres02].

Easterbrook et al. also discuss these four types of methodologies [ESSD07]. Table 1 **Error! Reference source not found.**summarizes and compares these four methodology types according to the following aspects: statement (what these types of methodologies claim), characteristics (how these types of methodologies differ from others), applied fields (to which domains these types of methodologies usually apply), preferred methods (what research methods these types of methodologies prefer), and associated scientific methods (which scientific methods these types of methodologies are associated with).

Table 1: Comparison of four types of methodologies [ESSD07]

Philosophical stance	**Positivism**	**Constructivism** [KlMy99]	**Critical Theory** [Calh95]	**Pragmatic** [Mena97]
Statement	All knowledge must be based on logical inference from a set of basic observable facts.	Scientific knowledge cannot be separated from its human context.	Scientific knowledge is judged by its ability to free people from restrictive systems of thought.	All knowledge is approximate and incomplete, and its value depends on the methods by which it was obtained.
Characteristics	Scientific knowledge is built up incrementally	The researcher should concentrate less on verifying theories, and more	Research is a political act, because knowledge empowers different	Knowledge is judged by how useful it is for solving practical

	from verifiable observations, and inferences based on them.	on understanding how different people make sense of the world, and how they assign meaning to actions.	groups within society, or entrenches existing power structures.	problems. Put simply, truth is whatever works at the time.
Applied fields	While still dominating the natural sciences, most positivists today are considered as post-positivists (they tend to accept the idea that we increase our confidence in a theory each time we fail to refute it).	This stance is often adopted in the social sciences, where positivist approaches have little to say about the richness of social interactions.	In sociology, critical theory is most closely associated with Marxist and feminist studies, etc. In software engineering, it includes research that actively seeks to challenge existing perceptions about software practice, such as the open source movement.	An engineering approach is adopted to research. It values practical knowledge over abstract knowledge, and uses whatever methods are appropriate to obtain it.
Preferred methods	Positivists prefer methods that start with precise theories from which verifiable hypotheses can be extracted and tested in isolation.	Constructivists prefer methods that collect rich qualitative data about human activities, from which local theories might emerge.	Critical theorists prefer participatory approaches in which the groups they are trying to help are engaged in the research, including helping to set its goals.	Pragmatists use any available methods.
Associated scientific methods	Positivism is most closely associated with the controlled experiment. Survey research and case studies are also frequently conducted with a positivist stance.	Constructivism is most closely associated with ethnographies, although constructivists often use exploratory case studies and survey research, too.	While most closely associated with action research, critical theorists often use case studies to draw attention to things that need changing.	Mixed methods research is strongly preferred, where several methods are used to shed light on the issue being studied.

Denning states "computing is a natural science [1][Denn07]." "The old definition of computer science—the study of phenomena surrounding computers—is now obsolete. Computing is the study of natural and artificial information processes." Therefore, just as

[1] Traditionally, computer science is not considered natural science.

in other arenas of natural science, *positivism* is considered the dominant methodology in computer science research.

Since some parts of our research follow an engineering approach, especially in the phases of identifying events from use cases and classifying those events into event types, we regard the methodology that we adopted in this dissertation to fall into the *pragmatic* or *mixed-method* category.

Science seeks to improve our understanding (i.e., scientific truth) of the world. Theory means "the best explanation for the available evidence [Weed07]." A theory of scientific truth must stand up to empirical scrutiny. Sometimes a theory must be thrown out in the face of new findings. Here are some definitions related to the concept of theory [East07]:

- **Model** is an abstract representation of a phenomenon or set of related phenomena. Although some details are included, others are excluded.
- **Theory** is a set of statements that explain a set of phenomena. Ideally, the theory has predictive power too (i.e., theory's *generality*).
- **Hypothesis** is a testable statement derived from a theory (i.e., a hypothesis is not a theory). Testing a hypothesis alone is pointless (i.e., a single flawed study), unless it builds evidence for a clearly stated theory.

So, what is *validation*? In engineering, *validation* and *verification* confirm that a product or service meets the needs of its users. It is the process of checking that a product, service, or system meets specifications and that it fulfills its intended purpose.[2]

In software, *validation* and *verification* check that a software system meets specifications and fulfills its intended purpose. It is normally part of the software testing

[2] Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Validation_and_verification

process of a project.[3] Software *verification* provides objective evidence that the design outputs of a particular phase of the software development life cycle meet all of the specified requirements for that phase by checking for consistency, completeness, and correctness of the software and its supporting documentation. *Validation*, on the other hand, is the confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled.[4] Figure 1-2 depicts the difference between validation and verification.

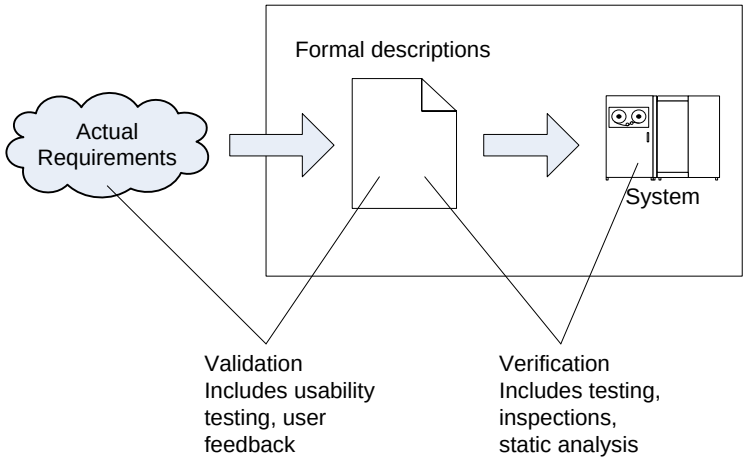


Figure 1-2: The difference between validation and verification [Dsso10]

Validation of engineering research traditionally follows the scientific inquiry tradition. This tradition demands "formal, rigorous and quantitative validation" [BaCa90], which is based primarily on logical induction and / or deduction. Since much engineering research is based on mathematical modeling, this kind of validation has worked and still works very well. However, there are other areas of engineering research that rely on subjective statements as well as mathematical modeling, which makes "formal, rigorous and quantitative" validation problematic [PEBA00]. One such area is

[3] Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Verification_and_Validation_(software)

[4] R. Jetley and B. Chelf, "Diagnosing Medical Device Software Defects Using Static Analysis", http://www.mddionline.com/article/diagnosing-medical-device-software-defects-using-static-analysis, last accessed 2010.

that of design methods within the field of engineering design. So, how shall we validate design research in general, and design methods in particular? Pedersen et. al. define "scientific knowledge within the field of engineering design as socially justifiable belief according to the *relativistic school* of *epistemology*" [PEBA00]. They do so since the open nature of design method synthesis, where new knowledge is associated with heuristics and non-precise representations. Thus, *knowledge* (e.g., *model*, *theory*) *validation* becomes "a process of building confidence in its usefulness with respect to a purpose" [PEBA00].

We recognize, in computer science, *theory validation* should:

1. Explain/interpret existing cases/phenomena (i.e., specific validity), just like *the periodic table of the chemical elements* could interpret recurring ("periodic") trends in the properties of known chemical elements;
2. Ideally, explain a new case (i.e., *generality*), just like validating *the periodic table* with a new chemical element.

Hence, we validate theories throughout this dissertation. In particular, we discuss validation in chapter summaries.

1.4 Research Questions

At the beginning of this project, the following exploratory questions helped us to get started on our research endeavour. Although we might not have perfect answers for all the questions in this dissertation, we made progress and contributed answers to all these questions.

- **What are suitable models, architectures, and workflows for RCAD?**
  They are the conceptual RCAD architecture that describes raw events, symptoms, syndromes and prescriptions; the event processing autonomic computing reference architecture (EPACRA) designed to achieve adaptive RCAD processes;

and net-casting information seeking model proposed by this dissertation that depicts the workflows of RCAD processes.

- **What automation mechanisms are employed in practice for monitoring, root cause analysis and diagnosis?**

  The RCAD processes are automated by event driven architecture (EDA). In the core of EDA are complex event processing (CEP) engines.

- **What are effective analysis techniques and tools with respect to performance, accuracy, ease of use and, portability for RCAD? What is the difference between non-adaptive and adaptive RCAD?**

  RCAD techniques and tools range from manual tools such as cause-and-effect diagram, interrelationship diagram and current reality tree, to computer-aided tools such as auto-prompted fishbone diagram. Computer-aided tools can be classified into rule-based system, codebook systems and artificial intelligent systems based on different event correlation techniques they employ [Tiff02]. Compared to rule-based system, artificial intelligent systems may have higher *recall* bout lower *precision.* The codebook approach always produces a diagnosis, as opposed to the rule-based systems, though the diagnoses may not always be accurate. The codebook technology needs the same expert knowledge as a rule-based system in order to accurately populate the codebook [Tiff02]. For designing a low-latency, high accuracy RCAD system, rule-based technique is our preferred approach.

  From the architecture level of viewpoint, a RCAD system built based upon autonomic computing paradigm is considered as an adaptive RCAD system (cf. 2.6). From the system component level of viewpoint, a RCAD system benefits from the flexibilities provided by CEP engines such as on-the-fly modifiable process rules and on-the-fly evolution is also considered as an adaptive RCAD system (cf. 10.5).

- **What are the most accessible technologies for developing a research platform for computer assisted RCAD?**
  They include the Eclipse development environment, the AspectJ technology and the open source CEP engine Esper [Espe10].

- **What research and industrial platforms exist for computer assisted RCAD?**
  Although many research tools (can also be leveraged as platforms) such as Pinpoint [CKFF02] and Magpie [BDIM04], and open source tools such as Glassbox [Glas10] exist, industrial platforms are just emerging such as CA's *system management* and *service availability management* (SAM). The OASIS white paper authored by CA Inc., IBM and Fujitsu researchers presents the vision for the *symptoms framework* (SF) [BBDL10], a specification that enables the automatic detection, optimization, and remediation of the operational aspects of complex systems. As an industry-wide new standard, symptoms framework (SF) should be regarded as a platform for computer assisted RCAD as well.

1.5 Dissertation Outline

To find the answer of the research questions, it is essential to survey related domains and accumulate background knowledge throughout the investigation. For this dissertation, we studied the literature of three different domains:

- Autonomic, self-adaptive and self-managing systems (cf. Chapter 2)
- Information seeking models (cf. Chapter 4)
- Complex event processing (CEP) systems (cf. Chapter 8)

All these domains contribute to the synthesized knowledge of this dissertation. Another large component of knowledge was acquired through experiential learning by participating in the CA Inc. NSERC CRD (Collaborating Research and Development) project.

Chapter 1 discusses the motivation for this research. Chapter 2 introduces autonomic computing concepts and autonomic computing reference architecture (ACRA). Chapter 3 explains and illustrates architecture patterns in the field of autonomic computing and application patterns identified by IBM researchers. Chapter 4 surveys various information seeking models. Chapter 5 proposes our new information-seeking model, called net-casting model. Chapter 6 introduces the conceptual RCAD architecture and compares three RCA tools. Chapter 7 describes how to automate the RCA process. Chapter 8 discusses the basic concepts and various aspects of complex event processing (CEP). A survey of popular CEP engines and event processing languages (EPL) is also presented. Chapter 9 describes the most important part of our research integrating results from previous chapters—the use-case-unit event processing network (EPN), based on CEP technology, real use-case workflows, and the net-casting information seeking model. Chapter 10 depicts the use-case-unit EPN as an essential part of our event processing autonomic computing reference architecture (EPACRA), which is an extension of IBM's ACRA model. Chapter 11 presents a case study—an intrusion detection system. Finally, related research methodologies and research validations are outlined in Figure 1-3 and Chapter 12 concludes the dissertation.

Figure 1-3 depicts the organizational flow of this dissertation. The arrows indicate prerequisite relationships among chapters.

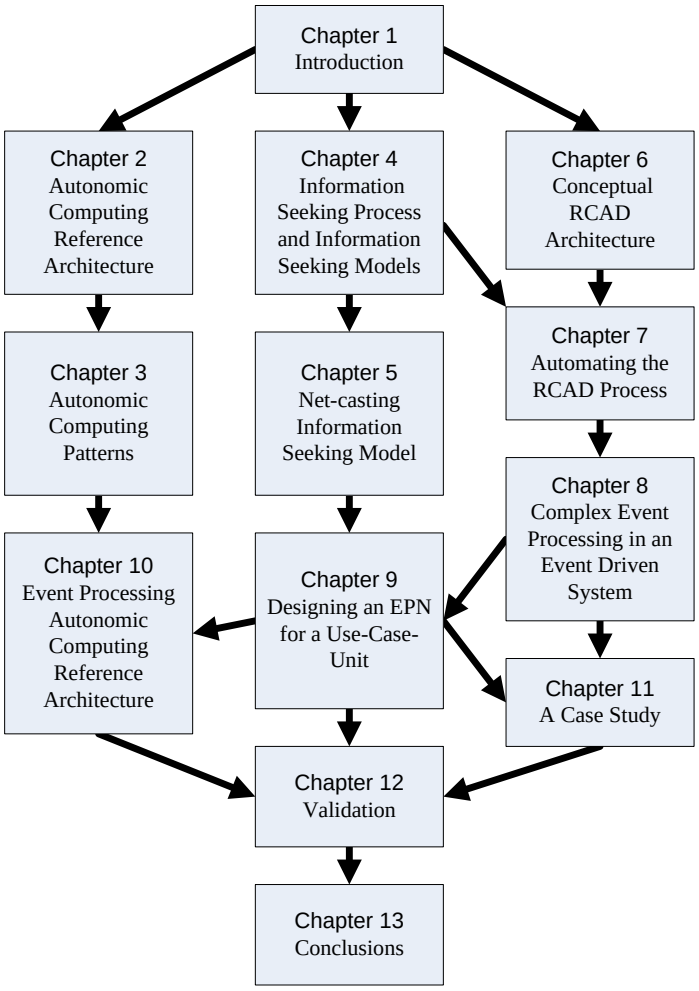


Figure 1-3: Organizational flow of the dissertation

Chapter 2 Autonomic Computing Reference Architecture

After IBM introduced autonomic computing technology with its autonomic computing initiative [KeCh03], researchers and practitioners made significant progress not only building autonomic capabilities into individual products, but also in creating open architectures for autonomic computing. IBM's architectural blueprint introduced the notion of an autonomic element, which is a fundamental building block for designing self-configuring, self-healing, self-protecting and self-optimizing systems, and autonomic computing reference architecture (ACRA), a common three layer architecture shared by many robotic systems, control systems and autonomic computing software systems [IBM06].

2.1 Autonomic Computing Concepts

Present-day IT environments are complex, heterogeneous tangles of hardware, middleware and software from multiple venders that are becoming increasingly difficult to integrate, install, configure, tune, and maintain. As software-based systems evolve, the overlapping connections, dependencies, and interacting applications call for administrative decision-making and responses faster than any human can deliver. The consequence is that pinpointing the root causes of failures becomes more difficult, while finding ways of increasing system efficiency generates problems with more variables than any human can hope to solve [Horn01].

To solve the enterprise scale management problem—simplify tasks for administrators and users of IT—we need to create a management system; in other words, we are "using technology to manage technology" [IBM06]. By embedding the complexity in the system infrastructure themselves, we are automating their management, so that computing systems are capable of self-configuration/reconfiguration, self-healing, self-optimization

and self-protection. A system that can achieve one of these self-* is considered as an autonomic computing system.

Salehie, in his dissertation [Sale09], selects 16 self-adaptive projects on the basis of their impact on the area of autonomic computing and the novelty/significance of their approach. He discusses the major self-* properties that are supported by each project, as shown in Table 2 ("✓" means supported). We shall notice that the majority of these projects focus on one or two of the known self-* properties and they are all considered as autonomic computing systems.

Table 2: Comparing projects in terms of self-* properties [Sale09]

Project	Self-Configuring	Self-Healing	Self-Optimizing	Self-Protecting
Quo [LBSZ98]	✓		✓	
IBM Oceano [AFFG01]	✓		✓	
Rainbow [GCHS04] [GaSc02]	✓	✓	✓	
Tivoli Risk Manager [TBHS03]				✓
KX [KPGV03] [VaKa03]	✓			
Accord [LPH04]	✓			
ROC [CCF04] [CKKF06]		✓		
TRAP [SMCS04] [SaMc04]	✓			
K-Component [DoCa04] [Dowl04]	✓			
Self-Adaptive [RoLa05]	✓	✓		
CASA [MuGl05]	✓		✓	
J3 [WSG05]			✓	
DEAS [LLMY05] [YLLM08]	✓	✓		
MADAM [FHSE06]	✓		✓	
M-Ware [KCCE07]	✓		✓	
ML-IDS [NKHL08]				✓

Originally, autonomic computing was introduced as a self-managing computing model and named after the human body's autonomic nervous system [Horn01]. An

autonomic computing system would control the functioning of computer applications and systems without input from the user, in the same way as the autonomic nervous system regulates body systems without conscious, intelligent control. Autonomic systems aim to manage the complexity of computing systems, to make decisions, and to respond quickly.

The goal of autonomic computing is to create systems that run themselves, capable of high-level functioning while keeping the system's complexity invisible to the user. Although technologies such as artificial intelligence may play important roles in the field of autonomic computing, autonomic computing is not focused on eliminating the human from the controlling loops. It only helps to eliminate mundane, repetitive IT tasks so that system administrators can apply technology to drive business objectives and set policies that guide decision-making. Autonomic computing keeps the complexity of computing systems to a minimum for administrators and users; however, it increases the complexity of computing systems themselves. Therefore, lightweight, fast-reacting, situation-detecting mechanisms designed to deal with a large amount of events are essential for the success of an autonomic system.

2.2 The Feedback Loop in Autonomic Managers

IBM’s architectural blueprint introduced the notion of an autonomic element (cf. Figure 2-1), which is a fundamental building block for designing self-configuring, self-healing, self-protecting and self-optimizing systems [IBM06].

At the core of an autonomic element is a closed-loop feedback control system. Its controller, also referred to as autonomic manager, manages the managed system, a set of resources, or other autonomic elements over a knowledge base. The autonomic manager operates in four phases [Gane07], as described in Table 3. It use policies (i.e., goals or objectives) to govern how each phase should be accomplished. The capabilities of an autonomic manager may be extended by reconfiguring these policies. For example, the monitor function can be extended by providing new symptom definitions, which help an

autonomic manager detect a condition in a resource that might require attention or reaction.

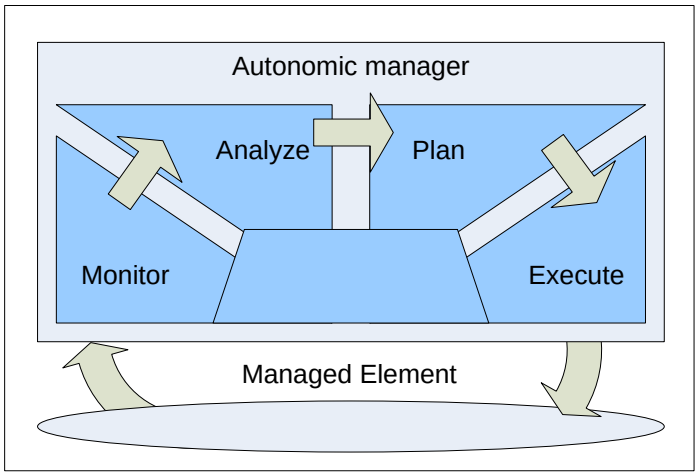


Figure 2-1: Autonomic element

Table 3: Four phases in autonomic manager [Gane07]

Phase	Tasks
Monitor	Collects, aggregates, correlates and filters events from managed resources through the *touchpoint* sensor interface, until it recognizes a symptom that needs to be analyzed. For example, a monitor might recognize an "increased transaction time" symptom based on response time metrics collected in real time from the system.
Analyze	Provides the mechanisms to observe and analyze symptoms to determine if some change needs to be made. For example, symptoms of "increased transaction time" might be analyzed to determine that more servers are needed to avoid violating a "response time" policy. In this case, a change request for "one more server to be assigned to the degraded application" might be generated to avoid a "response time" violation.
Plan	Generates an appropriate change plan, which represents a desired set of changes to be performed on the managed resource. The details of the change plan may be a simple command for a single managed resource, or it may be complex work flow that changes hundreds of managed resources.

Execute	Once an autonomic manager has generated a change plan that corresponds to a change request, some actions may need to be taken to modify the state of one or more managed resources. The actions are performed on the managed resource through the touchpoint-effector interface. In addition, part of the execution of the change plan could involve updating the knowledge that is used by the autonomic manager.

The aforementioned feedback loop is called *MAPE* or *MAPE-K loop* in the context of autonomic computing [KeCh03]. Dobson et al. refer to a similar loop as *autonomic control loop* in the context of autonomic communication, including *collect*, *analyze*, *decide* and *act* [DDFD06]. Oreizy et al. refer to this loop as *adaptation management*, which is composed of several processes for enacting changes and collecting observations, evaluating and monitoring observations, planning changes, and deploying change descriptions [OGTH99]. Salehie considers the feedback loop as an *adaptation loop* in self-adaptive systems [Sale09], which includes *monitoring* process, *detecting* process, *deciding* process and *acting* process, as described in Table 4.

Table 4: Four processes in an adaptation loop [Sale09]

Process	Description
Monitoring	Responsible for collecting and correlating data from sensors and converting them to behavioral patterns and symptoms. The process can be realized through event correlation, or simply threshold checking, as well as other methods.
Detecting	Responsible for analyzing the symptoms provided by the monitoring process and the history of the system, in order to detect when a change (response) is required. It also helps to identify where the source of a transition to a new state (deviation from desired states or goals) is.
Deciding	Determines what needs to be changed, and how to change it to achieve the best outcome. This relies on certain criteria to

	compare different ways of applying the change, for instance by different courses of action.
Acting	Responsible for applying the actions determined by the deciding process. This includes managing non-primitive actions through predefined workflows, or mapping actions to what is provided by effectors and their underlying dynamic adaptation techniques. This process relates to the questions of how, what, and when to change.

Although the nomenclature might be different in different contexts, the feedback loops are identical in nature. With four phases, a control loop in the autonomic computing context is formed to automate the tasks commonly performed by professionals in an IT organization.

2.3 Applying Autonomic Computing to IT Management

Many industry leaders, including IBM, HP, Oracle/Sun, and Microsoft are researching various components of autonomic computing. So far IBM's project is one of the most prominent and developed initiatives. IBM introduced their vision of self-managing systems in 2001 called it “the autonomic computing initiative” [Horn01] and distributed a series of documents called “an architectural blueprint for autonomic computing” [IBM06], such as the Autonomic Computing Tool Kit [ACTK05], to put their vision into practice. Meanwhile, many other companies pursued similar initiatives, such as Hewlett-Packard’s Adaptive Enterprise initiative [HP10] and Microsoft’s Dynamic Systems initiative [Micr10].

Considerable progress has been made not only in building autonomic capabilities into individual products, but also in creating open architectures for autonomic computing. Currently, industry standards that enable communication among heterogeneous

components are under development and many reference implementations for applying these standards are available to the public though the Internet. For instance, some of the protocols, standards, and formats that have been utilized includes: CBE (Common Base Events) [IBM05][IBM10A], WBEM (Web-Based Enterprise Management) [DMTF10A], which includes CIM (Common Information Model) [DMTF10B]. Other event notification services are designed and implemented by researchers in universities such as SIENA (Scalable Internet Event Notification Architectures) [CRW01]. Tewari and Milenkovic made a comprehensive survey of the standards for autonomic computing [TeMi06]. They described a standards stack (cf. Figure 2-2) for enabling autonomic computing spanning hardware management, OS/Application management, services, and business process management. The abbreviations used in Figure 2-2 are listed in Appendix A.

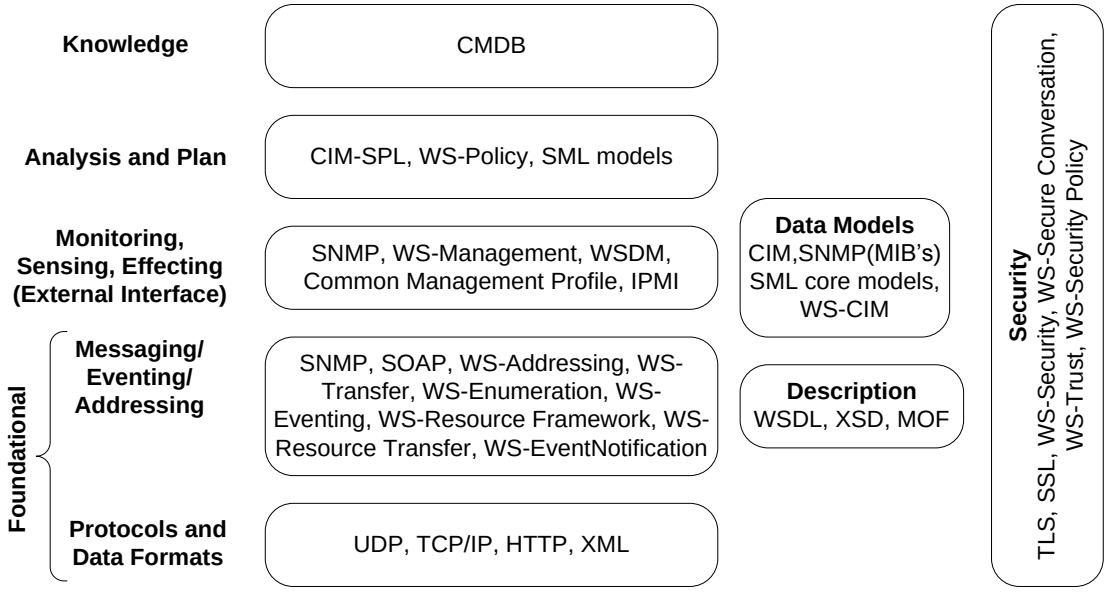


Figure 2-2: standards for autonomic computing [TeMi06]

One particular useful open standard for sensing is ARM (Application Response Measurement) [Open10], which enables developers to monitor and diagnose performance bottlenecks within complex enterprise applications that use loosely-coupled designs or

service-oriented architectures. SNMP (Simple Network Management Protocol) [IETF10] is used mostly in network management systems to monitor network-attached devices for conditions that warrant administrative attention. It is also applicable to autonomic computing systems.

Profiling tools and techniques can also be useful in defining desirable sensors such as JVMTI (Java Virtual Machine Tool Interface) [Sun10]. Software management frameworks, such as JMX (Java Management eXtensions) [Java10] provide powerful facilities for both sensing and effecting. The colleagues of my research group used *Java reflection* for monitoring [DDKM08].

Besides aforementioned protocols, standards, and formats, there are also open source projects for developers to leverage, such as: CASCADAS (Component-ware for Autonomic, Situation-aware Communications And Dynamically Adaptable), which provides component-ware for autonomic situation-aware communications, and dynamically adaptable services [CASC10]. ACE (Autonomic Communication Elements) Autonomic Toolkit, as a platform for setting up autonomic services in a distributed environment—it provides service discovery, service provisioning/usage, autonomic adaptation to the context/mobility, support for supervision and service aggregation [Sour10]. ANA (Autonomic Network Architecture), allows dynamic adaptation and re-organisation of the network according to the working, economical and social needs of the users [ANA10]. JADE, a framework for construction of autonomic systems, targets to autonomic management of complex systems including legacy software [Jade10]. SOCRATES (Self-Optimisation and self-ConfiguRATion in wirelEss networkS), aims at the development of self-organisation methods to enhance the operations of *wireless access networks*, by integrating network planning, configuration and optimisation into a single, mostly automated process requiring minimal manual intervention [Seve10].

2.4 Autonomic Computing Reference Architecture

Traditionally, IT operations have been organized based on individual silos—separated by component types and platform types. For example, a particular administrator might be only concerned with managing databases or managing application servers. The autonomic computing architecture formalizes a reference framework that identifies common functions across these silos; it also consists of different types of building blocks in the architecture. To achieve autonomic computing, these building blocks listed in Table 5 are essential [Gane07].

Table 5: The building blocks in autonomic computing systems [Gane07]

Component	Functionalities
Task manager	Enables IT personnel to perform management functions through a consistent user interface.
Autonomic manager	Automates common functions and management activities using an autonomic control loop. This control loop, including monitor, analyze, plan, and execute, is governed by humans (administrators) as well as rules and policies (defined by humans) and learned by the system.
Knowledge source	Provides information about the managed resources and data required to manage them, such as business and IT policies.
Enterprise service bus	Leverages Web standards to drive communications among components throughout the environment.
Touchpoint	Provides a standardized interface for managed resources such as servers, databases, and storage devices, etc. Autonomic managers sense and affect the behaviour of these resources only though the touchpoints.

To build an autonomic system, designers need an arrangement of collaborating autonomic elements working towards a common goal. The *autonomic computing reference architecture* (ACRA) presented in IBM's architectural blueprint defines a common approach for self-managing autonomic computing systems [IBM06]. This

blueprint organizes an autonomic computing system into layers and parts as depicted in Figure 2-3.

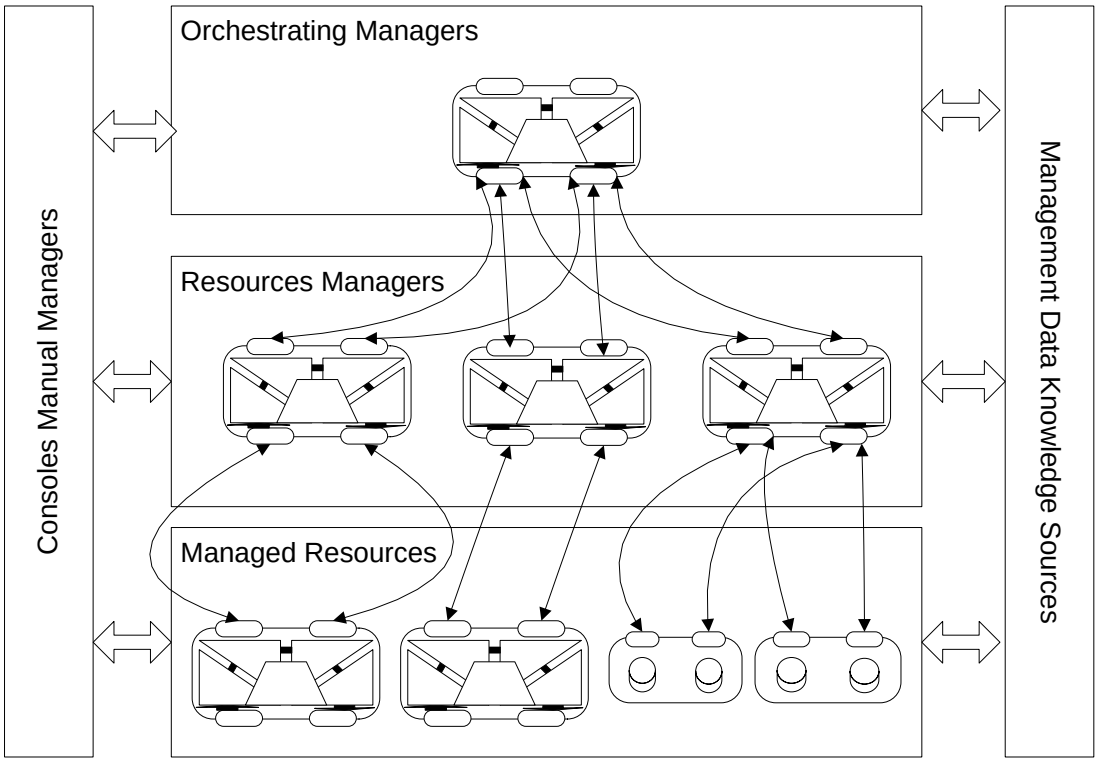


Figure 2-3: Autonomic Computing Reference Architecture (ACRA) Model

The lowest/bottom layer contains the system components or managed resources. These managed resources can be any type of resource, either hardware or software. These resources may have some embedded, self-managing attributes (i.e., two self-managed resources are depicted in the left side of the bottom layer in Figure 2-3). Each managed resource may incorporate standard manageability endpoints (sometimes called *touchpoints*) for accessing and controlling the managed resources.

The middle layer contains resource managers which are often classified into four categories: self-configuring, self-healing, self-optimizing and self-protecting. A particular resource may have one or more resource managers, each implementing a relevant control loop. The top layer contains autonomic managers that orchestrate resource managers.

These orchestrating autonomic managers deliver a system-wide autonomic capability by incorporating control loops that realize broad goals of the overall IT infrastructure. The left side in Figure 2-3 illustrates a manual manager that provides a common system management interface for the IT professional using an integrated solutions console. The various manual and autonomic manager layers can obtain and share knowledge via knowledge sources which are depicted on the right side in Figure 2-3.

All building blocks in the ACRA model, such as endpoints for managed resources, knowledge sources, resource managers and manual managers, are connected using Enterprise Service Bus Pattern (c.f 3.1.1 Application Pattern 1a) that allow the components to collaborate using standard mechanisms such as Web services.

2.5 A Three Level Hierarchical View

In fact, the three layer architecture described by ACRA model represents a common three layer architecture shared by many robotic systems, control systems and autonomic computing software systems. To some extent (particularly, in my research), all three layers reflect the different abstraction levels of self-* properties. In Salehie's dissertation [Sale09], he illustrates a three-level hierarchy of self-* properties, as depicted in Figure 2-4.

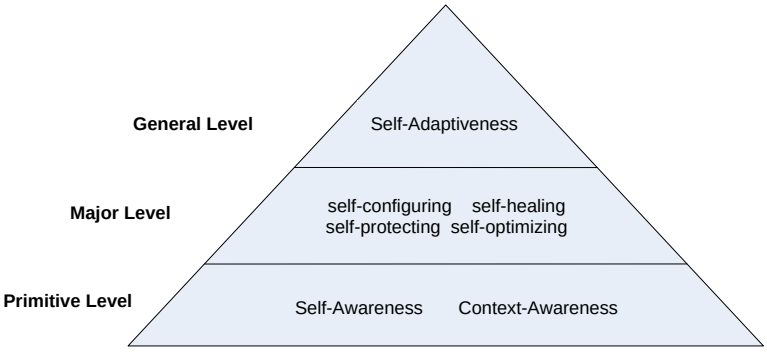


Figure 2-4: A three-level hierarchy of self-* properties [Sale09]

Among them, self-adaptiveness is a *general* property, which is at the top level of the hierarchy. It can be decomposed into *major* properties at the middle level and further into *primitive* properties at the bottom level of the hierarchy. The major properties include: self-configuration, self-healing, self-optimization and self-protection. The primitive properties include: self-awareness and context-awareness. All the properties described by Salehie [Sale09] are listed in Table 6.

Table 6: The list of all self-* properties described by Salehie [Sale09]

Level	Description	
General level	Contains global properties of self-adaptive software. A subset of these properties, which falls into the category of self-adaptiveness [OGTH99], consists of self-managing, self-governing, self-maintenance [KeCh03], self-control [KBE99], and self-evaluating [Ladd06]. Another subset at this level is self-organizing [JBLN06] [SFHK03], which emphasizes decentralization and emergent functionalities.	
Major level	The IBM autonomic computing initiative defines a set of four properties at this level [Horn01]. These properties have been defined in accordance to biological self-adaptation mechanisms [KeCh03]. For instance, the human body has similar properties in order to adapt itself to changes in its context (e.g., changing temperature in the environment) or self (e.g., an injury or failure in one of the internal organs). The following list further elaborates on the details.	
	Self-configuring	Refer to the capability of reconfiguring automatically and dynamically in response to changes by installing, updating, integrating, and composing or decomposing software entities.
	Self-healing	Related to self-diagnosing [RoLa05] or self-repairing [LeFi02], is the capability of discovering, diagnosing, and reacting to disruptions. It can also anticipate potential problems, and take proper actions accordingly to prevent a failure. Self-diagnosing refers to diagnosing errors, faults and failures, while self-repairing focuses on recovering from them.
	Self-optimizing	Also called self-tuning or self-adjusting [SPTU05], is the capability of managing performance and resource allocation in order to satisfy the requirements of different users. End-to-end

		response time, throughput, utilization, and workload are examples of important concerns related to this property.
	Self-protecting	Refer to the capability of detecting security breaches and recovering from their effects. It has two aspects, namely defending the system against malicious attacks, and anticipating problems and taking actions to avoid them or to mitigate their effects.
Primitive level	Self-awareness, self-monitoring, self-situated, and context-awareness are the underlying primitive properties [Horn01][SaTa05]. The following list further elaborates on the details.	
	Self-awareness [HiSt06]	The system is aware of its self states and behaviours. This property is based on self-monitoring which reflects what is monitored.
	Context-awareness [PaHa05]	The system is aware of its context, which is its operational environment.

As mentioned above, the three layer architecture is prevalent in service-oriented software systems, automation systems, decision-support systems, and many other types of adaptive and self-managing systems. The AI and robotics communities and in particular Gat (1991) [Gat91] [Gat97], Shibata & Fukuda (1994) [ShFu94], Bonasso et. al. (1998) [BKJJ98], and Kramer & Magee (2007) [KrMa07] generated several closely related three-layer reference control architectures that are applicable to self-managing systems. Table 7 shows that many three-layer reference architectures naturally correspond to the three-level hierarchy of self-* properties.

The rationale for the three tiers is usually not explicitly stated, but it is frequently a natural fit. It reflects the complexity and flexibility of the self-* properties within systems. Generally speaking, self-adaptive and self-organizing are designated to satisfy more general and abstract “high-level” requirements, whereas self-awareness and context-awareness are more incline to primitive and detailed “low-level” requirements.

The complexity of reasoning (i.e., intelligence) increases from the bottom to the top level. Conversely, the flexibility of policies (i.e., precision) decreases from top to bottom level [GDZK10].

Table 7: Three-layer reference control architectures matches to hierarchy of self-* properties

Hierarchy of self-* properties [Sale09]	**ATLANTIS** [Gat91]	**HICS** [ShFu94]	**3T** [BKJJ98]	**ACRA** [IBM06]	**Kramer & Magee** [KrMa07]	**Adaptive SOA** [GrRo08]
General	Deliberator	Organization	Planning	Orchestrating managers	Goal management	User management
Major	Sequencer	Coordination	Sequencing	Resource managers	Change management	Workflow management
Primitive	Controller	Execution	Skill	Managed Resources	Component control	Service management

2.6 Focus of our Research

Our RCAD research falls under the umbrella of self-healing, which is at the *major level* of the hierarchy of self-* properties, and closely related to design and implementation of *resource managers* in the middle layer of the ACRA model. So the main effort of this dissertation is on how to leverage existing technology to construct resource managers—essentially, they are autonomic elements in the middle level of the ACRA model. The result of this research is event processing networks (EPN), an innovative solution to automate the processes of RCAD tasks. Driven by events, these EPNs are characterized with feedback loops of autonomic elements. Since the EPN solution is a new complex event processing (CEP) based approach to implement resource managers, we call our ACRA model as event processing autonomic computing reference architecture (EPACRA). It is a variant of ACRA model presented by IBM.

In the long run, more effort is needed in the *general level* of the hierarchy of self-* properties, particularly the global properties of self-adaptiveness, which includes self-managing, self-governing, self-maintenance, self-control and self-evaluating. According to Salehie's [Sale09], *Self* means the whole body of software, mostly implemented in several layers, while the context encompasses everything in the operating environment that affects the system properties and its behaviour. Autonomic systems aim at adjusting various components and attributes in response to changes in the self and in the context of a software system. It is the right technology to achieve self-adaptation in software systems. And the RCAD system built based upon autonomic computing paradigm is considered as an *adaptive* RCAD system.

Summary

This chapter introduced the concepts of autonomic computing, autonomic element, as well as IBM's ACRA model. We also surveyed industry standards related to autonomic computing technologies and reference implementations. We briefly introduced popular open source projects that are widely available on the Web. One of the main findings of this chapter is that our research focus is in the middle layer of ACRA.

Chapter 3 Autonomic Computing Patterns

Although our research has identified six patterns of autonomic elements [ZLKM08], it is unclear how autonomic elements interact with each other and how individual autonomic elements form feedback loops in these systems. Based on the Application Pattern 2 *Manager-of-Manager Interactions* and Pattern 3a *Composed Autonomic Managers* identified by Sweitzer and Draper [SwDr07], we are able to evolve the six patterns of autonomic elements into a set of *autonomic computing architecture patterns.* After we unambiguously illustrate these architecture patterns with structure diagrams for the first time, we can clearly observe missed patterns related to the *monitor sector*. This problem is solved by adding a new architecture pattern, called chain-of-monitors, into the lattice of autonomic computing architecture patterns.

3.1 Application Patterns for Autonomic Computing

Sweitzer and Draper [SwDr07] present eight patterns and demonstrate how application patterns should be applied in the autonomic systems to solve specific problems. In their article, each of the application patterns is described in two parts: problem and solution.

- **Problem:** States the problem to be solved; summarizes the problem itself (highlighted in bold in Table 8 to Table 13 below) and describes the underlying issues.
- **Solution**: Consists of a summary of the solution (highlighted in bold in Table 8 to Table 13 below); a solution description, and the results or benefits that can be realized.

Except Pattern 5 *Delegating Tasks in a Process Context* and Pattern 6 *Process Integration through Shared Data*, which are specifically applied in IT management processes, the other six patterns (Pattern 1a, 1b, 2, 3a, 3b and 4) are more general and can be applied in many autonomic systems. These six patterns are slightly restated as follows:

3.1.1 Pattern 1a: Use of Enterprise Service Bus for Manager-to-Resource Interactions

The simplest autonomic manager pattern is a single manager managing a single resource. In autonomic systems, there is a situation where multiple managers, each designed to meet a specific management goal, interact with the same resource.

Table 8: Pattern 1a: Use of Enterprise Service Bus for Manager-to-Resource Interactions [SwDr07]

Problem	**How can multiple autonomic managers manage the same resource, without driving an N×M increase in configuration complexity?** Whenever a new resource is added, not only does the resource need to be configured, but all managers that are designed to manage the new resource need to be informed. Likewise, whenever a new manager is added, not only does the manager need to be configured, but all of the resources that were designed to be managed by this manager need to be informed as well.
Solution	**Configuration of manager-to-resource interactions should be externalized into functions within an Enterprise Service Bus (ESB)** The details of the interactions among autonomic managers and resources should be handled by the mechanism provided by the ESB, such as the publish-subscribe mechanism. Externalizing the configuration of manager-to-resource interactions makes the configuration of the management system more visible and manageable. However, this pattern does not resolve conflicts between managers.

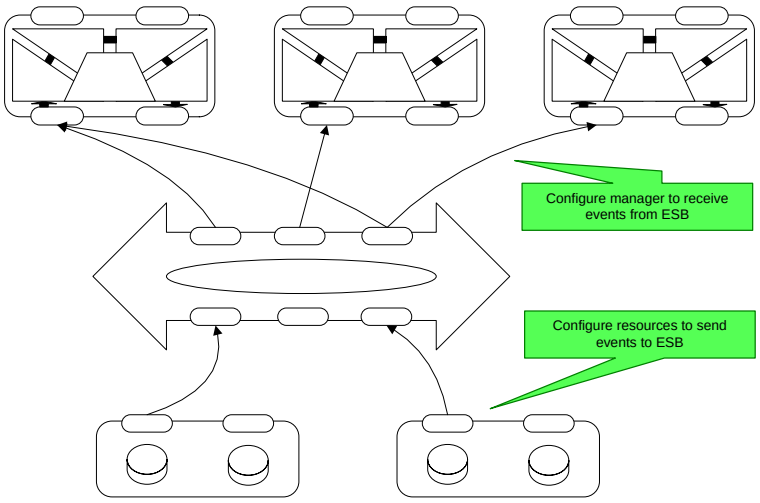


Figure 3-1: Use the ESB to manage notification from resources to autonomic managers

3.1.2 Pattern 1b: Shared Resource Data among Managers

By using the Enterprise Service Bus (ESB) for manager-to-resource interactions, multiple autonomic managers may manage a single resource and share the data of that resource. ITIL describes the idea of a *configuration management database* (CMDB) as a resource that tracks the configurable elements of a system as well as the relationships among them [Bon07].

Table 9: Pattern 1b: Shared Resource Data among Managers [SwDr07]

Problem	**How can data about resources be shared efficiently among multiple autonomic managers?** Information about manageable resources is held in multiple locations, including knowledge base, autonomic managers and the managed resource's touchpoint. Potentially, autonomic managers may have out-of-date knowledge about the resource.
Solution	**A federated Configuration Management Database (CMDB) provides efficient access to information about resources that may be held in multiple sources.**

	Important information about resource and their relationships is stored in a knowledge base—CMDB. This allows resource data to be accessed by a requester, who does not have to know how that data was acquired. One example is CMDB populated with information about a solution it has deployed (including component resources and dependencies). This relationship information might then be used by problem determination for problem diagnosis.

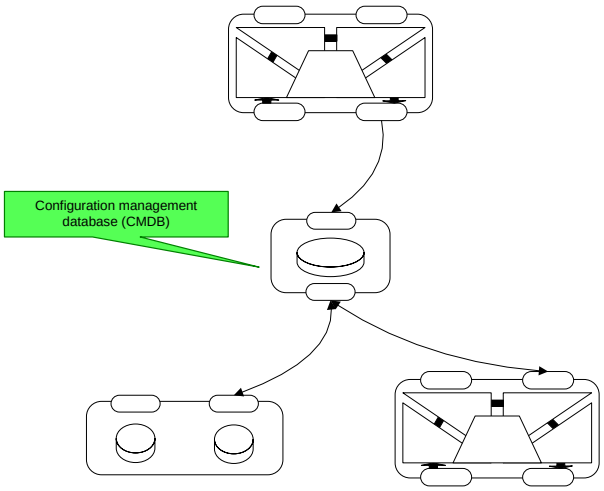


Figure 3-2: Federate accesses to resource information through CMDB

3.1.3 Pattern 2: Manager-of-Manager Interactions

Autonomic managers within an autonomic computing system need to be managed in a coordinated way.

Table 10: Pattern 2: Manager-of-Manager Interactions [SwDr07]

Problem	**How are the elements of an autonomic management system managed?** One of the potential pitfalls of any management system is that the management technology itself introduces even more complexity into the environment. To avoid the paradoxical situation where the autonomic

	elements themselves need extensive maintenance work, they need to be managed in a straightforward and consistent way.
Solution	**Autonomic managers can manage other autonomic managers using the same sensor and effector interfaces that are used to manage resources.** Autonomic managers have unique manageability capabilities that allow the manager of an autonomic manager to assign manageable resources to the managed autonomic manager, and set the policies that govern how the managed autonomic manager accomplishes its mission. An example of this pattern is a transaction workload manager that delegates management of a tier of resources to a local manager, which optimizes performance within that tier.

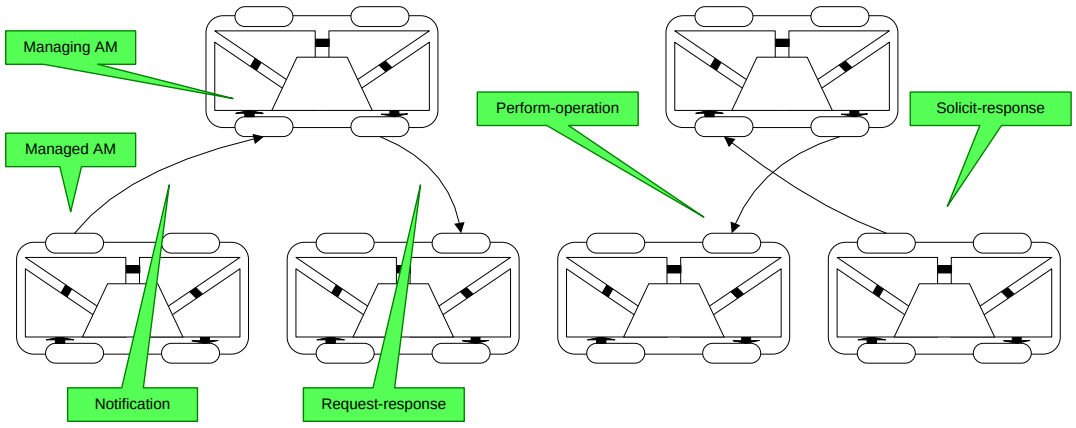


Figure 3-3: Manager-of-manager interactions use same interface as resources

3.1.4 Pattern 3a: Composed Autonomic Managers

Partial autonomic managers are a useful building block because they allow customers to acquire and implement autonomic technology in an incremental manner. As an organization seeks to increase the autonomic maturity of its management systems, these incremental blocks of automation (potentially from different vendors) need to be connected together.

Table 11: Pattern 3a: Composed Autonomic Managers [SwDr07]

Problem	**How can multiple partial autonomic managers be integrated together to provide a complete control loop?** An architected way is needed to compose the autonomic managers (this pattern) and to manage the federation of the data shared among the managers.
Solution	**Partial autonomic managers should be composed to form more complete control loops, using canonical interfaces and data types.** Use of this composition pattern provides a structured way to compose partial autonomic managers into complete control loops. For example, a partial autonomic manager performing an analyze function might provide a "diagnose problem" interface that consumes a canonical symptom. In cases in which these interfaces are not provided natively by the implementation, they could be implemented using standard application integration patterns, such as adapters. These canonical interfaces can then be used to compose multiple partial autonomic managers into more complete control loops.

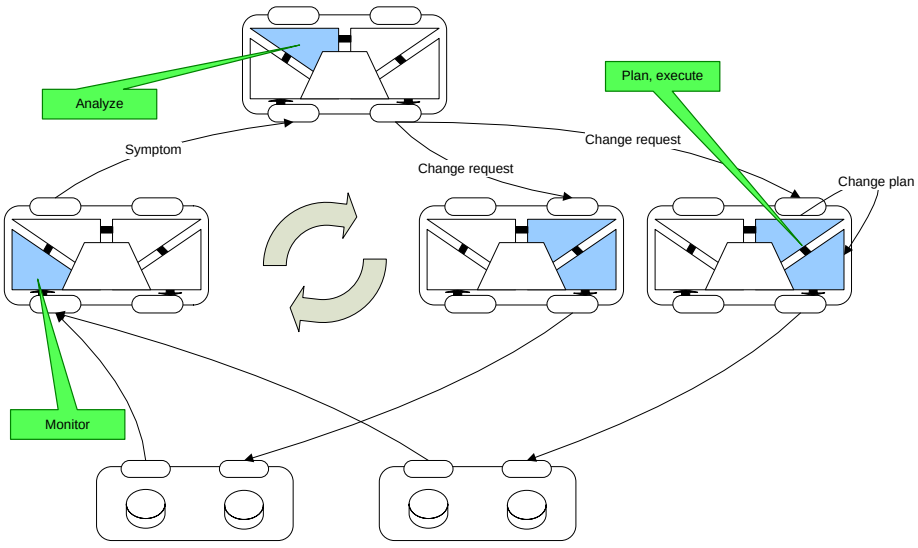


Figure 3-4: Composing partial autonomic managers

3.1.5 Pattern 3b: Use of ESB for Composing Autonomic Managers

The "Composing autonomic managers" pattern (Pattern 3a) describes how partial autonomic managers can be integrated together to form a more complete control loop. However, the composition of managers is static (e.g., built into the manager or the adapter for the manager), which makes it more difficult for the IT organization to respond to changes and problems.

Table 12: Pattern 3b: Use of ESB for Composing Autonomic Managers [SwDr07]

Problem	**How can the composition of autonomic managers be made readily reconfigurable?** A way to separate the routing decisions from the autonomic manager implementations is needed, allowing a single consistent configuration mechanism.
Solution	**Configuration of manager-to-manager communications should be externalized into a function within the ESB.** The communication between two autonomic managers should be constructed so that the invoking manager is unaware of the details of the invoked manager. These details should be handled by patterns within the ESB, such as mediation and content-based routing. Use of the ESB to externalize the configuration of manager-to-manager interactions facilitates the dynamic reconfiguration of the management system in response to change. It also facilitates non-intrusive monitoring of the management system. Manager-to-manager events can be monitored as they flow across the ESB.

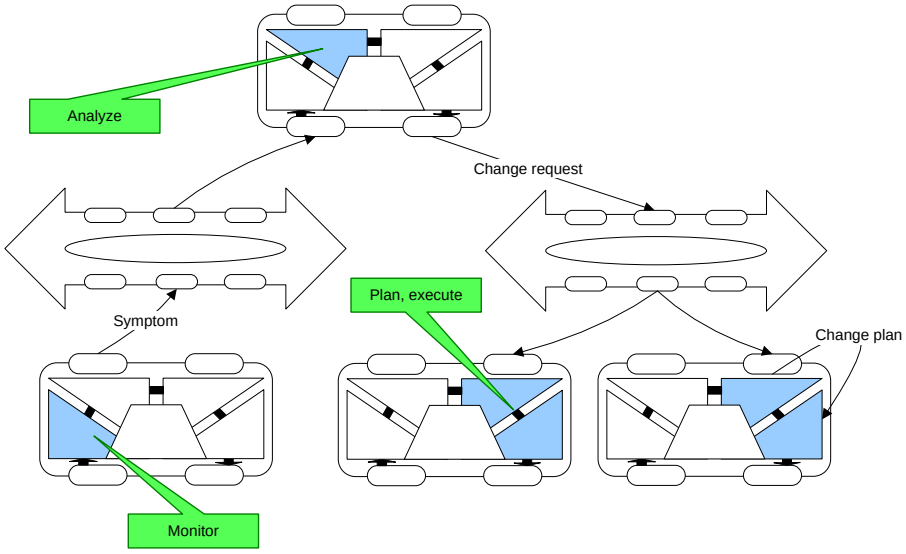


Figure 3-5: Using the ESB to configure interactions between autonomic managers

3.1.6 Pattern 4: Embedded Autonomic Manager

Resource providers build logic into the runtime of their managed resources to perform management tasks. When any of the details for the control loop are visible, this pattern is used so that the control loop can be configured through the manageability interface of the managed resource (e.g., a disk drive).

Table 13: Pattern 4: Embedded Autonomic Manager [SwDr07]

Problem	**How can the management function delivered in the runtime of a resource participate in the autonomic computing system?** The engineers of these embedded management capabilities are challenged to incorporate sophisticated management functions without impacting the performance of the resource.
Solution	**The embedded management function is exposed as an autonomic manager manageable resource through the touchpoint for the managed resource.** The logic embedded in the resource exposes its functionality as an "autonomic manager" manageable resource through the touchpoint. A

	critical characteristic of this pattern is that the "embedded management function" is exposed to the rest of the system as an autonomic manager, just like any other management function in the system. An example of using this pattern is a database optimizer that monitors typical database transactions and creates or recommends appropriate search indexes to optimize overall database performance.

These application patterns are identified and distilled by Sweitzer and Draper from practical cases [SwDr07]. We recognized that the foundations of these application patterns are some general software design principles such as *levels of indirection*, and *separation of concern*. Among them, Pattern 2: *Manager-of-Manager Interactions* and Pattern 3a: *Composed Autonomic Managers* are the most essential to our work. Based on these two application patterns, we are able to describe and illustrate the architecture patterns for autonomic systems without ambiguity.

3.2 Autonomic Element Patterns

The engineering of self-adaptive software systems is a comparably new discipline although system architectures based on feedback loops have a rich history in control theory and in many engineering disciplines. We have attempted to collect and identify selected common forms of autonomic element designs and distilled them into autonomic element patterns [ZLKM08]. Among these six patterns, three are from the architectural styles of Hawthorne & Perry's self-adaptive system [HaPe10], one from the externalizing application logic of Chan and Chieu [ChCh03], and two from the goal model solutions of Lapouchnian et al. [LYLM06]. The inherent relationship of these patterns is laid out in Figure 3-6.

Hawthorne and Perry presented and discussed several reference architectural styles for incorporating run-time reflection and adaptation into software systems [HaPe10]. They are called aspect-peer-to-peer, aggregator-escalator-peer, and chain-of-configurator.

These styles are specifically for self-healing systems, which have been extended to ABASs (Attribute-Based Architectural Styles) by Neti and Müller [NeMü07].

Table 14: Hawthorne & Perry's architectural styles

Aspect-peer-to-peer	This is a simple architectural style consisting of a monitor component for each aspect of the system and its environment in need of monitoring, with a peer configurator component to reconfigure the system.
Aggregator-escalator-peer	This style overcomes the limitations of the strict aspect-peer-to-peer by allowing monitors to pass their outputs to higher-level aggregator monitors. A higher-level configurator can then make more informed configuration decisions.
Chain-of-configurators	There are two variants to this style. The first variant is similar to the chain-of-responsibility design pattern where multiple configurators are chained together. The second variant uses a visitor pattern in which configurator chaining is used to compose higher order configuration functionality.

We noticed that among the architectural styles of Hawthorne & Perry's, the aspect peer-to-peer pattern is the simplest form—often linked to ad hoc solutions consisting of hard-coded inline program statements that check for and respond to specific run-time conditions. Aggregator-escalator-peer improves management capability and maintainability by adding in an extra tier (level of indirection) from the monitor sector and chain-of-configurators from the configurator sector. Both styles enhance loose coupling and promote maintainability.

Another popular architectural design used by Chan and Chieu is externalizing application logic [ChCh03]. By representing the reasoning logic as rules, this approach improves maintainability from the reasoning sectors (including the analyzing and planning parts) of autonomic elements.

Lapouchnian et al. proposed a design technique for autonomic systems based on goal-driven requirements engineering [LYLM06]. This method produces an arrangement of autonomic elements that is structurally similar to the goal hierarchy of the corresponding goal model. One extreme solution is to realize the entire goal model using a single autonomic element. On the other side of the spectrum is a solution that allocates an autonomic element for each node in the goal model. However, a combination of these two extreme mappings appears to be most practical.

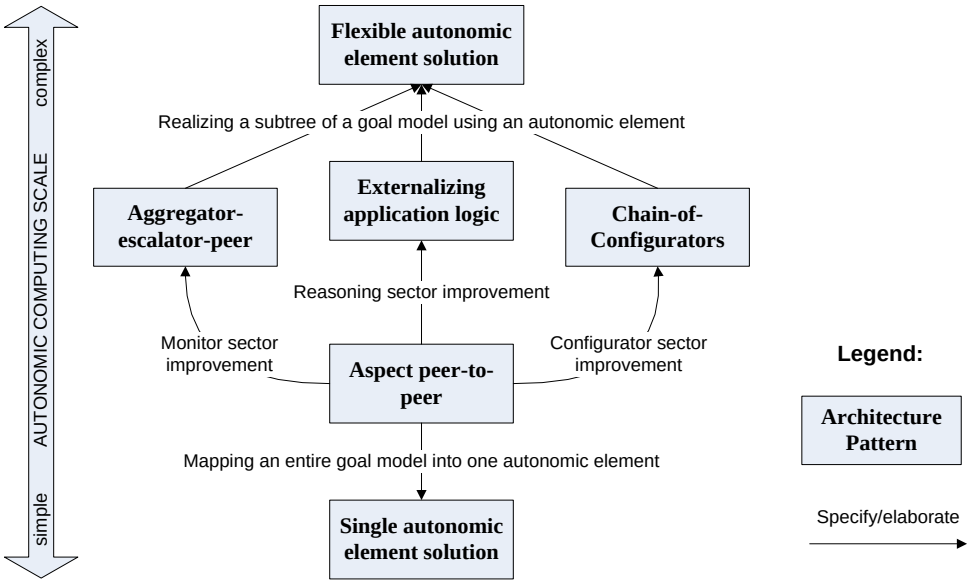


Figure 3-6: Lattice of autonomic element patterns

Figure 3-6 organizes these autonomic architecture patterns into a lattice from the single autonomic element solution (e.g., mapping an entire goal model into one autonomic element) at the bottom of the diagram to the most flexible (e.g., loose coupling) solution at the top (e.g., different sub-trees of a goal model are optimally mapped to a collaborative arrangement of autonomic elements). The middle levels comprise selected solutions which optimize specific quality criteria such as maintainability and extensibility (e.g., the patterns by Hawthorne and Perry patterns and by Chan and Chieu). The arrows in the diagram indicate the inherent relationships among the different architectures—the abstraction part of one pattern can be specified / elaborated by another pattern.

3.2.1 The Sensors and Effectors of an Autonomic Element

We noticed that there is some confusion between the terms *configurator* and *executor* in autonomic element patterns. For example, instead of peering with executor, monitor sector peers with the configurator sector in the aspect-peer-to-peer autonomic element pattern [ZLKM08]. To clarify this matter, we elaborate the structure of an autonomic element as follows:

- The controller of the closed-loop within an autonomic element, also referred to as the autonomic manager, operates in four phases: monitor, analyze, plan and execute.
- The realizations of these four phases are four sectors: monitor, analyzer, planner, and executor.
- An autonomic manager manages the managed system, a set of resources, or other autonomic elements through a set of sensors and effectors. Figure 3-7 depicts the sensors and effectors of an autonomic element.

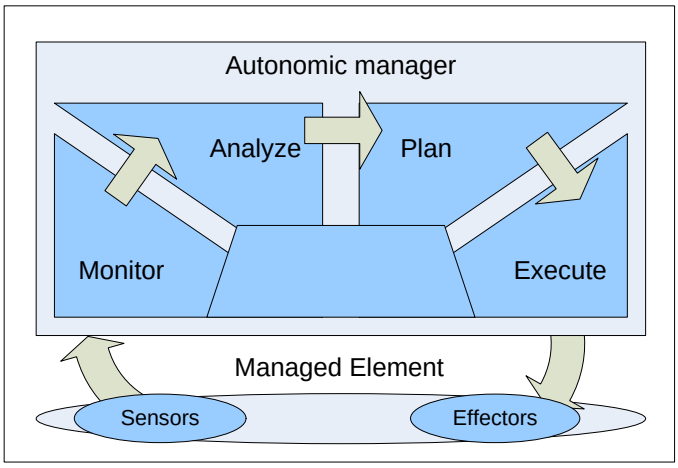


Figure 3-7: The sensors and effectors of an autonomic element

We identified a group of six autonomic element patterns [ZLKM08]. However, how feedback loops within an autonomic computing system related to one another is still unclear. Based on the structure of an autonomic element, and the application patterns Pattern 2 and Pattern 3a introduced above, we evolve six autonomic element patterns into

a set of autonomic computing architecture patterns from the feedback loop perspective. In particular, we rename the chain-of-configurators pattern as chain-of-executors architecture pattern (The term chain-of-configurators origins from Hawthorne and Perry's adaptable self-healing systems [HaPe10] and suitable for self-configurating and self-healing systems only. The term chain-of-executors is more appropriate in a general autonomic computing context); we also revise and rename the lattice of autonomic element patterns as the lattice of autonomic computing architecture patterns. To qualify the representation of the lattice, we define:

Definition 1:

A monitor is shown by M, and a set of monitors is shown by M. An executor is represents by E, and a set of executors are E*. Also, a single application logic (includes an analyzer and a planner) is referred as A, and a set of application logics is referred as A*. For simplicity, we show any possible multiplicity by *. It means that there are multiple same sectors/phases of autonomic elements in the system.*

Definition 2:

Let L be the existing hierarchy relationship among same sectors/phases of autonomic elements.

3.2.2 Aspect-peer-to-peer Architecture Pattern

Neti and Müller [NeMü07] suggested the aspect-peer-to-peer style of ABAS can be viewed as a set of control loops (i.e., one control loop exists for each component of the system). Further, each control loop in the set is independent and complete on its own. We submit that these characters are readily applicable to the aspect-peer-to-peer architecture pattern. From a maintainability point of view, the aspect-peer-to-peer architecture pattern is suitable for adaptive systems with a small number of sensors and effectors since extra resources are needed to respond to each low-level sensor alert separately. In this pattern, an executor does not require output from more than one monitor to make a decision. Usually one control loop has only one monitor and one executor. Different control loops

do not interact with each other. By leveraging application Pattern 3a Composed Autonomic Managers, we are able to illustrate aspect-peer-to-peer architecture pattern in Figure 3-8.

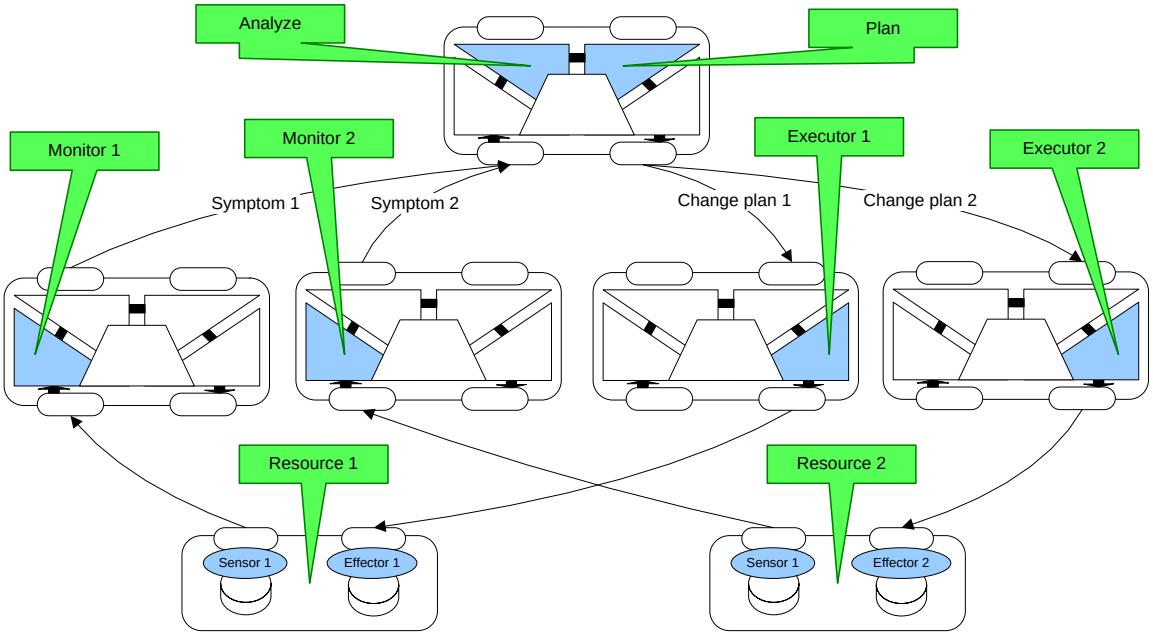


Figure 3-8: Aspect-peer-to-peer architecture pattern

The aspect-peer-to-peer architecture pattern is,

$$\mathrm{AP2P} = \{\mathrm{M}^*, \mathrm{E}^*, \mathrm{A}\} \quad (4)$$

Where, M* and E* denote multiple monitors and executors, respectively. The A presents single application logic.

Logically, a monitor and its peer executor belong to the same autonomic element. Physically, they might be realized in the same or different implementation entities (i.e., components, objects, or methods). A set of autonomic elements may monitor and execute upon different resources of a system, or on different sections (e.g., aspects) of one resource. Between multiple autonomic elements and multiple managed resources, application pattern 1a *Use of Enterprise Service Bus for Manager-to-Resource Interactions* could be applied to alleviate the complexity problem.

A strict peer-to-peer approach decreases coupling, which in turn increases system modifiability. But increased modifiability comes at the cost of limited versatility and operational efficiency since a better configuration decision cannot be made, although their reasoning sectors are realized in the same implementation entity.

3.2.3 Single Autonomic Element Architecture Pattern

The most straightforward approach to overcome the limitations of the aspect-peer-to-peer architecture pattern is to define a single autonomic manager that is responsible for the whole system. The autonomic manager therefore takes the majority of autonomic management tasks. With all of the analysis and monitoring in one place, this approach can be helpful in achieving optimal performance by minimizing overhead and omitting levels of indirection. However, the single autonomic element architecture pattern can make the system less flexible and harder to maintain. Figure 3-9 illustrates the single autonomic element architecture pattern.

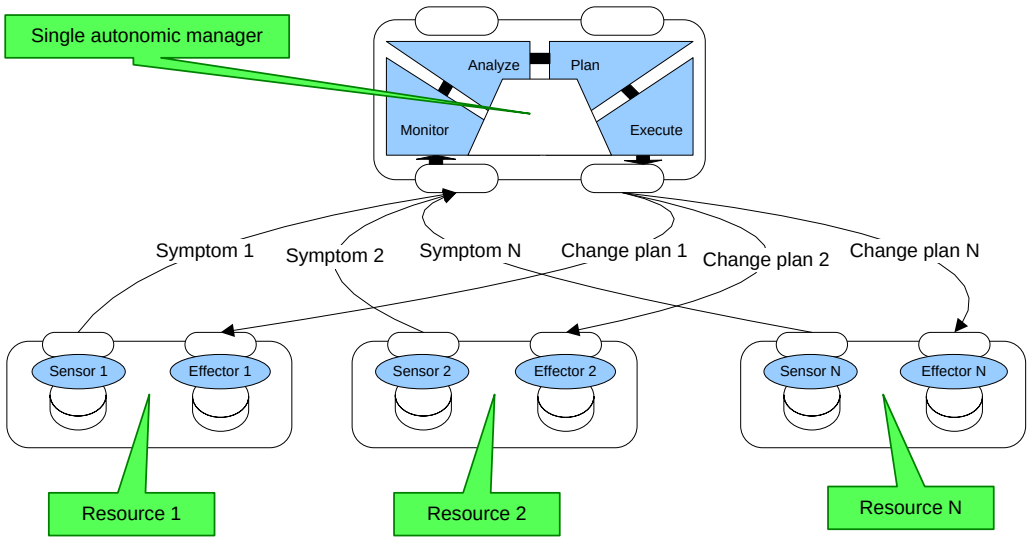


Figure 3-9: Single autonomic element architecture pattern

The single autonomic element architecture solution is,

$$\mathrm{SAE} = \{\mathrm{M}, \mathrm{E}, \mathrm{A}\} \quad (1)$$

Where, M, E, and A denote a single monitor, a single executor, and a single application logic, respectively.

This pattern is suitable for self-adaptive systems with a modest number of resources since the reasoning components need to respond to each low-level sensor alert simultaneously. One effector may use outputs from more than one sensor to make a decision. That means multiple internal control loops might entangle with each other and the relationships between sensors and effectors may be convoluted, thus making the system hard to maintain. However, due to ready access to all of the available system information, better configuration decisions can be made by the reasoning component (i.e., the analyzer and planner).

3.2.4 Aggregator-escalator-peer Architecture Pattern

Similar to the peer-to-peer pattern, the aggregator-escalator-peer architecture pattern decomposes the system into a set of autonomic elements. However, in this case the autonomic elements are not completely independent and not on equal footing, but are arranged in a hierarchy. Higher-level elements rely on information provided by lower-level elements to function properly. This means each feedback loop includes two types of monitors and two types of executors, including the aggregate monitor and the escalate effector. Since the aggregate monitor provides full monitor input and the escalate effector provides full executor output, the system may offer better configuration decisions and therefore achieve more efficient operation. However, each layer of monitoring comes at the cost of a performance penalty due to the overhead of the interactions made when traversing the layers. By leveraging application Pattern 3a Composed Autonomic Managers, we are able to illustrate aggregator-escalator-peer autonomic architecture pattern in Figure 3-10.

The aggregator-escalator-peer architecture pattern is,

$$\mathrm{AEP} = \{\mathrm{M}^*, \mathrm{E}^*, \mathrm{L}\} \quad (9)$$

That means several monitors and several executors are managed by a higher level monitor and a higher level executor, respectively.

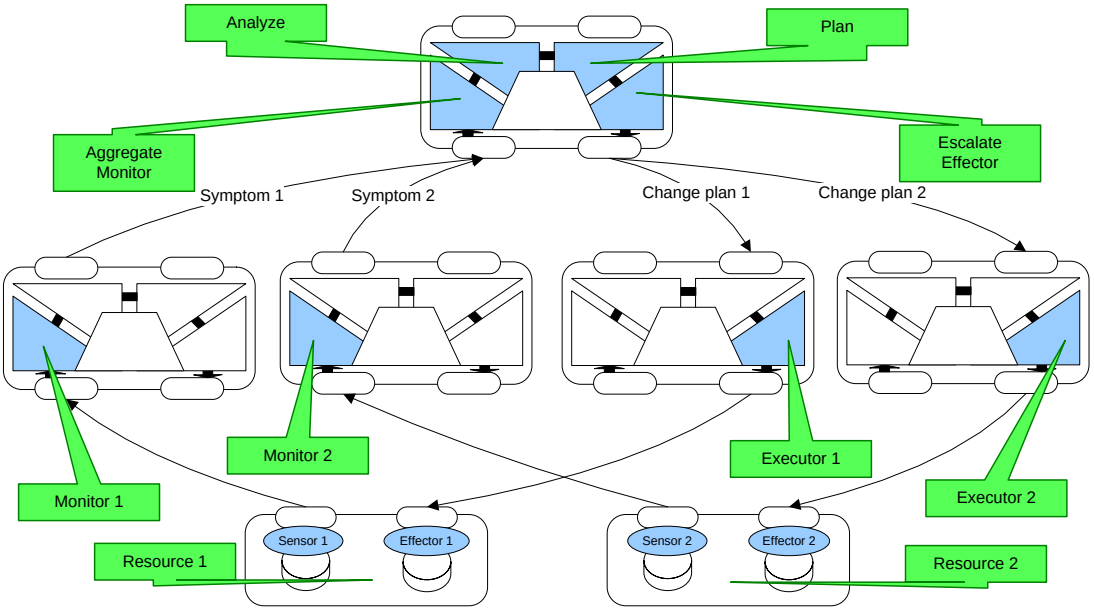


Figure 3-10: Aggregator-escalator-peer architecture pattern

The aggregator-escalator-peer architecture pattern can operate more efficiently than it would if it received and responded to each low-level monitor separately like the aspect-peer-to-peer architecture pattern does. This pattern can also be viewed as a set of autonomic elements which consists of a feedback loop for each component of the system. But in this case each feedback loop is not completely independent.

3.2.5 Chain-of-executors Architecture Pattern

The chain-of-executors architecture pattern has two variants: (1) chain-of-responsibility pattern; (2) visitor-pattern.

In the style of the chain-of-responsibility pattern variant, each executor is physically separated from its corresponding autonomic element, but all the executors are chained

together. In this case, a single sensor or a set of sensors are allocated to the entire system. For each specific problem, an optimal effect can be chosen as a solution. Figure 3-11 illustrates the first variant of chain-of-executors pattern.

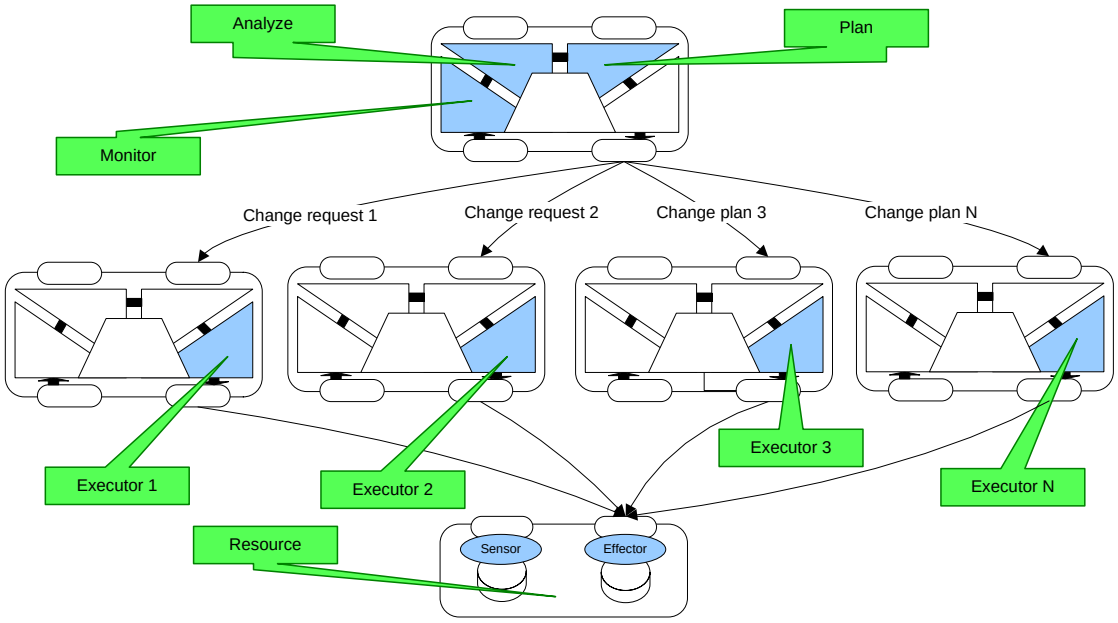


Figure 3-11: Chain-of-executor architecture pattern (chain-of-responsibility variant)

Chain-of-executor architecture pattern (chain-of-responsibility variant) is,

$$\mathrm{CECR} = \{\mathrm{M}, \mathrm{E}^*, \mathrm{A}\} \quad (3)$$

Where, E* denote multiple executors. The M represents a single monitor. The A stands for a single application logic.

One scenario is if one executor fails to handle a request, it can pass the request to the next executor in the chain. Another scenario is when each executor performs its part of the task and then passes the request to the next executor until all the actions have been performed.

In the style of the visitor-pattern variant, executor chaining is used to compose higher-order actions using a group of lower-order executors. The higher-level executor many distribute its task to all lower-level executors. In some cases, lower-level executors

might even be arranged / rearranged by their higher-level executor. Figure 3-12 illustrates the second variant of chain-of-executors pattern.

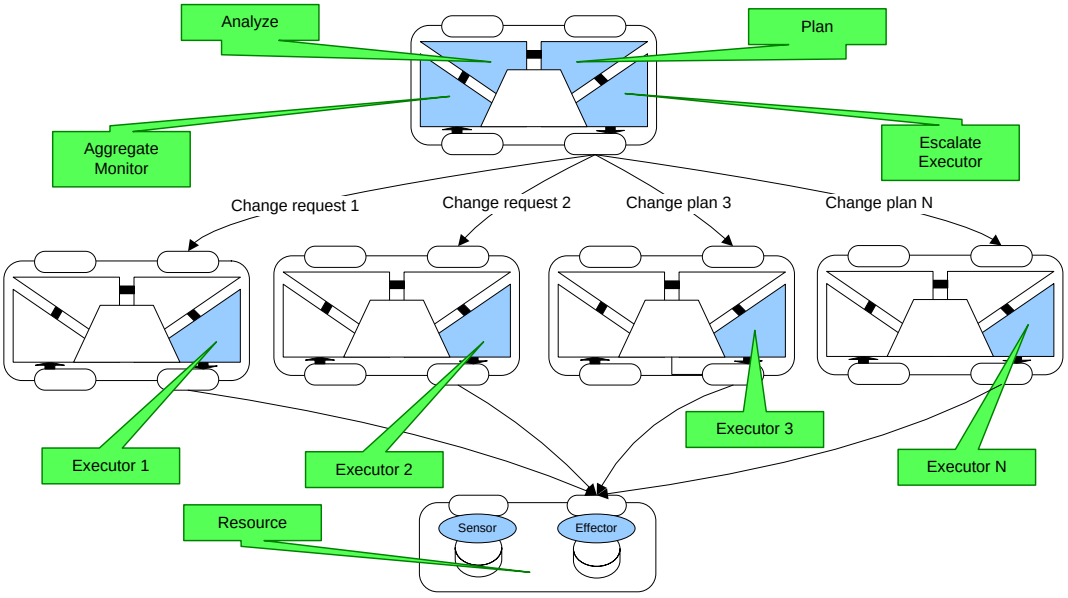


Figure 3-12: Chain-of-executor architecture pattern (visitor-pattern variant)

The chain-of-executors architecture pattern (visitor-pattern variant) is,

$$\mathrm{CEVP} = \{\mathrm{E}^{*}, \mathrm{L}\} \quad (7)$$

That means several executors are managed by a higher level executor.

The chain-of-executors architecture pattern allows a self-healing system to try all available configuration strategies to repair a given problem. The system can then promote successful strategies and demote less successful strategies while the system is running—just by manipulating its executor list.

3.2.6 Externalizing Autonomic Application Logic Architecture Pattern

If an application contains a large number of decision points encoded in different autonomic elements scattered around the application code, externalizing the self-managing logic away from application objects will relieve the burden for future maintainers. This can be achieved by representing the reasoning logic as rules and using a

rule engine to execute the logic at application decision points. A decision point is the point in an application where a decision needs to be made based on a set of conditions [ChCh03]. Figure 3-13 illustrates the externalizing autonomic application logic architecture pattern.

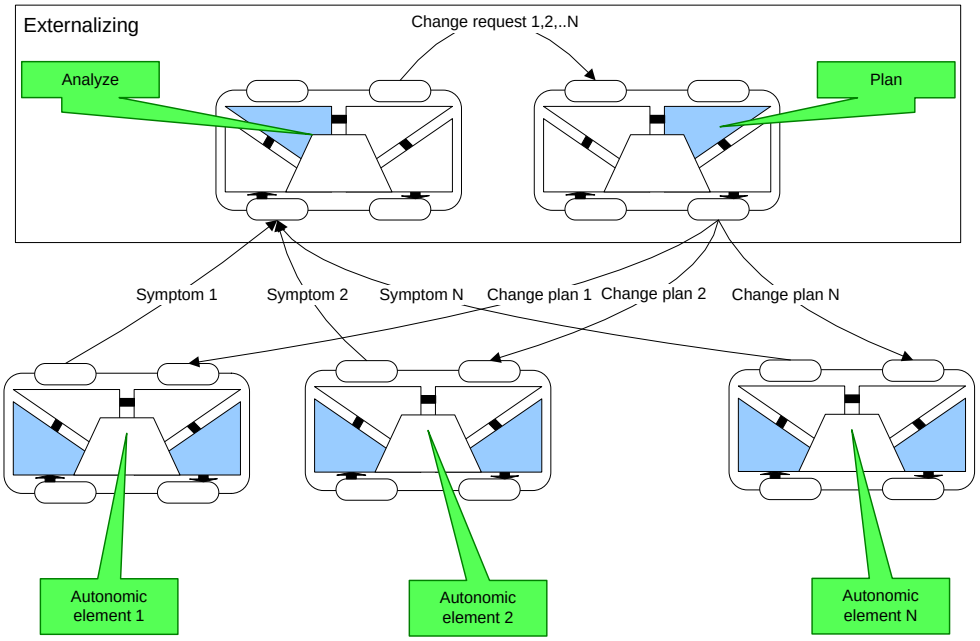


Figure 3-13: Externalizing autonomic application logic architecture pattern

The externalizing autonomic application logic architecture pattern is,

$$\mathrm{EXAL} = \{\mathrm{M}, \mathrm{E}, \mathrm{A}^{*}\} \quad (5)$$

Where, M and E denote a single monitor and a single executor, respectively. The process of externalizing application logic can be viewed / considered as decompose its rules into multiple application logics. A* represents the application logic is externalized.

This approach will improve the maintainability of the reasoning logic of the autonomic elements. In contrast, finding and managing reasoning rules if they are not externalized, such as in the case of single autonomic element pattern, becomes increasingly difficult as time passes and as the number of rules buried in the code grows.

3.2.7 Escalating Autonomic Application Logic Architecture Pattern

Decoupling the reasoning logic from the application (i.e., establishing a level of indirection) is a popular engineering solution with several key advantages.

- First, making logic explicit allows developers to understand the rules the self-managing system is currently operating under. Because the rules are external to the application that depends upon them, the variable reasoning logic contained in it can easily be changed.
- Second, externalizing logic allows users to cut the cost of application maintenance [RDRE00]. The management of the externalized reasoning rules can be done explicitly through a rule management facility. By introducing a hierarchy into the application logic, it is easier to understand what rules exist and how to locate them when changes are required. We identify "the management of the externalized reasoning rules through a rule management facility" as a new pattern: *escalating application logic architecture pattern*, which is depicted in Figure 3-14.

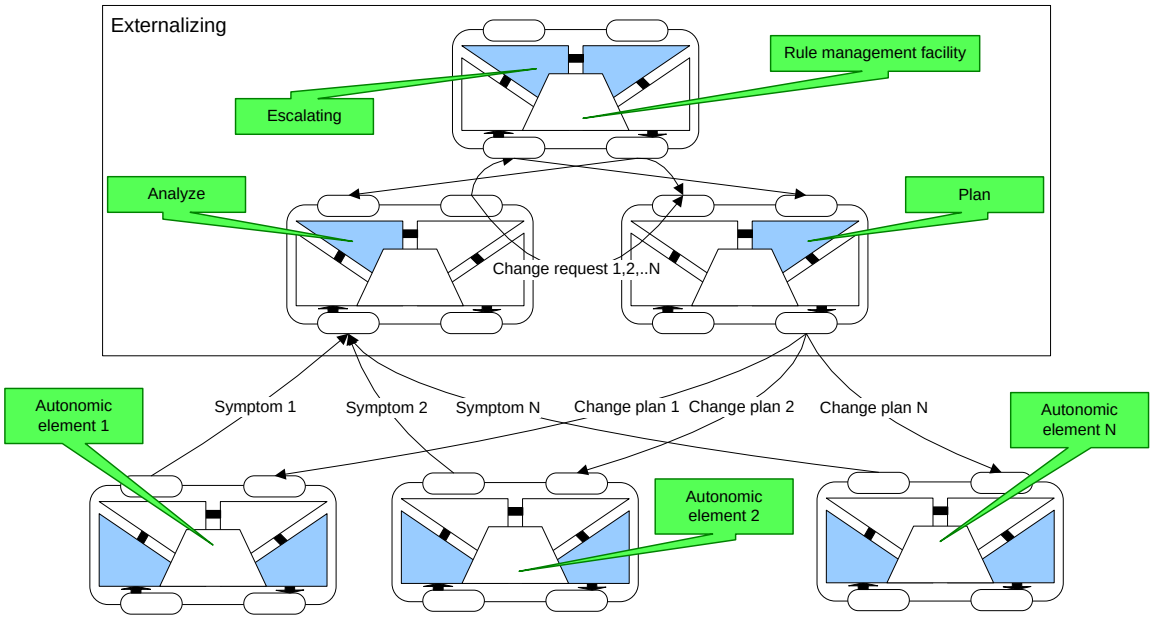


Figure 3-14: Escalating autonomic application logic architecture pattern

The escalating autonomic application logic architecture pattern is,

$$\mathrm{ESAL} = \{\mathrm{A}^*, \mathrm{L}\} \quad (8)$$

That means several application logics are managed by higher level application logic.

3.2.8 Flexible Autonomic Computing

In the most extreme case mentioned by Lapouchnian, each node in a goal model can be associated with an autonomic element to achieve the desired goal [LYLM06]. The leaf autonomic elements (i.e., the elements that correspond to the leaf nodes in the goal tree) can tune and optimize managed resources. Interior autonomic elements do not directly actuate managed components, but are used to orchestrate the leaf elements and achieve intermediate goals. The root autonomic element represents the root node of the goal model, which dictates the overall objective of the self-managed software system.

Such a flexible autonomic element solution is,

$$\mathrm{FAES} = \{\mathrm{M}^*, \mathrm{E}^*, \mathrm{A}^*, \mathrm{L}\} \quad (10)$$

That means hierarchies existing in all autonomic element phases/sectors. This form is the most flexible one.

One example of flexible autonomic computing is IBM's *autonomic computing reference architecture* (ACRA) [IBM06]. The lowest layer of ACRA contains the system components or managed resources. Managed resources usually include one or multiple embedded feedback loops implementing Application Pattern 4 *Embedded Autonomic Manager*. These components or resources incorporate standard manageability endpoints for accessing and controlling the managed resources.

Layer 2 contains resource managers. Usually system components or managed resources are visited by resource managers through ESB, which represents Application Pattern 1a *Use of Enterprise Service Bus for Manager-to-Resource Interactions*. A

particular resource may have one or more resource managers, each implementing a relevant control feedback loop. Layer 3 contains orchestrating managers. These orchestrating managers deliver system-wide autonomic capability, which is in the form of Application Pattern 2 *Manager-of-Manager Interactions*. Some systems employ CMDB as a part of Management Data Knowledge Sources, which is another way to implement Application Pattern 1b *Shared Resource Data among Managers*.

Application Pattern 3a *Composed Autonomic Managers* and Pattern 3b *Use of ESB for Composing Autonomic Managers* are really individual designer's choices (i.e., Figure 3-15), where architecture pattern aspect-peer-to-peer, aggregator-escalator-peer, and chain-of-executors, and externalizing application logic can apply.

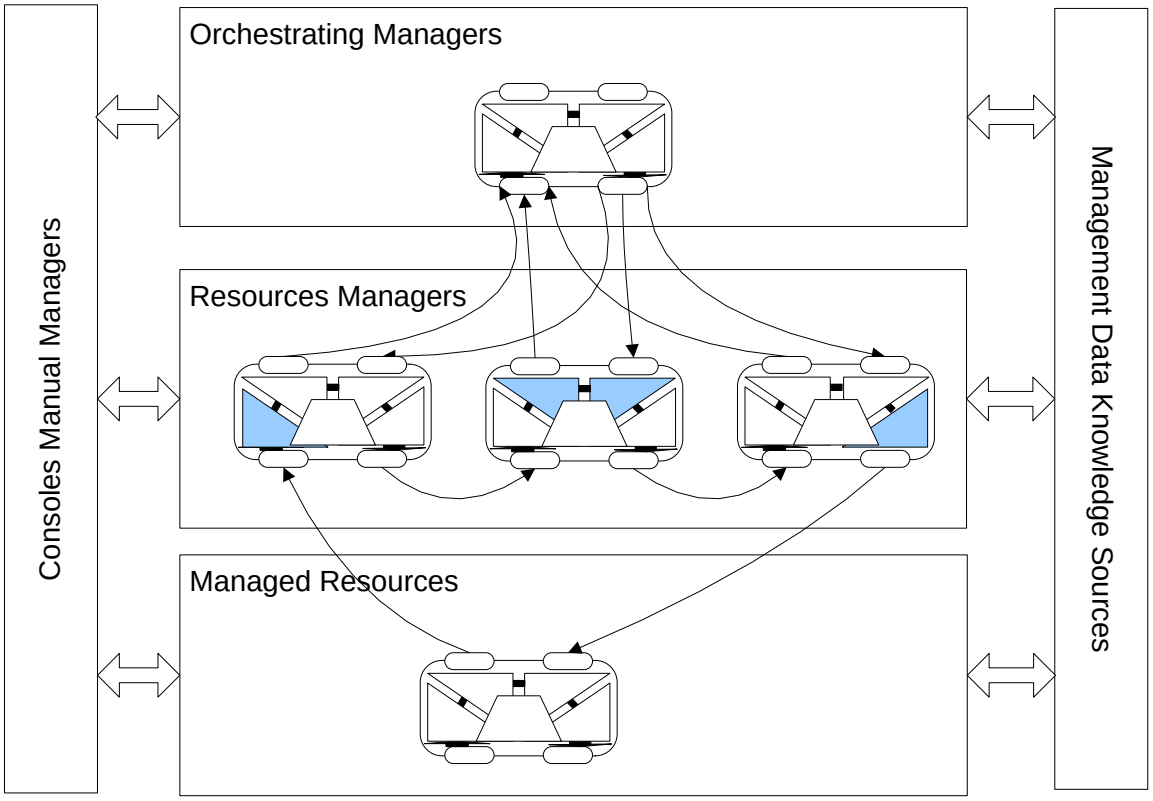


Figure 3-15: Composed autonomic managers in ACRA

3.2.9 Chain-of-monitors Architecture Pattern

From the view point of architecture patterns, there is a special case in which only monitors but not executors are arranged in a hierarchy. Hence, this special case is named as the chain-of-monitors architecture pattern. To some extent, it is more like chain-of-executors applied upon the monitor sector, which is composed of higher-order monitoring using a group of lower-order monitors. Similarly, it also has two variants: chain-of-responsibility variant and visitor-pattern variant.

The Chain-of-monitors architecture pattern (chain-of-responsibility variant) is depicted in Figure 3-16.

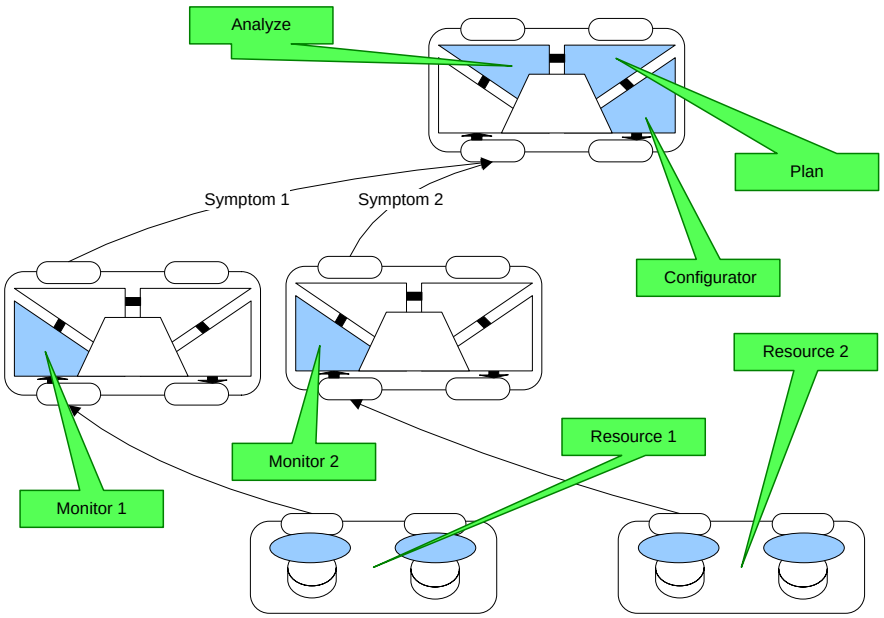


Figure 3-16: Chain-of-monitors architecture pattern (chain-of-responsibility variant)

Chain-of-monitors architecture pattern (chain-of-responsibility variant) is,

$$\mathrm{CMCR} = \{\mathrm{M}^*, \mathrm{E}, \mathrm{A}\} \quad (2)$$

Where, M* denote multiple monitors. The E represents a single executor. The A stands for a single application logic.

One scenario is if one monitor fails to find any symptom, it can pass the task to the next monitor in the chain. Another scenario is when each monitor collects part of needed information and then passes the task to the next monitor until all needed information have been collected.

The Chain-of-monitors architecture pattern (visitor-pattern variant) is depicted in Figure 3-17.

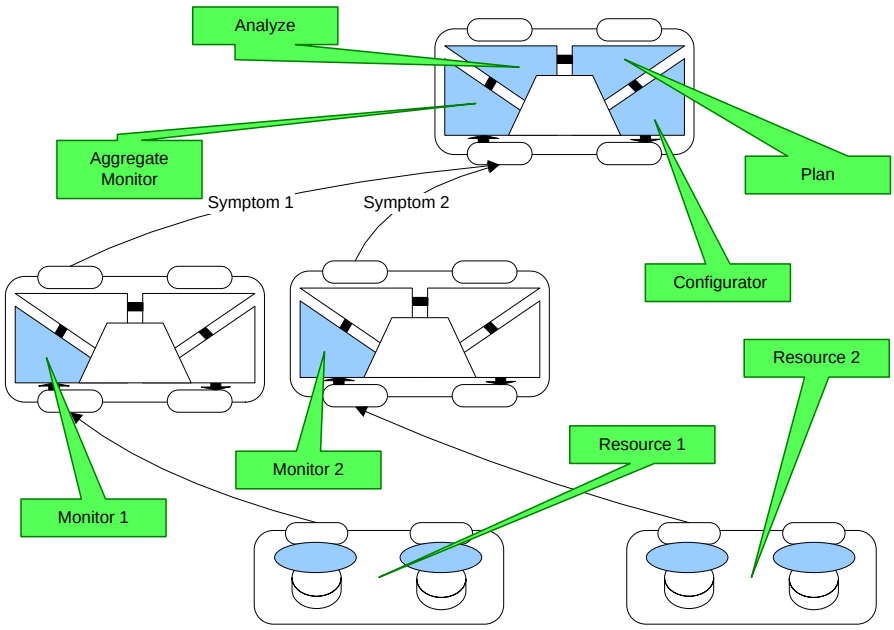


Figure 3-17: Chain-of-monitors architecture pattern (visitor-pattern variant)

The chain-of-monitors architecture pattern (visitor-pattern variant) is,

$$\mathrm{CMVP} = \{\mathrm{M}^{*}, \mathrm{L}\} \quad (6)$$

That means several monitors are managed by a higher level monitor.

In the style of the visitor-pattern variant, higher-level monitors aggregate the information provided from lower-level monitors. In some cases, lower-level monitors might even be arranged / rearranged by their higher-level monitor.

3.3 Lattice of autonomic architecture patterns

With the new chain-of-monitors architecture pattern and escalating autonomic application logic architecture pattern, the lattice of autonomic element patterns is revised/evolved into the *lattice of autonomic architecture patterns*, which is depicted in Figure 3-18.

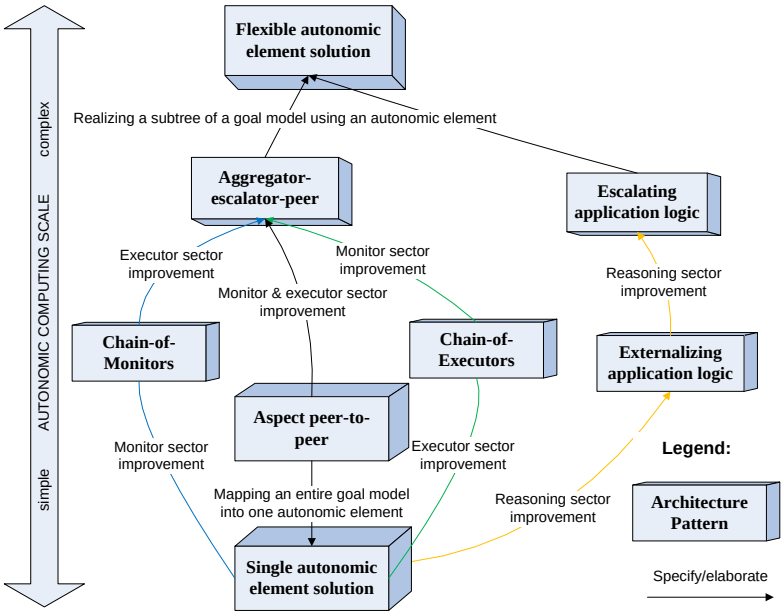


Figure 3-18: Lattice of autonomic architecture patterns

On one hand, we added the chain-of-monitors pattern as the counterpart of the chain-of-executors pattern, which makes the updated lattice of autonomic architecture patterns better balanced and more symmetric. On the other hand, we may regard the aggregator-escalator-peer pattern as the aggregation of chain-of-monitors pattern and chain-of-executors pattern.

To elaborate the order relationship among patterns, the lattice of autonomic architecture patterns can be represented as a Hasse diagram, which is depicted in Figure 3-19. The patterns presented in the lattice Figure 3-19 are qualified by the aforementioned specific feature sets (1) to (10) (e.g. the chain-of-executors is qualified by the {M, E*, A} feature set).

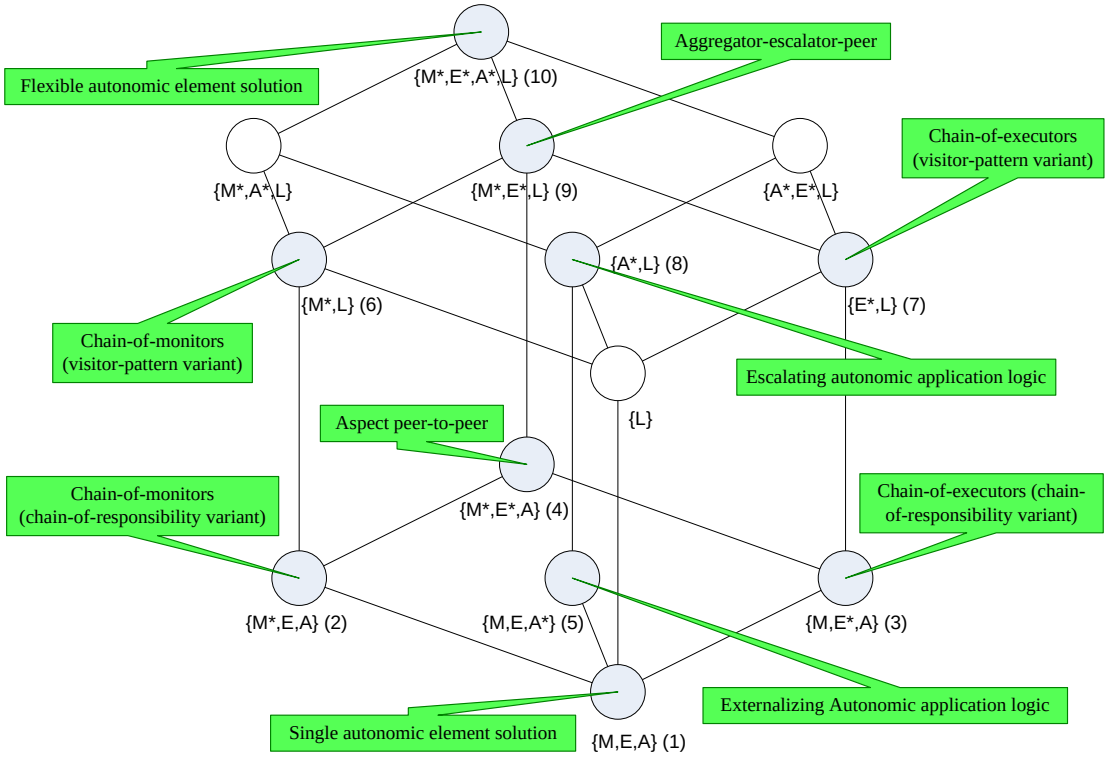


Figure 3-19: Lattice of autonomic architecture patterns represented as a Hasse diagram

Summary

In this chapter, we introduce six autonomic computing application patterns (Pattern 1a, 1b, 2, 3a, 3b and 4) identified by Sweitzer and Draper [SwDr07]. These application patterns are applied in the autonomic systems to solve specific low-level problems, such as Pattern 3a *Composed Autonomic Managers*, which describes how to use partial autonomic managers as building blocks to compose a full autonomic manager.

In our work [ZLKM08], we identified six types of existing autonomic element patterns through survey and categorised them into a lattice of autonomic element patterns, ranging from simple to complex forms. Among them, orchestrated-autonomic-computing is the most complex and sophisticated architecture pattern. One typical form of this pattern is IBM ACRA, which can include any one of the application patterns as its part to achieve a sub goal. ACRA is one of the foundations of our adaptive RCAD system

and we intend to utilize aggregator-escalator-peer architecture pattern in the middle layer of ACRA. However, we still feel that the aggregator-escalator-peer architecture pattern cannot describe our case properly.

Based on Application Pattern 2 *Manager-of-Manager Interactions* and Pattern 3a *Composed Autonomic Managers*, we are able to unambiguously illustrate a set of autonomic computing architecture patterns with structure diagrams for the first time—this is the first contribution we made in this chapter. After we illustrate these architecture patterns, we can clearly observe missed patterns related to the monitor sector. This problem is solved by adding a new architecture pattern, called *chain-of-monitors*, into the lattice of autonomic computing architecture patterns—this is the second contribution of this chapter.

Therefore, the inherent relationship (i.e., a *theory*) of all these autonomic computing architecture patterns is laid out in the lattice of autonomic architecture patterns. From validation viewpoint, first, the architecture patterns are illustrated to interpret existing forms of autonomic element arrangements (i.e., existing pattern validation). Secondly, we validated the lattice of autonomic architecture patterns with a new pattern. With small adjustments (due to a small flaw in the lattice), the lattice is able to accommodate the new *chain-of-monitor* architecture pattern. Therefore, its generality is validated.

Chapter 4 Information Seeking Process and Information Seeking Models

There are three traditional information seeking models (1) Shneiderman's Model, (2) Marchionini's Model, and (3) Hearst's Model, which are used to describe the information seeking processes for retrieving static information. However, in reality the information needed is often not static but changes dynamically throughout the process. Such an evolving search process fits better with Bates' Berry-picking model [Bat89], which is an analogy to the action of picking huckleberries or blueberries in a forest.

4.1 Information Seeking Process

When users access an information system, they often have only limited/fuzzy knowledge of how they can discover the information they want. Along with the search process, a user acquires related knowledge, accumulates useful information, and gradually understands the domain of interest, eventually finding the information required. Therefore, from a human-computer interaction point of view, the user interface of a computer system should be able to help the user to formulate his/her queries, select related information sources, understand search results, and keep track of the progress of the searches.

A user who engages in an information-seeking process usually has one or more goals in his/her mind [Hear99]. Those goals can vary quite widely, from investigating an allegation of fraud in a banking system to searching for the root cause(s) of a particular symptom in a computer network environment.

Information-seeking tasks are used to achieve those goals. A study on business analysts found three main kinds of information-seeking tasks [Hear99] (in the original paper [ODJe93] called "three basic search modes"):

- Monitoring a well-known topic over time. For example, a financial analyst calls the librarian every quarter to request a search on a competitor company.
- Following a plan or stereotyped series of searches to achieve a particular goal. For example, a process-benchmarking specialist helps his company improve their business processes in areas such as order fulfillment, and he/she follows a plan in his/her library searches.
- Exploring a topic in an undirected fashion. Before getting started with the client's problem, a manager may request general information on an unfamiliar industry.

Although the goals might be different, these information-seeking processes which aid the users to achieve their goals share one common core feature—an iterative process with different stages within it. This process can be modeled from different perspectives and granularities. Models proposed by Shneiderman et al. [SBC98], Marchionini [Mar97] and Hearst [Hear99] are known as "traditional" information-seeking models.

4.2 Traditional Information Seeking Models

4.2.1 Shneiderman's Model

Shneiderman et al. [SBC98] define information-seeking as a process comprising four stages:

- **Formulation**: What happens before the user starts a search.
- **Action**: Starting the search.
- **Review of results**: What the user sees resulting from the search.
- **Refinement**: What happens after a review of the results and before going back to formulation stage.

Figure 4-1 depicts the four stages of Shneiderman's model. Arrows indicate logical sequence of steps.

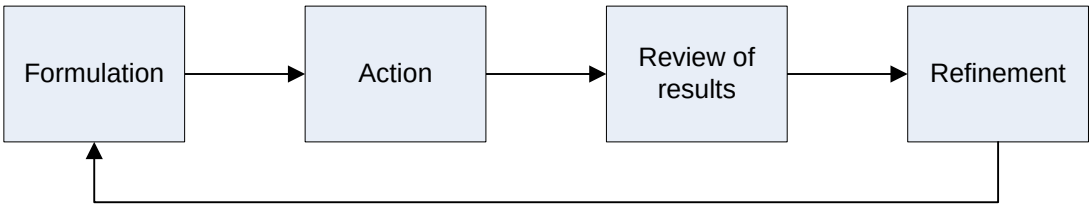


Figure 4-1: Shneiderman's information seeking model

4.2.2 Marchionini's Model

Marchionini [Mar97] proposed a more refined model to represent the information seeking process with eight stages:

- **Recognize and accept**: Aware that there is a need for search.
- **Define problem**: Decide on a plan.
- **Select source**: Where to search.
- **Formulate query**: What to search.
- **Execute search**: Start search.
- **Examine results**: What information is relevant.
- **Extract information**: Classify/copy/store useful information.
- **Reflect/iterate/stop**: Whether to refine the query and repeat the seeking process.

In Figure 4-2, boldface/thick arrows indicate logical/default sequence of steps; solid lightface arrows indicate high-probability/probable interaction; and, dashed lightface arrows indicate low-probability/possible iteration. Figure 4-2 depicts the eight stages of Marchionini's model.

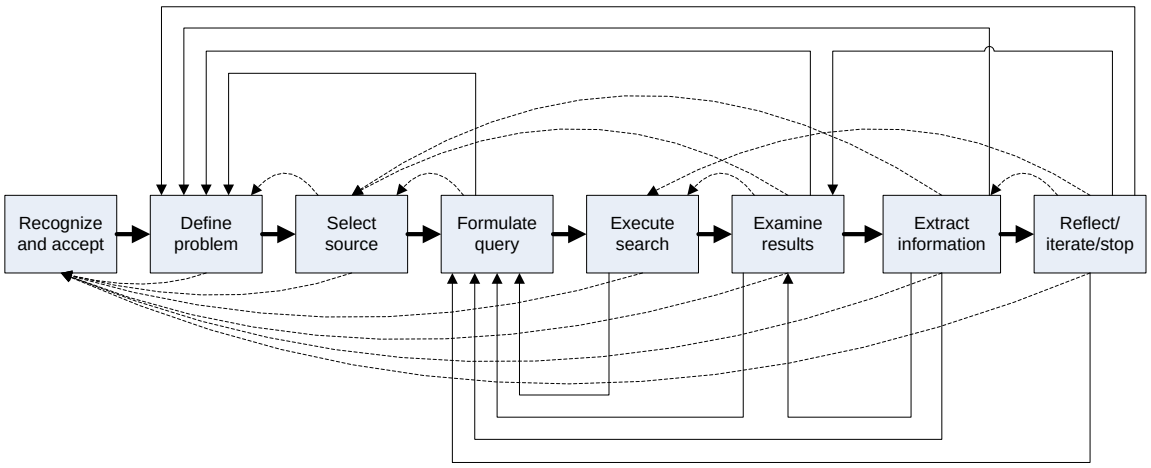


Figure 4-2: Marchionini's information seeking model

4.2.3 Hearst's Model

Hearst's [Hear99] model closely overlaps with Shneiderman's model and Marchionini's model. Figure 4-3 depicts the seven stages of Hearst's model. These stages are:

- **Information need**: Recognize there is a need for a search.
- **Query**: Decide on what to search and where to search.
- **Send to system**: Submit a query.
- **Receive result**: Receive the retrieved documents.
- **Evaluate result**: Browse and assess the retrieved documents.
- **Stop**: If the needed information is found, users end the process at this stage.
- **Reformulate**: If users are not satisfied with the information discovered, they re-specify the query and iterate the seeking process.

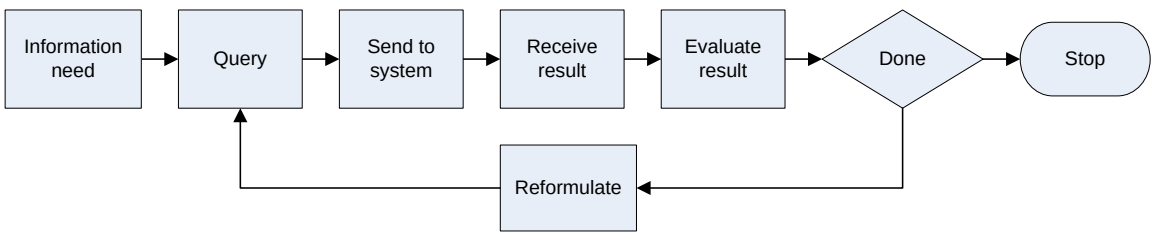


Figure 4-3: Hearst's information seeking model

4.3 Comparison of Traditional Information-Seeking Models

Hernandez discusses similarities and differences between these three traditional information-seeking models in her Ph.D. dissertation [Hern09]. A more integrated comparison via a table is presented in Table 15.

Table 15: Comparison of traditional information-seeking models

Model	Shneiderman's	Marchionini's	Hearst's
Stages	N/A	**Recognize and accept**: Users become aware that there is a need for information and decide whether to seek the answer.	**Information need**: Users recognize there is a need for information and select an information collection to search from.
	Formulation: This phase is the most complex; it involves multiple levels of cognitive processing and decisions of several types, including the sources of the search (that is, where to search); which fields in which documents to search; which text to search for; and which variants of that text to accept.	**Define problem**: In this stage users come up with a plan to solve the information problem. They identify key facts relevant to the information need and formulate hypotheses as the possible answers. The result of this stage is called a "formalized need".	**Query**: Users formulate a query by matching the information needs to the syntax and semantics specification of the selected search engine.
		Select source: Users select the source of the search. The decision is based on experience and domain knowledge.	
		Formulate query: Users construct a query based on the "formalized need" and the syntax and semantics specification of the selected search engine.	
	Action: Searches may be started explicitly or	**Execute search:** Users submit a query to the search system or	**Send to system**: Users submit a query to start

	implicitly. The typical process today is for users to click on a Search button to initiate the search, and then wait for the results.	browse the system through hyperlinks to find the desired information.	the search.
	N/A	N/A	**Receive result**: Users receive the results of their query.
	Review of results: Many information retrieval interfaces let users specify result set size (for example, a maximum of 100 documents), contents (which fields are displayed), sequencing of documents (such as alphabetical, chronological, or relevance ranked), and, occasionally, clustering (by field value and topics). All of these capabilities can be valuable in trying to make a list of documents easier to handle.	**Examine results**: Retrieved documents are examined to determine what information is relevant. As intermediate results, these documents will be evaluated by precision and usefulness of the information they contain.	**Evaluate result**: Users browse and assess the retrieved documents.
		Extract information: Users extract information from the documents that were selected as relevant in examine results stage. Typical users skim or read through the documents and then classify/copy/store the information extracted from them.	N/A
	N/A	**Reflect/iterate/stop**: Users decide whether they have found the information they search for or whether they will refine their query and repeat the seeking process.	**Stop**: If users have found the needed information, they end the process at this stage.
	Refinement: This is the stage when a user has finished reviewing the results of a search but before returning to the formulation stage to conduct a refined search.	Assumed step / implicit stage	**Reformulate**: If users are not satisfied with the information they have retrieved, they re-specify the query and iterate the seeking process.

	Relevance feedback is one of the most important ways current information retrieval technology supports progressive refinement. Another aspect of refinement is support for successive queries. As searches are made, the system should keep track in a history buffer, allowing review, alteration, and resubmission of earlier searches.		

- **Shneiderman's Model vs. Marchionini's Model**: Unlike Shneiderman's model, Marchionini's model has an initial stage in which the user *recognizes* that there is a need for information. Furthermore, Marchionini divides the *formulation* stage in Shneiderman's model into three stages: *define the problem, select the source*, and *formulate the query*. He also divides the *review of results* stage in Shneiderman's model into two stages: *examine the result* and *extract the information*. On the other hand, Shneiderman's model includes a *refinement* stage. In Marchionini's model, it is merely an assumed step.

- **Shneiderman's Model vs. Hearst's Model**: Hearst's model is similar to Shneiderman's, whereas Hearst's model calls for a *reformulate* stage and Shneiderman's calls for a *refinement* stage. Unlike Shneiderman's model, Hearst's has an extra stage, called *receive result*.

- **Marchionini's Model vs. Hearst's Model**: Although largely overlapping with Marchionini's model, Hearst's model omits the *extract information* stage and has *one query* stage instead of three small stages: *define the problem, select the source* and *formulate the query*. Unlike Marchionini's model, Hearst's has an extra stage

called *receive result*. Hearst's model includes a *reformulate* stage. In Marchionini's model, it is merely an assumed step.

4.4 Information-Seeking Model in Evolving Environment

In traditional models, the query is formulated to retrieve static information, although the query itself might be rephrased to find a better result. However, in reality the information needed is often not static but changes dynamically throughout the process, and the queries are adjusted to these changes. Users gradually learn which domain they are interested in during the search process and employ various query strategies to cope with the evolving environment and emerging directions.

4.4.1 Berry-picking Information-Seeking Model

Bates [Bat89] observed that in real-life manual source searches, users may start with one general topic or one relevant reference, and then move through a variety of information sources. Each piece of new information they encounter gives them new ideas and new directions to follow. Consequently, they have a new conception of the query. At each stage they are not just modifying the search terms used in order to get a better match for a single query, but also the query itself. This type of search is here called an *evolving search*.

The information seeking model representing this evolving search process introduced by Bates [Bat89] is called Berry-picking (as depicted in Figure 4-4), which is an analogy to the action of picking huckleberries or blueberries in a forest. The berries are scattered in the bushes and do not come in bunches. They must be picked one at a time. This bears resemblance to the evolving search processes whenever users seek and extract information from different sources in the universe of knowledge. It has two main differences from traditional information-seeking models [Hear99]:

- The user's information needs are not fulfilled by a single final set of documents, but rather by extracting bits and chunks of information along the information-seeking process.
- The user's information needs and corresponding queries are constantly changing. Information acquired at one point in a search might lead to searching towards another unanticipated direction.

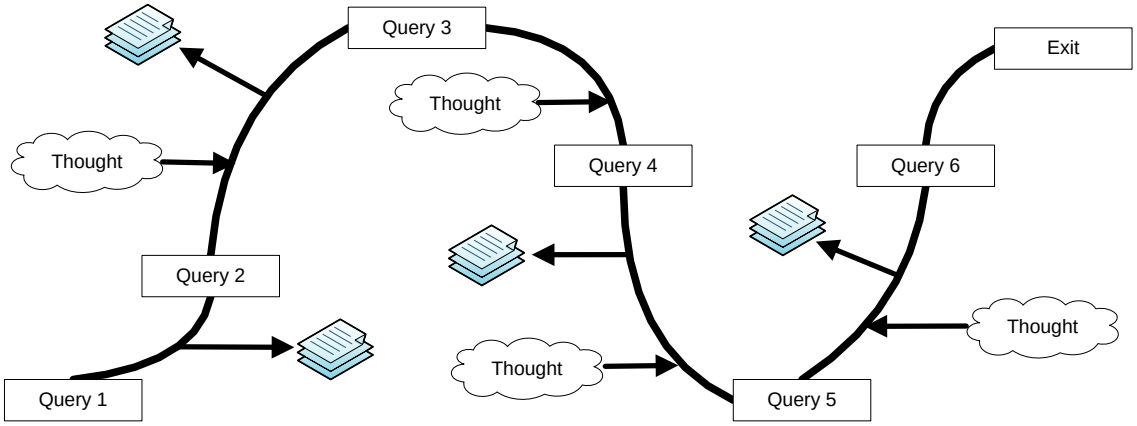


Figure 4-4: Berry-picking information-seeking model

Bates [Bat89] describes six strategies (more exist) employed by users in the information-seeking process. These six strategies are listed in Table 16.

Table 16: Six strategies in Berry-picking information-seeking process [Bat89]

Strategy	Description
Backward chaining	Typically, this technique involves following footnotes (called "footnote chasing") found in books and articles of interest, and therefore moving backward in successive leaps through reference lists. This technique is also employed by a diagnostician in the process of RCAD. When a diagnostician notifies an effect such as a set of symptoms, he or she will check all the possible causes by following certain guidelines, such as a cause-and-effect diagram. Some high-level causes might be caused by deeper-level causes. The goal is to find the root cause(s) of the present symptoms.

Forward chaining	In citation searching, a user begins with a citation, finds out who cites it by looking it up in a citation index, and thus leaps forward.
Journal run	This approach exploits Bradford's Law: the core journals in a subject area are going to have very high rates of relevant materials in that area. If the journal is central enough to the searcher's interests, this technique will have fairly good precision.
Area scanning	Browsing the materials that are closely, physically located to a source of information already identified as relevant is a widely used and effective technique.
Subject searches	Many bibliographies and most indexing services are arranged by subject. These forms of subject description (classifications and indexing languages) constitute the most common forms of "document representation" that resemble the traditional model of information retrieval discussed earlier.
Author searching	Searching "known-item" by author is an approach that contrasts with searching by subject.

In the Berry-picking model, a user can start searching with one strategy and subsequently switch to different strategies along the information-seeking process. For example, Query 1 in Figure 4-4 could use subject searching, Query 2 could employ area scanning, and then Query 3 could be formulated as the backward chaining strategy.

4.4.2 Hernandez's Integrated Model

Information-seeking is the iterative process of searching and browsing for information, which is conducted by users who often switch strategies during such a process. In Hernandez's Ph.D. dissertation [Hern09], she argues that Bates does not bring any evidence to support that the behaviour of on-line users is the same as those of manual information-retrieval users. Concerns are especially raised from the fact that Bates' Berry-picking model is based on the older studies of manually seeking information. She suggests that Pirolli's [Piro07] information-foraging theory depicts browsing activities of Internet users more properly. Analogous with the optimal foraging theory in ecology,

Pirolli suggested that human beings pursue a low cost-benefit rate by searching for the maximum amount of useful information in the least amount of time, just as predators pursue high efficiency by hunting for prey with the highest calories in the least amount of time. Based on the information-seeking models and the information-foraging model, Hernandez explicitly describes the interchange between the processes of searching and browsing in her *integrated information seeking model* (cf. Hernandez's dissertation for further details [Hern09]).

Summary

This chapter survey existing information seeking models including three traditional information seeking model (1) Shneiderman's Model, (2) Marchionini's Model, and (3) Hearst's Model, along with Bates' Berry-picking model, which represents evolving search processes and Hernandez's Integrated Model, which describes the interchange between the processes of searching and browsing. This chapter provides the foundation of proposing a new information seeking model, which is covered in Chapter 5.

Chapter 5 Net-casting Information-Seeking Model

In contrast to Bates' Berry-picking model, which describes manual information retrieval process, a new information-seeking model is needed to depict *autonomic information seeking process*. The new model has five stages, which arises from three traditional information seeking models introduced in Chapter 4. Proposing this model is inspired by Hernandez's work, that deals with similar shortcomings about existing information seeking models (including Bates' Berry-picking model) not being able to describe the new form of information-retrieval behaviour. However, the new model itself is completely different from Hernandez's Integrated Model.

5.1 Introduction of Net-casting Information Seeking Model

Most descriptions of the information-seeking process assume an iteration cycle consisting of query formulation, search execution, receiving and examining results, and then either stopping or reformulating the query and repeating the process until a perfect result set is found. Traditional information-seeking models describe this iteration cycle (with multiple stages) in various perspectives and granularities. They are all extremely useful in designing information seeking processes. However, there are some drawbacks with those models as Hearst points out [Hear99]:

- There is an assumption that a user's information need is static,
- The retrieved results do not directly address the information needs,
- The interaction and role of source selection is downplayed.

Bate's Berry-picking model suggests that these interaction processes are not simply repetitive with respect to search strategy and target information, which means, in an evolving environment, each iteration might use a different strategy to find the information users need. When the notion of a Hyperlink becomes an essential part of the

information-seeking process, it becomes impractical to neglect the role of browsing and navigating within the information-seeking process. By presenting an integrated information-seeking model, Hernandez intends to make the role of browsing and navigating more explicit within the information seeking process, although the integrated model itself needs more improvement and perfecting [Hern09].

From a human-computer-interaction point of view, there is a wide variety of information seeking processes spanning the spectrum from heavily interactive processes with intensive human intervention to light interactive processes with little human intervention—hence *autonomic* information seeking process. In the latter case, the lion's share of the workload especially some routinely performed tasks shift from human operators to autonomic computing systems. A new information-seeking model is needed to depict this type of information-seeking process. This model is created based on the aforementioned existing information seeking models and observations. I name it the **net-casting information-seeking model** or **net-casting model.**

Figure 5-1: Throwing the cast-net from shallow waters, 19th century drawing[5]

[5] Fishing in the Bible and the Ancient Near East, http://thewikibible.pbworks.com, 2008.

A cast net, also called a throw net, is a simple device that has been used for thousands of years for shallow-water fishing as depicted in Figure 5-1. It is a circular net with small weights attached around its edge. The net is cast or thrown by hand in such a manner that it spreads out on the water and sinks. Fish are caught as the net is hauled back in. This technique is called net-casting or net-throwing[6]. There are two main actions in this technique: throw a net and haul it back. It shows resemblance to the two main steps in an autonomic information seeking process: search and retrieval information. Typical actions of these two steps are: formulate queries and review of results. In analogy, these two actions echo the two main actions in the net-casting fish technique: (1) prepare a net and throw it; then wait (no action) until the net has sunk; (2) haul back the net and count fish.

Based upon studying and comparing traditional information-seeking models, we propose that a net-casting information-seeking model should include iterative processes that consist of five stages: recognize the problem, formulate queries, review the results, treat the problem and reformulate as depicted in Figure 5-2. If users are not satisfied with the outcome of the current treatment, or a new problem is emerging in reformulation stage, the information-seeking process will start another cycle of iteration.

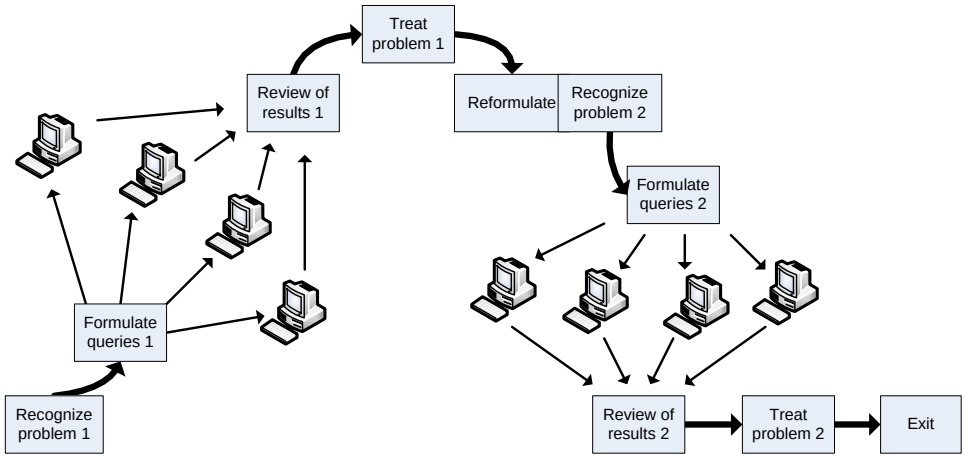


Figure 5-2: Net-casting information seeking model

[6] E. B. Burnley, *Surf Fishing the Atlantic Coast*, Stackpole Books, 2006.

The stage names and tasks of these five stages—recognize problem, formulate queries, review of results, treat problem and reformulate, are listed in Table 17.

Table 17: Five stages of the net-casting model

Stage	Tasks	I/O Events
Recognize problem	**Recognize and accept**: Be aware that there is a symptom and decide whether to seek the root cause.	**Input:** Symptom **Output:** Hypothesis
	Define problem: Identify key facts relevant to the information need and formulate hypotheses as the possible answers.	
Formulate queries	**Select source:** Select the source of the search. The decision is based on a pre-defined knowledge base.	**Input:** Hypothesis **Output:** Check or Inquire
	Formulate queries: Construct queries based on information needs and the syntax and semantics specification of selected search engines.	
	Execute search: Submit a query to the search systems.	
Review of results	**Receive results**: Users receive the results of their query.	**Input:** Diagnose or Result **Output:** Cause
	Examine results: Examine retrieved information and determine if this information is sufficient for diagnosis.	
	Aggregate results: Users aggregate candidate diagnoses from different sources and make a conclusion.	
Treat problem	**Treatment:** Apply treatment according to the cause found.	**Input:** Cause **Output:** Treatment
Reformulate (repeat first Stage: **Recognize problem**)	**Reflect/iterate/stop**: Users decide whether they have found the solution they searched for or whether they will refine their query and repeat the seeking process.	

5.2 Characteristics of Net-casting Information Seeking Model

In the autonomic computing context, the net-casting model differs from previous information-seeking models as follows:

- Information-seeking processes are undertaken with minimal human intervention.
- Instead of seeking information from various resources for the same information, net-cast model searches different information from different locations for a set of hypotheses. The verification of these hypotheses will lead to the answer of the original problem.
- If evidence of the hypotheses can be collected and verified in parallel, they could be undertaken simultaneously.
- Unlike the Berry-picking model which employs many types of query strategies and users can switch to different strategies as they wish, the net-casting model has pre-defined and pre-coded search strategies and the choice of the strategy will be decided by the system and accomplished automatically in real-time.

In the net-casting model, some of the workload is shifted from human operators to the autonomic system. For example, in some cases of the Berry-picking model, users may encounter a "trigger" that causes them to pursue a different strategy temporarily and return to a currently unfinished activity at a later time. Hearst mentioned GUI should support methods for monitoring the status of the current strategy in relation to the user's current task and a high-level goal [Hear99]. This kind of approach assumes that at any point of the search process, if another strategy presents a higher utility than the current one, the current strategy should be abandoned in favour of the new strategy. In autonomic computing, a goal tree with soft-goals defined by cost/benefit rates will make a search process abandon the current strategy and yield to a new strategy, as the new strategy presents a lower cost/benefit rate.

In traditional information-seeking models, the first step in performing a search is to decide where to search (sources). The location is often not just a single physical database but rather multiple and distributed databases which are located across a network. Even if it were technically and economically feasible, searching all libraries or all collections in a library is often undesirable. When users know where the relevant material is, they generally prefer to limit their searches to a limited number of libraries or collections or range of documents. In the net-casting model, different information needs are pursued simultaneously at different locations which are located across a network. Each search might employ a search strategy predefined by users, just like fishing nets with different mesh sizes being knitted before catching fish. There is no guarantee that a desired answer will be found at the end of one search iteration, just like no fish is guaranteed to be caught after the fishing net is cast.

Table 18: Comparison of traditional, Berry-picking and Net-casting information-seeking models

Information seeking models	Traditional models	Berry-picking model	Net-cast model
Nature of information	static	dynamic	static or dynamic
Nature of process	manual	evolving	autonomic
Human intervention	heavy	heavy	light
Query evolving	weak	strong	strong
Parallel queries	weak	weak	strong

Table 18 is a small comparison of our net-casting model and models introduced in Chapter 4. Generally, net-casting model can display more spatial information, and Berry-picking model can display more temporal information. Using a simple metaphor, traditional models are able to depict information seeking processes with “still pictures”, while Berry-picking model depicts “motion pictures” and net-casting model depicts “3D pictures”. We might still use a traditional model to describe an autonomic information seeking process. But such a model won’t be able to capture all details regarding multiple queries per iteration and query evolving between consequent iterations of the autonomic information seeking process.

5.3 Human Aspect in Information-Seeking Model

Research on information seeking, derived from information science, has two equally important aspects: a human aspect and a technological aspect [Hear99]. In non-interactive (between human and computers) systems or light interactive processes such as net-casting models, *precision* and *recall* are common measurements used for comparing the ranking results and observing the system as a whole. The standard evaluations emphasize high recall levels and systems are often compared to see how well they return the desired result.

However, in interactive settings, users only need a few relevant documents and do not care about high recall to evaluate the information-seeking systems. As human beings are more complex than computers, their motivation and performance are more difficult to measure and characterize. Evaluation criteria based on human needs are beyond precision and recall. They include: learnability (time required to learn the system), efficiency (time required to achieve goals on benchmark tasks), effectivity (low error rates), and re-accessibility (retention of user interface over time), and commonly used to evaluate systems from the viewpoint of human-computer interaction.

Observing the three main kinds of information-seeking tasks mentioned in Section 4.1, the net-casting model is suitable to the first two types: “monitoring a well-known topic over time” and “following a plan or stereotyped series of searches to achieve a particular goal”. It can automate these types of processes and relieve humans from some routinely-performed tasks. However, the third type of task, “exploring a topic in an undirected fashion,” still largely relies on a human’s brain power. A berry-picking model will be better to describe the evolving search processes.

Summary

So far, there is no existing information-seeking model that can be used to reflect the autonomic-information-seeking process (in contrast, Bate's Berry-picking model describes the interactive information-seeking processes with intensive human intervention [Bat89]). As the main contribution of this chapter, a new information-seeking model called the net-casting model is proposed to reflect the autonomic-information-seeking process—a process with minimal human intervention.

After identifying the commonality amongst three traditional information-seeking models through comparison, we propose that the new net-casting model should include five stages: recognize the problem, formulate queries, review the results, treat the problem and reformulate.

Chapter 6 Conceptual RCAD Architecture

Today, when IT administrators for enterprise application systems want to view events of warning or alert information about their web application environment, such as high CPU utilization, low disk space, or Web/database server overloads, they have a variety of application management tools to choose from, such as CA's Wily Introscope [CA10]. But administrators still need to investigate the fundamental causes arising from the information that these tools collect. IT administrators must be able to infer the problems (diagnose) in the system with their brain power and to take appropriate actions (either treatment or preventative) accordingly. The process of inference is called *root cause analysis and diagnosis (RCAD)* process.

6.1 Diagnosis Related Concepts

The word *diagnosis* is closely related to medical practice. Rather than invent new nomenclature to describe the RCAD process, we try to study diagnosis-related concepts from medical perspective first, and then "borrow" a few of them from the nomenclature of medicine. In the medical context, *diagnosis* refers both to the process of attempting to determine the identity of a possible disease or disorder and to the opinion reached by this process. The term *diagnostic criterion* designates the combination of signs, symptoms, and test results that the clinician uses to attempt to determine the correct diagnosis. The plural of diagnosis is *diagnoses*, the verb is *to diagnose*, and a person who diagnoses is called a *diagnostician*.[7]

The key diagnosis-related concepts from medical perspective are defined as follows:

Sign: is an objective indication of some medical fact or characteristic that may be detected by a physician during a physical examination of a patient. Signs may have no

[7] Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Medical_diagnosis

meaning for, or even be noticed by, the patient, but may be full of meaning for the healthcare provider, and are often significant in assisting a healthcare provider in diagnosing medical condition(s) responsible for the patient's symptoms. Examples include elevated blood pressure, a clubbing of the fingers (which may be a sign of lung disease, or many other things). [8]

Symptom: a departure from normal function or feeling which is noticed by a patient, indicating the presence of disease or abnormality. A symptom is subjective, observed by the patient, and not measured. [9]

Syndrome: Refers to the association of several clinically recognizable features, signs (observed by a physician), symptoms (reported by the patient), phenomena or characteristics that often occur together, so that the presence of one feature alerts the physician to the presence of the others. In recent decades the term has been used outside of medicine to refer to a combination of phenomena seen in association. The description of a syndrome usually includes a number of essential characteristics, which when concurrent lead to the diagnosis of the condition. Frequently these are classified as a combination of typical major symptoms and signs—essential to the diagnosis—together with minor findings, some or all of which may be absent. A formal description may specify the minimum number of major and minor findings respectively, which are required for the diagnosis. [10,11]

Prescription: A health-care program implemented by a physician or other medical practitioner in the form of instructions that govern the plan of care for an individual patient. Commonly, the term prescription is used to mean an order to take certain medications. Prescriptions have legal implications, as they may indicate that the

[8] Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Medical_sign

[9] Answer.com, http://www.answers.com/topic/symptom

[10] Answer.com, http://www.answers.com/topic/syndrome

[11] Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Syndrome

prescriber takes responsibility for the clinical care of the patient and in particular for monitoring efficacy and safety. This implies continuous monitoring and treatment [BLMY08].

The OASIS white paper authored by CA Inc., IBM and Fujitsu researchers [BBDL10], presents the vision for the *symptoms framework* (SF), a specification that enables the automatic detection, optimization, and remediation of the operational aspects of complex systems with applicability to both IT and non-IT domains. As expected, the symptoms framework also use the mature, well-understood and tested nomenclature of medicine, somewhat simplified and customized for its purposes. A *symptom*, as used by this framework, is an indication of an observed or self-reported negative or positive condition—an indication that one or more events of potential interest have happened. A symptom might correspond to one or more *syndromes* which contain a set of possible *protocols*, templates for performing diagnostic tests and treatments (when the condition is negative) or actions (when the condition is positive). Protocols will turn into concrete *prescriptions* that apply to the particular situation, when specific information (arguments) is supplied about the target of the prescription.

Symptom: An indication of an observed or self-reported negative or positive condition—an indication that one or more events of potential interest have happened based upon a signature (pattern) recognized in a corresponding syndrome [BBDL10].

Syndrome: A description and signature (pattern) that identifies when an occurrence of related symptoms might exist. A syndrome may also refer to protocols that specify diagnostics that might confirm that an instance of a syndrome exists or actions to remediate, improve, or recognize the condition [BBDL10].

Protocol: a template for a process that may be used to further remediate, improve, encourage or prevent, or simply further confirm the possibility of a syndrome [BBDL10].

Prescription: A prescription is a concrete instance of a process (possibly corresponding to a protocol) with specific arguments about the subject or component that is the target of the process [BBDL10].

Although the nomenclatures in both domains are almost identical, the taxonomies of symptom, syndrome and prescription in the OASIS symptoms framework are slightly simplified and customized from those ones in medical practice (i.e., the taxonomies in the OASIS symptoms framework are subsumed under the taxonomies in medical practice) . There is no *sign* concept in symptoms framework. There is no difference between *minor findings* and *major finds* in syndrome defined by symptoms framework. Furthermore, the prescription of a protocol in symptoms framework includes both *diagnostic tests* and *treatments* and the concept of protocol itself doesn't exist in the conversional nomenclature of medicine.

6.2 Medical Illustration of Symptom Concepts

The OASIS white paper authored by CA Inc., IBM and Fujitsu researchers [BBDL10] uses a common, easily understood medical scenario to introduce the key concepts [BBDL10]. For example (slightly simplified from [BBDL10]), there is a person who may have a fever. Fever is a *symptom* of many *syndromes*, from influenza to much more serious infections. Clearly, a fever might be indicated by a number of observable and reported events, although it may not be initially clear that they are related or what form these events will take.

Initially, this person might complain of a headache to the nurse. The nurse might also note that the subject is irritable (emit a symptom on behalf of the subject). These two symptoms (irritability and headache) by themselves may not be much cause for alarm, but now acting in the role of diagnostician the nurse scans her mental *syndrome catalogue* for the best *signature* (a pattern) that matches those two events. The syndrome

of fever has a high likelihood, so the nurse again acts in the role of diagnostician to further verify the *fever syndrome*.

The nurse knows that the signature of the fever syndrome specifies a measurable high body temperature and activates the appropriate protocol. The protocol's prescription in this case is diagnostic. She takes the patient's temperature using a thermometer (acting as fever symptom source). The fever syndrome is verified. She creates a medical record of the fever symptom for this patient. The nurse, now in the role of practitioner, initiates a different kind of prescription, more treatment-oriented than diagnostic. The nurse refers the patient and his medical record to a doctor. In other words, the nurse passes on the subject's fever symptom to a different diagnostician who will review the fever symptom and possibly take further action.

Taking this process one step further illustrates how the symptoms framework can be applied to even higher-level abstractions rather than being confined to just measurable, observable events. The doctor has a more sophisticated and detailed catalogue of syndromes including one for influenza. He is aware in this case that merely treating the fever symptom is not an adequate response. Other symptoms need to be investigated. Influenza must be deduced based on the possibility of certain other symptoms occurring together (the signature of influenza) such as cough and nasal congestion. Acting as diagnostician, the doctor realizes that indicate the patient has both of these symptoms as well.

Lastly, the doctor issues a prescription, either acting in the role of practitioner personally or perhaps requesting the action from a colleague or nurse acting in the role of practitioner, to put the subject on an anti-viral regiment due to the higher risks associated with the patient.

It is important to note that there are two circles within this process—the iterative cycle taken by the nurse (cf. black text in Figure 6-1) and later examining doctor (cf. brown text in Figure 6-1), both acting in the role of diagnostician and possibly

practitioner. Both have different levels of knowledge and access to different policy and case histories. In each circle, the nurse and examining doctor follow the scientific method, which generally includes these three steps: (1) hypothesis generation, (2) conducting experiments, and (3) observing effects [Zell06].

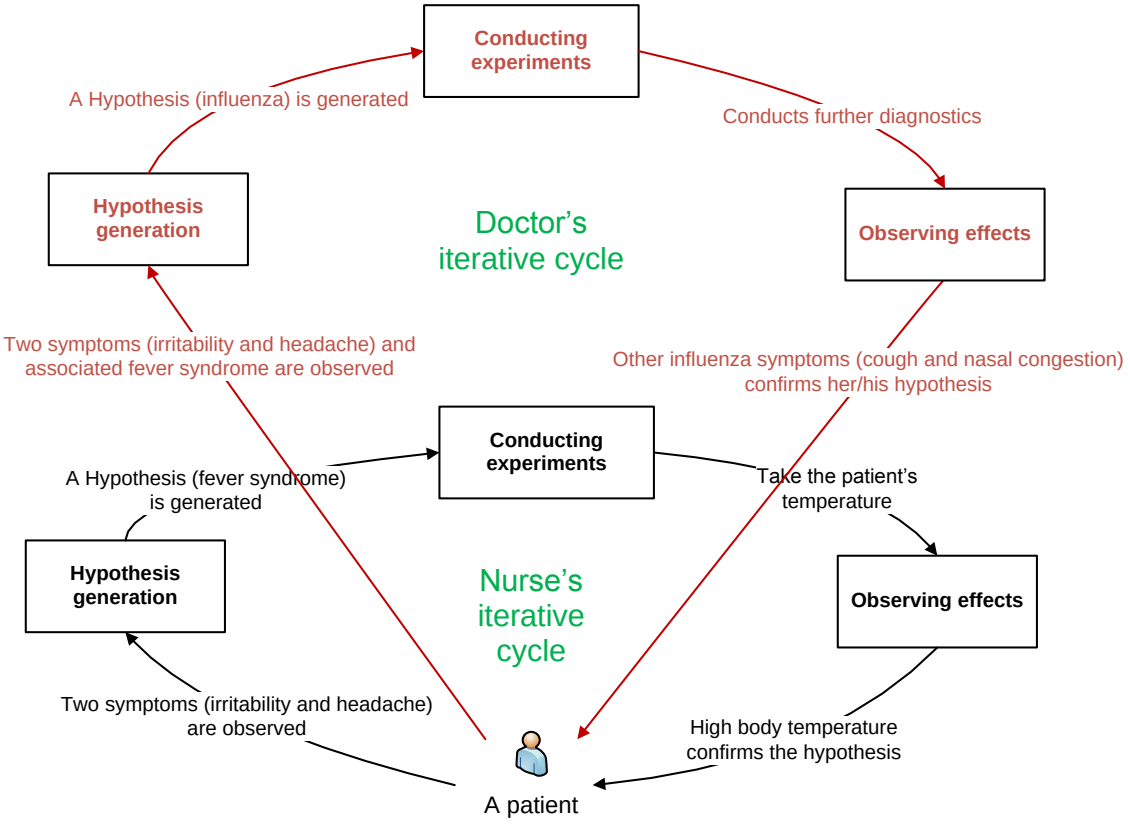


Figure 6-1: Two iterative circles within the diagnosis process

In the first circle, (1) a hypothesis—fever syndrome is generated by the nurse according to two symptoms (irritability and headache). And then (2) the nurse conducts an experiment—taking the patient's temperature using a thermometer. Finally (3) the nurse observes the effect—the patient has a high body temperature, which confirms her/his hypothesis.

In the second circle, (1) a hypothesis—influenza is generated by the doctor according to two known symptoms (irritability and headache) and associated fever syndrome. And

then (2) the doctor conducts further diagnostics. Finally (3) the doctor realizes that the patient has other influenza symptoms (cough and nasal congestion) as well, which confirms her/his hypothesis.

Although the proposed OASIS symptoms framework proposed is designated to model more complex RCAD processes in both IT and non-IT systems, a typical RCAD process in IT settings such as the use case introduced in the following section is similar to the medical scenario described above.

6.3 A Use Case from CA Inc.

Today, most RCAD processes in IT settings are carried out by administrators following standard scenarios. Figure 6-2 displays a real world scenario: *Use Case One* provided by CA Inc.

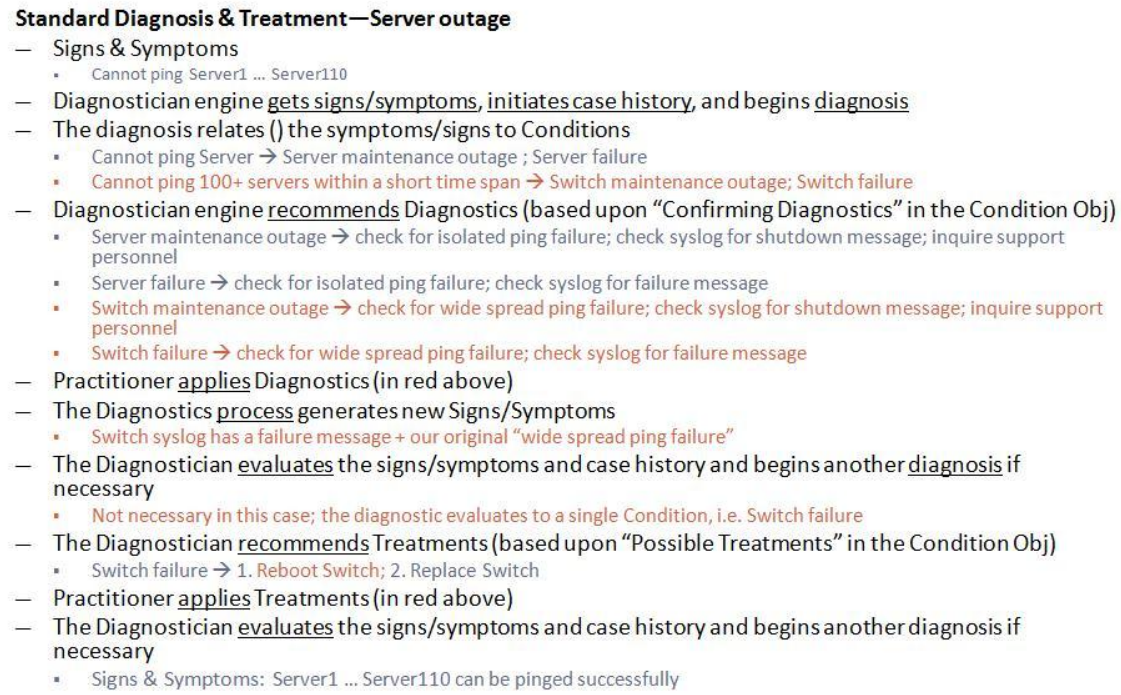


Figure 6-2: Use Case One from CA Inc.

When investigating Use Case One closely, we realize that this is a workflow which describes the scenario of diagnosing a *server outage* problem in an enterprise application environment. The workflow can be interpreted as follows:

(1) When a network administrator finds himself / herself failing to ping a server, a standard diagnosis and treatment process will be undertaken as follows:
   (a) If there is only one server he / she cannot ping, then the potential root cause could be *server maintenance outage* or *server failure*;
   (b) If the administrator cannot ping a wide range of servers, the possible root cause could be *switch maintenance outage* or *switch failure*. The situation of multiple servers failing in a short period of time (almost at the same time) is highly unlikely and can be neglected in this scenario.

(2) The investigation process, therefore, leads to two separate investigation paths: one is (cf. black text in Figure 6-2) the investigation of the server problems (follows (a) above); and the other (cf. brown text in Figure 6-2) is the investigation of the switch problems (follows (b) above). In both of these cases, collecting more data/evidence is essential to perform the diagnosis.

(3) The scenario that takes place when the symptom (a) is discovered is described as follows. If a network administrator diagnoses that it is truly an isolated ping failure and there is also a shutdown message in the *syslog (system log files)*, the diagnosis would be *server maintenance outage*. The network administrator can also ask support personnel to discover the real problem (about the server maintenance outage). If the network administrator identifies an isolated ping failure and there is a failure message in the *syslog*, the diagnosis would be *server failure*.

(4) The scenario that takes place when the symptom (b) is discovered is described as follows. If the network administrator finds a widespread ping failure and there is a shutdown message in the *syslog*, the diagnosis would be *switch maintenance*

outage. The network administrator can also ask support personnel to find out the truth (about the switch maintenance outage). If the network administrator discovers the ping failure is widespread and there is a failure message in the *syslog*, the diagnosis would be *switch failure*. In this case the recommended treatments would be: *reboot the switch* or *replace the switch*.

Use cases like this one are still carried out by IT administrators manually today. Similar use cases (i.e., Use Case Two and Three) are presented in Appendices B and C.

6.4 Conceptual RCAD Architecture

Our vision is: making the RCAD tasks that are still carried out by an IT administrator manually today fully autonomic. Figure 6-3 illustrates today's state-of-the-art system management.

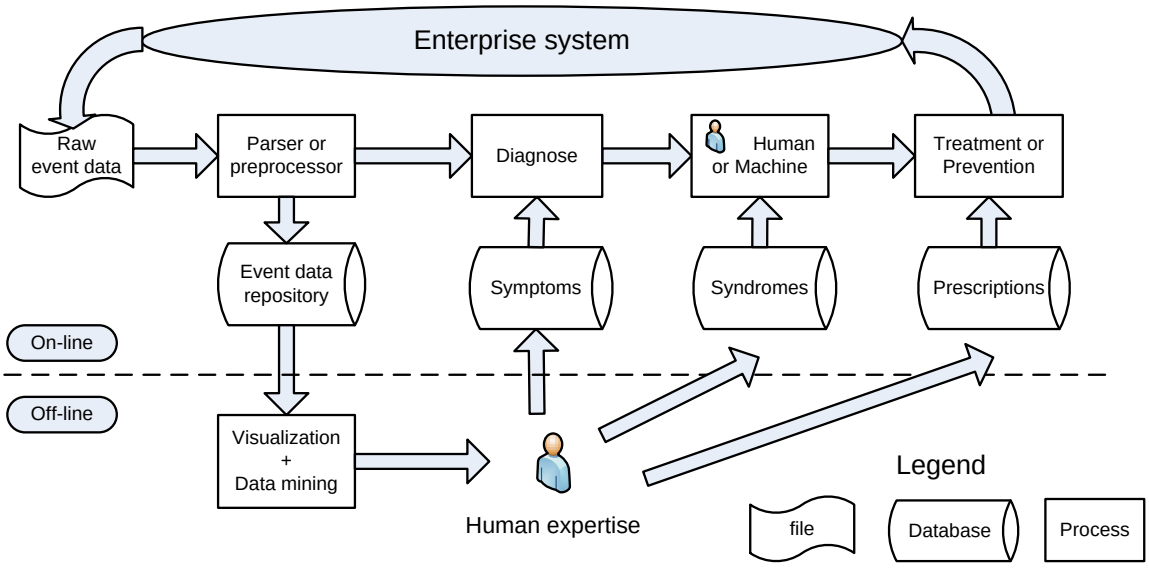


Figure 6-3: Conceptual RCAD architecture

The area below the dotted line depicts the rule construction part. Raw events flow into an event parser or a pre-processor, where they are parsed and unrelated information

is filtered. Then, one destination of these events is stored in an event-data repository and ready for off-line rule construction. Constructed rules include symptom definitions, syndrome definitions and prescription definitions. These rule constructions are generally achieved by human experts where they often are assisted by two sets of technologies. The first is to visualize large volumes of event data to aid in determining relationships among events. The second employs data mining techniques to automate the search for patterns. These two approaches can be used separately, but they are more effective when employed in combination.

The area above the dotted line depicts the conceptual RCAD architecture adopted by many researchers and industrial companies such as IBM and CA Inc. There is the other destination of pre-processed raw events—interpreted by a correlation engine (using correlation rules) in real time. Some events are filtered, while others are amalgamated. The results are symptoms, or other kinds of synthesized events. Correlation rules used by the correlation engine are usually structured as "if-then" statements. The if-part (or left-hand side) describes a situation in which actions are to be taken. The then-part (or right-hand side) details what is to be done when the condition is satisfied. For example, IBM provides symptom catalogues (available at the IBM Tivoli Open Process Automation Library[12]) for problem determination triage. Hundreds of symptoms have been defined in these catalogues and straightforward reactions can be taken when those symptoms are detected by correlation engines inside autonomic tools such as Log and Trace Analyzer for Java Desktop (LTA-JD).

However, these automated problem processes are limited only to single symptom detection/diagnosis. More complex RCAD processes for a set of symptoms (called syndrome) still need human engagement by following RCAD workflow scenarios or fishbone diagrams, even with tools such as the one Kontogiannis and Wong built (cf. Figure 7-3). Our vision is to push the RCAD processes one step further towards

[12] Build a framework for problem determination triage, https://www.ibm.com/developerworks/autonomic/library/ac-pdtriage1/

autonomic operation, to be able to automate the processes of needed information collection (the information operators gather to carry out an RCAD task is not always at the right abstraction level), to carry out each branch of the use-case workflow simultaneously, and to make the RCAD process for an individual syndrome autonomic. In particular, the use-case workflow should be managed by an autonomic element.

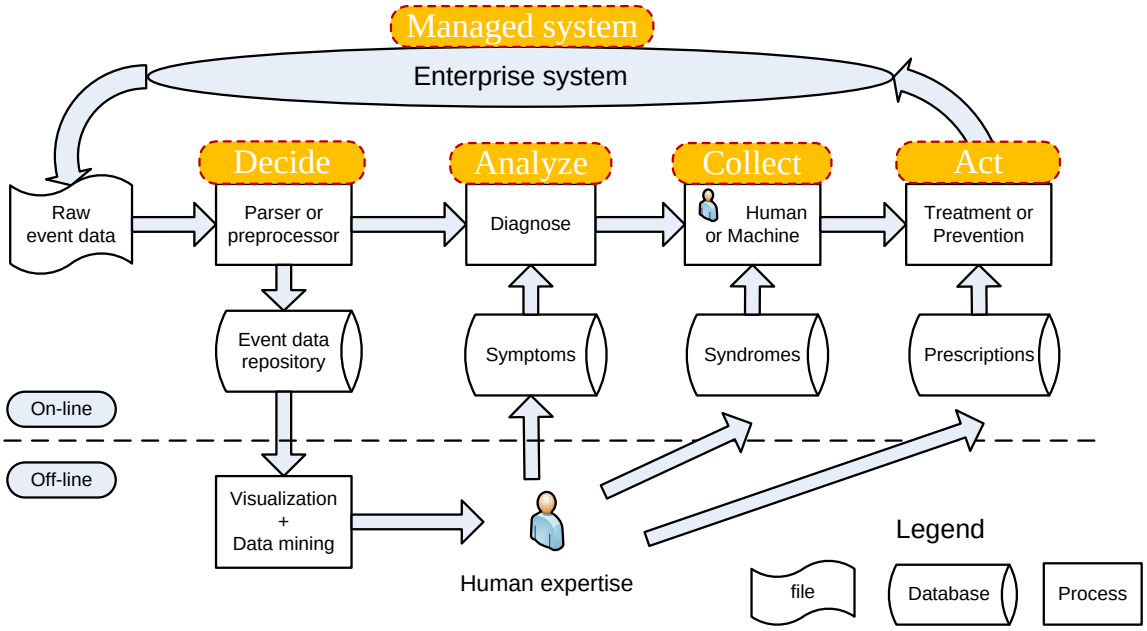


Figure 6-4: An autonomic control loop in conceptual RCAD architecture

The ultimate goal of this research is to make the RCAD system fully autonomic, which means that the Conceptual RCAD Architecture itself forms a control loop. In this loop, the parser or pre-processor "collects" information from traditional network sensors and reporting streams. The diagnose unit "analyzes" this information based on the rules provided by a symptom database. A machine rather than human will further decide if there is a syndrome based on the decisions and hypothesis provided by the syndrome database, after which a corresponding treatment or prescription will be actuated onto the managed system. The integration of the four databases—event-data repository, symptom database, syndrome database, and prescription database—is regarded as the knowledge base of this autonomic system (cf. Figure 6-4).

6.5 Root Case Analysis

Root Cause Analysis (RCA) has historically been used to identify the most basic factors that contribute to an incident [PaBu88][LJP01][LaLa02]. The practice of RCA is predicated on the belief that problems are best solved by attempting to correct or eliminate root causes, as opposed to merely addressing the immediately obvious symptoms. RCA is often considered to be an iterative process, and is frequently viewed as a "tool" of continuous improvement. Typically associated with operational activities, it is often used in *project management*. Hence, the definition of *root cause* by Paradies and Busch [PaBu88] is as follows:

*A **root cause** is the most basic cause that can reasonably be identified and that management has control to fix. Root cause analysis is the task of identifying root causes.*

The three key words in the definition of root causes are *basic*, *reasonably*, and *fix*. On the one hand, root causes must be so basic (or specific) that one can fix them. On the other hand, given that fixing them is the whole point, it is not reasonable to further split root causes into more basic causes. Consequently, these more basic causes are not root causes, and root causes lie at the "highest" level where fixing is possible [Juli03].

Root cause analysis is not a single, sharply-defined methodology; depending on the domain and application, there are many different tools, processes, and philosophies for RCA in existence. Tools that help groups and individuals identify potential root causes of problems are known as *root cause analysis tools* [Dogg05]. The cause-and-effect diagram (CED), the interrelationship diagram (ID), and the current reality tree (CRT) are three root cause analysis tools frequently identified in the literature as viable mechanisms for solving problems and making decisions.

6.5.1 Cause-and-Effect Diagram

The cause-and-effect diagram was designed to sort the potential causes of a problem while organizing the causal relationships. Professor Kaoru Ishikawa developed this tool in 1943 to explain to a group of engineers at Kawasaki Steel Works how various manufacturing factors could be sorted and interrelated. As it came into widespread use for quality control, it became known as the Ishikawa diagram, or more informally, the "fishbone" diagram because of its appearance upon completion (Ishikawa 1982, Arcaro 1997) [Ishi82] [Arca97].

Ishikawa [Ishi82] outlines the following steps for constructing a cause-and-effect diagram:

1. Decide on the problem to improve or control.
2. Write the problem on the right side and draw an arrow from the left to the right side, as shown in Figure 6-5 (A).
3. Write the main factors that may be causing the problem by drawing major branch arrows to the main arrow, as shown in Figure 6-5 (B).
4. For each major branch, detailed causal factors are written as twigs on each major branch of the diagram. On the twigs, still more detailed causal factors are written to make smaller twigs, as shown in Figure 6-5 (C).
5. Ensure all the items that may be causing the problem are included in the diagram.

Major cause category branches can be initially identified using the four Ms: *material, methods, machines,* and *manpower,* or more correctly, the four Ps: *parts* (raw materials), *procedures, plant* (equipment), and *people*. Sometimes *measurement* or *environment* is the fifth category. Arcaro [Arca97] suggests using no more than eight major categories.

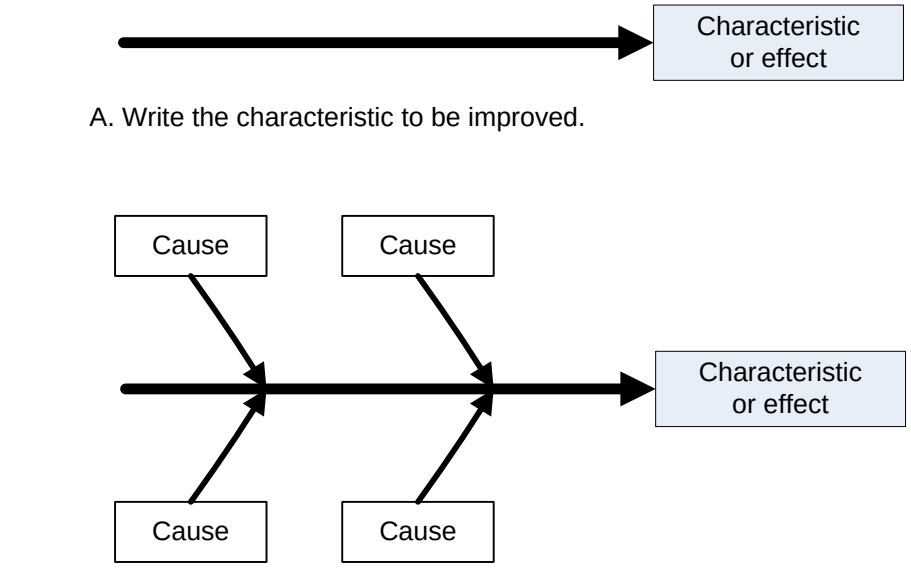


Figure 6-5: Steps in building a cause-and-effect diagram (CED) [Ishi82]

A drawback to using the cause-and-effect diagram is that there is no specific mechanism for identifying a particular root cause once complete. One technique is to look at the diagram for causes that appear repeatedly within or across major categories. Selecting a single root cause, however, may prove difficult unless the characteristics of the problem are well known or documented [Dogg05].

6.5.2 Interrelationship Diagram

Originally known as the *relations diagram*, the *interrelationship diagram* was developed by the Society of Quality Control Technique Development in association with the Union of Japanese Scientists and Engineers in 1976. It was designed to clarify the intertwined causal relationships of a complex problem in order to find an appropriate solution. Brassard and Ritter suggest that the interrelationship diagram uses arrows to show cause-and-effect relationships among a number of potential problem factors. Short sentences or phrases expressing the factor are enclosed in rectangles or ovals [BrRi94]. Arrows drawn between the factors represent a relationship. As a rule, the arrow points from the cause to the effect or from the means to the objective. The arrow, however, may be reversed if it suits the purpose of the analysis [Mizu88].

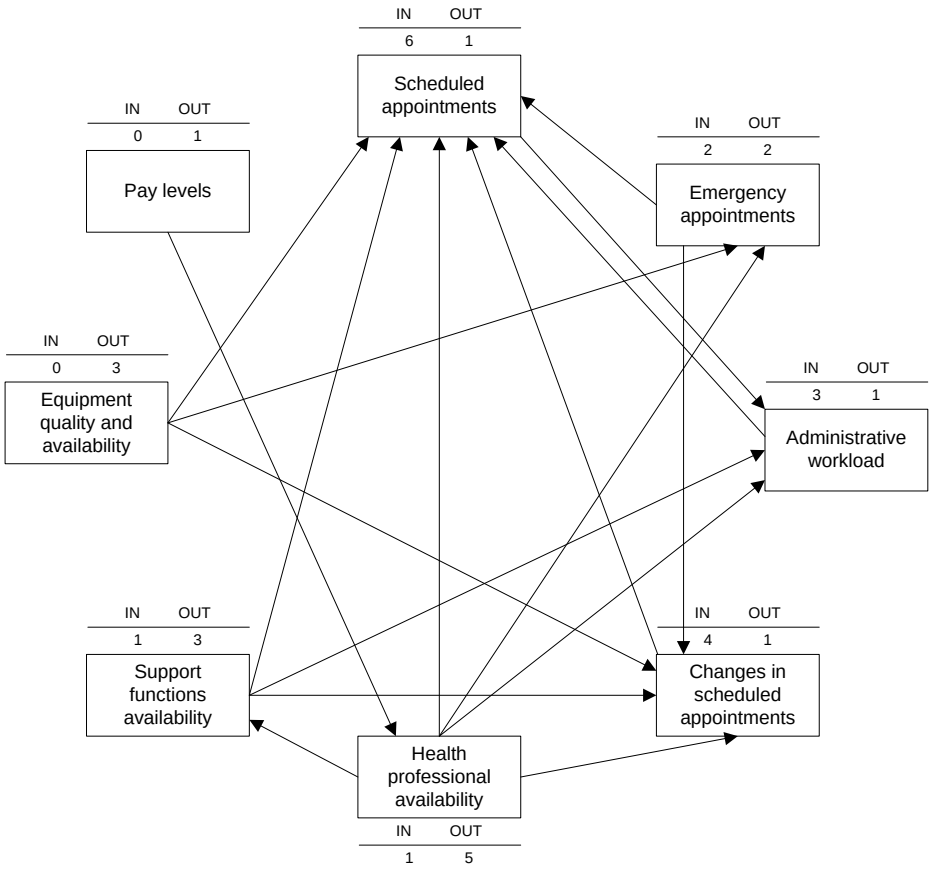


Figure 6-6: Example of an interrelationship diagram (ID) [Dogg05]

Andersen and Fagerhaug write that the first step for using an interrelationship diagram is to determine and label the factors, then place them on an easel or whiteboard in a circular shape and assess the relationship of each factor on other factors using arrows. After all relationships have been assessed, count the number of arrows pointing into or out of each factor. A factor with more "out" arrows than "in" arrows is a cause, while a factor with more "in" arrows than "out" arrows is an effect [AnFa00]. Figure 6-6 shows an example of an interrelationship diagram.

A particular concern of the interrelationship diagram is that it does not have a mechanism for evaluating the integrity of the selected root cause. Some users may simply count the number of arrows and select a root cause without thoroughly analyzing or testing their assumptions about the problem.

6.5.3 Current Reality Tree

In the early 1990's, Goldratt promotes the idea that the factors of problems are interdependent and result from a few core (root) causes [Gold90]. Goldratt expands on the idea of problem solving through *focused intuition* in the book "It's Not Luck" [Goldra94], which introduces the *current reality tree* (CRT). The current reality tree was designed to show the current state of reality as it exists in a system. It addresses problems by relating multiple factors rather than isolated events. Its purpose is to help practitioners find the links between symptomatic factors, called *undesirable effects* (UDEs), of the core problem. Figure 6-7 shows an example of a current reality tree.

The procedure for constructing a current reality tree was first described in the book "It's Not Luck" [Goldra94]. Cox and Spencer [CoSp98] later restate the procedure as follows:

1. List between five and ten problems or UDEs related to the situation.

2. Test each UDE for clarity and search for a causal relationship between any two UDEs.
3. Determine which UDE is the cause and which is the effect.
4. Test the relationship using categories of *legitimate reservation*.
5. Continue the process of connecting the UDEs using "if-then" logic until all the UDEs are connected.
6. Sometimes the cause by itself may not seem to be enough to create the effect. Additional dependent causes can be shown using the "and" connector.
7. Logical relationships can be strengthened using words like some, few, many, frequently, and sometimes.

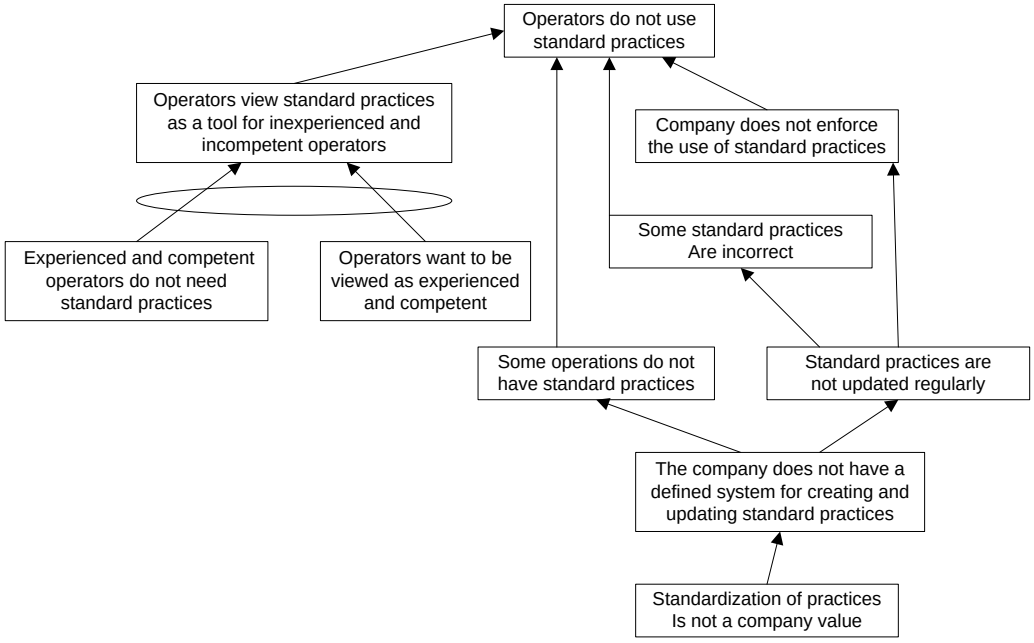


Figure 6-7: Example of a current reality tree (CRT) [Dogg05]

A particular concern of the current reality tree is its complexity of construction and rigorous logic system. Practitioners may find the application of the current reality tree too difficult or time consuming to deal with or to be practical [Dogg05].

6.5.4 Comparison of three root cause analysis tools

Most of the literature describes the three tools independently [Dogg05]. Doggett reviews many related papers and compares the three RCA tools discussed above through a table. Table 19 is a tailored version of this comparison table.

Table 19: Head-to-head comparison of three RCA tools [Dogg05]

Performance criteria	Cause-and-Effect Diagram	Interrelationship Diagram	Current Reality Tree
Ability to find a specific root cause	No	Yes	Yes
Ability to find a reasonable root cause	No	Mixed	Yes
Ability to show systematic causes of effect	No	No	Yes
Shows causal interdependency	No	No	Yes
Identifies factor relationships	No	Yes	Yes
Shows intermediate factors	No	No	Yes
Construction process time	Low	Low	High
Construction accuracy required	High	Medium	Low
Extent of subjective influence on output	High	High	Low
Ease of use	High	High	Low
Overall readability	Low	Low	High

We may notice from the table that although the cause-and-effect diagram is relatively easy to use, it (alone) is lacking in the ability to find a specific/reasonable root cause. Such kinds of drawback could be remedied by computer technology—the cause-and-effect diagram can be built dynamically using a model of relationship between effects and CMDB data (cf. Section 3.1.2).

Summary

This chapter introduced diagnosis-related concepts including symptom, syndrome and prescription in both the medical and computing context. We pointed out the differences between the nomenclatures of these two domains. This is the first contribution of this chapter. We then described a slightly simplified version of a medical scenario provided in the OASIS white paper authored by CA Inc., IBM and Fujitsu researchers [BBDL10] as a non-technical illustration of symptom concepts. We match this example to two iterative circles of the three-step scientific method defined by Zeller [Zell06]. This is the second contribution of this chapter. This chapter also introduces a real world RCAD scenario *Use Case One* as provided by CA. Finally, the Conceptual RCAD Architecture, today's state-of-the-art system management and the foundation of our RCAD research, is illustrated.

Chapter 7 Automating the RCAD Process

To track causalities in layered enterprise systems, cause-and-effect relationships can be represented as fault trees or fishbone diagrams. RCAD processes, for symptoms (effects), can be described by use-case scenarios or workflows. Kontogiannis and Wong proposed a tool prototype, which assists users to follow the workflows by auto-prompted fishbone diagrams in an RCAD process. My vision is to leverage autonomic computing technology, processing each branch of the workflow simultaneously, and making the RCAD process for an individually-investigated effect autonomic. The RCAD process for one effect should reflect a single iteration of the net casting information seeking model.

7.1 Tracking Causality in Layered Enterprise Systems

A key to a successful RCAD process is knowing what caused the symptoms and related events, and having that causal knowledge at the time the events happen [Luck05]. The causal relation between the events is sometimes called *horizontal causality*, which means to emphasize that the causing and caused events happen on the same conceptual level in the overall system. Events can also be components of other events, in which case the component events are thought of as happening at a lower level. The component events of an event also cause that event. This kind of causal relation between events is called *vertical causality* [Luck05].

The diagnostic tasks for IT administrators are often not only tracking horizontal causalities, but also vertical causalities. The information they gathered to carry out an RCAD task is not always from the appropriate abstraction level. If the causes and effects belong to different concept layers, the diagnosis process should reflect the inference steps for tracking vertical causalities up and down the layers.

Tracking vertical causality is not a simple job because there is almost as much dynamism in layers of our enterprise systems as there is in the communication among them across the Internet. For example, the ability of a service provider to load-balance the incoming requests may depend on the loads of low-level services that are continually changing.

There is a general principle of layered enterprise systems. An activity at the top causes activities at successively lower levels, which in turn causes other activities to happen at the top [Luck05]. Activity at each layer is translated into activities at the layers below and vice versa. These lower-level activities must complete successfully in order for the high-level activities to also complete successfully. A successful completion means that all the functional and non-functional requirements of the activities are met. Such relationships between activities at different levels can be represented as the *goal model* of that enterprise system [Zhu08].

7.2 From Goal Models, Goal Trees and Fault Trees to Fishbone Diagrams

In the requirement engineering community, goal models have been used to capture and analyze stakeholder intentions, where functional and non-functional requirements are represented as *hard goals* and *soft goals*, respectively [WMYM07]. In goal models, hard goals represent the stakeholder intentions when there is no ambiguity, such as "day trading". Soft goals represent the stakeholder intentions when the goals are qualitative and there are no clear-cut criteria to judge their fulfillment, such as "performance". A hard goal relates to its lower-level sub-goals through an AND/OR relationship that has obvious semantics—with an AND relationship, a hard goal will be satisfied only if all of its sub-goals are satisfied; with an OR relationship, a hard goal can be satisfied when one of its sub-goals is satisfied. In addition, Chung et al. advocate that hard goals can be related to soft goals through help (+), hurt (-), make (++) or break (--) relationships [CNYM99]. A goal model construction process aims at capturing and analyzing

stakeholders' intentions during requirements engineering. The resulting goal trees represent links between system component/source code and goals/tasks.

A partial goal model that represents the trade-off between performance and cost with respect to utilizing the cache in the Daytrading system (cf. Chapter 11) is depicted on the lefthand side of Figure 7-1 below. If higher performance is preferred to lower cost, the resulting goal tree is depicted in the center of Figure 7-1. Here, *clouds* represent soft goals; *rectangles* represent hard goals, and *ovals* denote *operationalizations* [CNYM99]. In order to not clutter the figure, only *performance* related nodes are drawn in the partial goal model.

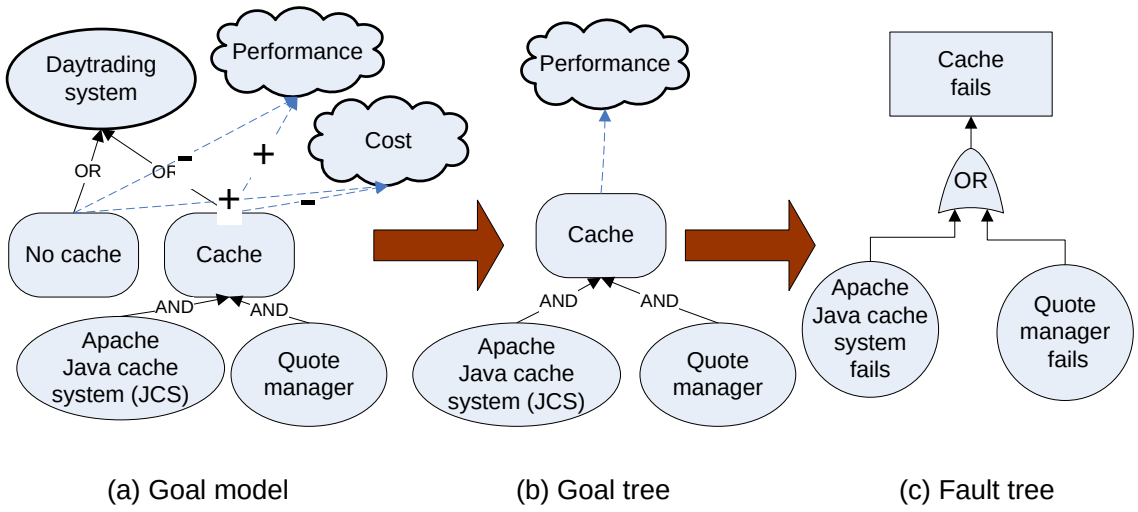


Figure 7-1: Goal model, goal tree and fault tree

One of our key ideas is a guide map—a set of goal trees and fault trees—to aid users in the process of choosing (i.e., supported by high-level goal trees) and applying (i.e., supported by low-level fault trees) suitable diagnostic tools [Zhu08].

A *fault tree* model is a graphical representation of logical relationships among events (i.e., usually failure events). A fault tree consists of the undesired top event (i.e., system or subsystem failure) linked to more basic events by logic gates. The top event is resolved

into its constituent causes, connected by AND, OR and M-out-of-N logic gates, which are then further resolved until basic events are identified. Fault tree technology has been adapted to the fault tree models to handle specific complexities associated with computing systems [Duga00]. Since fault trees have been used to verify (safety) goals and a hierarchy of events in a fault tree is structurally similar to the goal hierarchy of a goal tree, we can derive fault trees from the goal trees of the subject system. The transformation result of the sample goal tree in the center of Figure 7-1 is depicted on the right hand side of Figure 7-1.

However, as McManus pointed out, fault trees typically fail because [McMa10]:

(1) People do not use them in a disciplined manner to develop multiple problem causes at each level (i.e., fault trees are often incomplete);
(2) Multiple levels of potential causes exist to be sorted through for each problem type (i.e., a myriad of fault trees are hard to follow by a human);
(3) They are opinion-driven. They often tend to be a blend of *cause-and-effect diagrams* and a *flow chart*. In such cases, the user can easily get lost and not arrive at any particular root cause.

In reality, fault trees for an enterprise system are often incomplete. Even so, these incomplete fault trees are used in great numbers and most of them have many levels. These trees usually are difficult to display properly on a computer monitor screen, making it hard for administrators to navigate through multiple pages without strong cognitive support [Wale02]. The alternative technique to overcoming these drawbacks is to use computer-aided *cause-and-effect diagrams*.

A typical cause-and-effect diagram depicted in Figure 7-2 has a box called *fish head* at the right hand side, in which is written the effect that is to be examined. The main body of the diagram is a horizontal line from which stems the large *bones*, representing the cause categories that need to be investigated. The large bones are drawn towards the left-hand side of the paper and each labelled with the *generic heading*. Off each of the large bones there may be smaller bones highlighting more specific aspects of a certain cause

category. Each smaller bone is labelled with an *idea*, which represents a potential cause. Sometimes there may be a third level of bones or more.

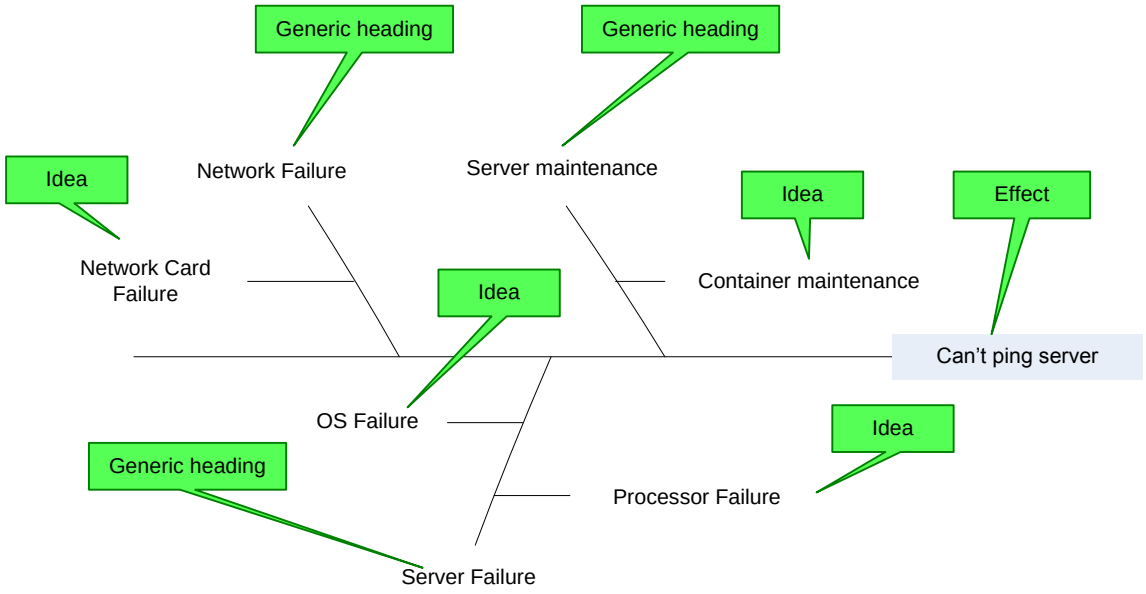


Figure 7-2: A cause-and-effect diagram

With less than four or five levels of branches, a cause-and-effect diagram can be displayed properly in scale on a computer screen as depicted in Figure 7-2 and is easy to navigate manually. However, this advantage in display may bring up a disadvantage in navigation, which means users may encounter the *tunnel vision* problem.

7.3 Tunnel Vision Problem

In medical terms, tunnel vision is the loss of peripheral vision with retention of central vision, resulting in a constricted circular tunnel-like field of vision. In an RCAD process, IT administrators are generally more inclined towards focusing on one point at a time rather than having a broad set of many possibilities in the field of view. In particular, it is difficult for human operators to expand the correlation horizon and collect all the possible cause-and-effect diagrams (i.e., with different fish heads) that share the same cause(s).

The tool prototype built by Kontogiannis and Wong (cf. Figure 7-3) for computer-aided diagnoses is based on the following idea: managing tunnel vision by expanding the correlation horizon using auto-prompted *fishbone diagrams* (although two terms cause-and-effect diagram and fishbone diagram are generally used indistinguishably and interchangeably, in this dissertation we use the term *fishbone diagram* specifically for those cause-and-effect diagrams where fish head on the left hand side and supported by Kontogiannis and Wong's tool).

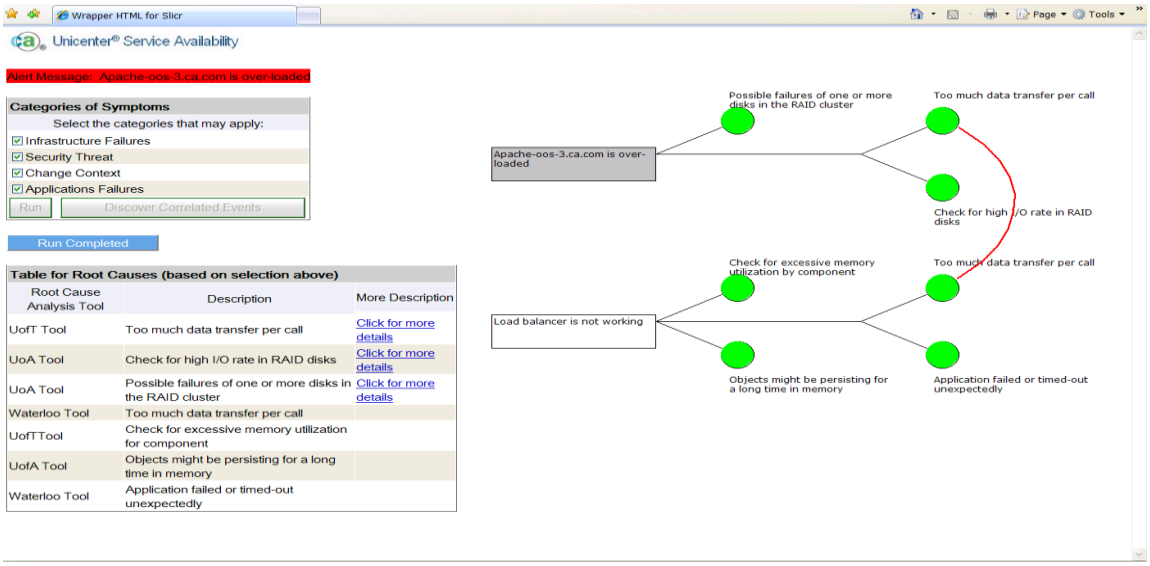


Figure 7-3: Computer aided diagnosis

In this tool prototype, failures are evaluated against the *failure cause-effect model* represented in the fishbone diagram of the situation; the fishbone diagram is built dynamically using a model of relationships between effects and CMDB data (cf. Section 3.1.2). In the process of the investigation, an administrator selects an effect (fish head) for investigation. The system analyzes all known relationships of the selected effect with other effects and builds a list of observed and non-observed effects.

When an IT administrator investigates an effect, he/she will check all the causes according to the fishbone diagram with respect to the effect. Some causes will contribute to other effects; the fishbone diagrams of those effects will be provided by the tool automatically (i.e., auto-prompted). The same cause(s) contributing to the fishbone diagrams of different effects will be linked by red color arches (cf. Figure 7-3). When the administrator observes that several (i.e., more than two) effects share the same cause(s) in the system, the cause(s) is most likely the root cause(s) of the effects. The RCAD process is done semi-manually with the assistance of this diagnostic tool. To interpret this solution better, an abstract example is given in Figure 7-4.

There are three fishbone diagrams (i.e., FD1, FD2 and FD3) in this example. In fishbone diagram FD1, effect E1 (written in the fish head) has three causes: C1, C2 and C3. In fishbone diagram FD2, effect E2 has three causes, and one of them is C2. In Fishbone diagram FD3, effect E3 has four causes. One of them is C3.

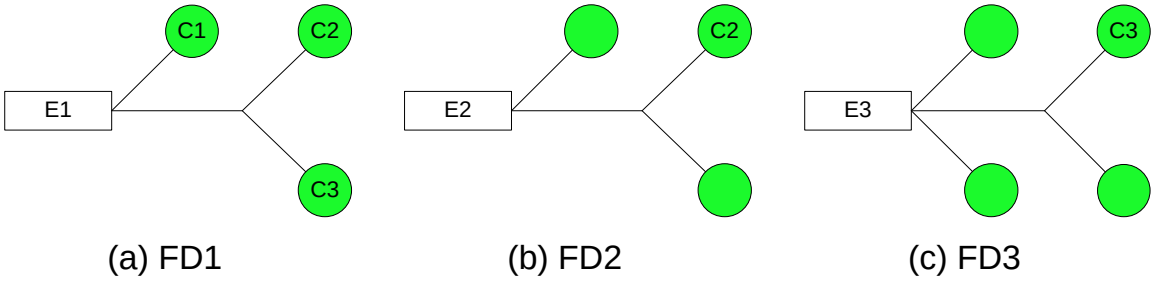


Figure 7-4: Fishbone diagram FD1, FD2 and FD3

When an IT administrator investigates effect E1, FD1 will be displayed on the computer monitor screen, with E1 written in a gray fish head (cf. Figure 7-5). The administrator will check all three causes C1, C2 and C3 one by one according to the fishbone diagram FD1. Some causes will contribute to other effects. In this example, C2 contributes to E2 in fishbone diagram FD2; C3 contributes to E3 in fishbone diagram FD3 respectively (cf. Figure 7-4).

Since cause C1 only contributes to E1, no other fishbone diagrams will be prompted on the screen when the administrator clicks on C1 in the fishbone diagram FD1 to allow for further investigation. In contrast, fishbone diagram FD2 will be automatically displayed when the administrator clicks on C2 in the fishbone diagram FD1 to allow for further investigation (i.e., related information of C2 will also be displayed in the tool). Fishbone diagrams FD1 and FD2 will now be linked through C2 as depicted in Figure 7-5.

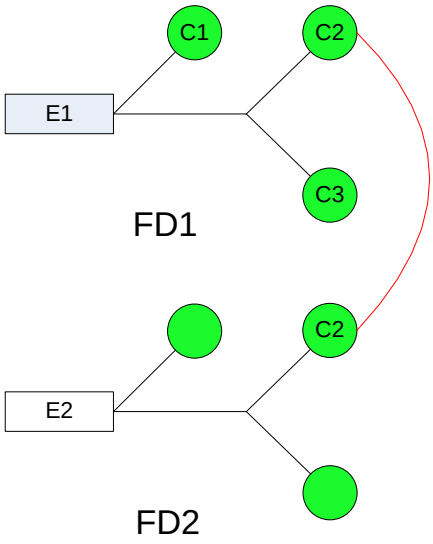


Figure 7-5: Linking fishbone diagrams through common causes (i.e., C2)

Similarly, fishbone diagram FD3 will be automatically displayed when the administrator clicks on cause C3 in fishbone diagram FD1 to allow for further investigation (cf. Figure 7-6).

If effect E1 and E2 are observed by the administrator during the investigation, then cause C2 is most likely the root cause of effect E1, since effect E1 and E2 share the same cause C2 in the system. Similarly, cause C3 is most likely the root cause of effect E1, if effect E1 and E3 are observed. If only effect E1 is observed, then cause C1 is most likely the root cause of effect E1. Of course more complex situation can arise, such as both effects E2 and E3 are observed, which means (1) both causes C2 and C3 are the root

causes, or (2) causes C2 and C3 themselves share a further cause (which would imply that causes C2 and C3 are not really the root-level causes).

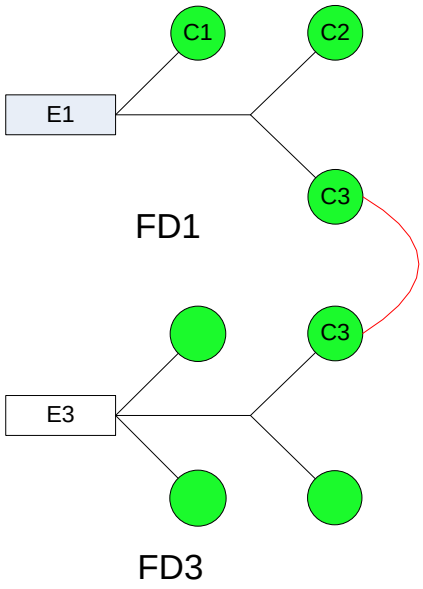


Figure 7-6: Linking fishbone diagrams through common causes (i.e., C3)

7.4 Executing a Workflow Manually

When an RCAD process is carried out manually with the assistance of Kontogiannis and Wong's diagnostic tool, users are only able to focus on one branch of a fishbone diagram at a time since human cognition is both powered and limited by their brains [Wale02]. A berry-picking information seeking model in Figure **7-7** can perfectly illustrate such an RCAD process.

When we take a close look at the berry-picking information-seeking process, we may find that due to the complexity of target systems, an RCAD process is often an inference process which generally follows the *scientific method*.

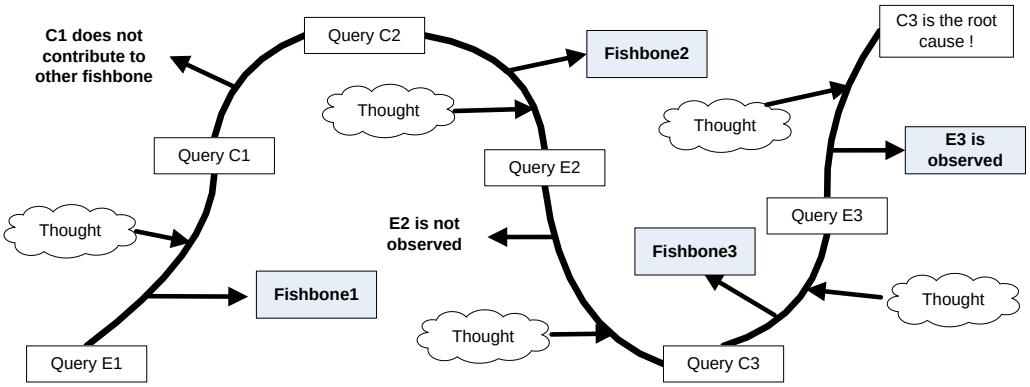


Figure 7-7**: Berry-picking information seeking process for effect E1**

In general terms, the scientific method is a process of obtaining a theory that explains some aspect of the universe [Zell06]. Under the RCAD reigns, it is an appropriate process for obtaining problem diagnostics, which should include these three steps: (1) hypothesis generation, (2) conducting experiments, and (3) observing effects. An RCAD process is an inference process that consists of many rounds of these three-step scientific methods with many iterative/recursive/interactive activities that are conducted by humans upon subject systems. As an inference process, the RCAD process for effect E1 can be represented as a workflow diagram as illustrated in Figure 7-8 below.

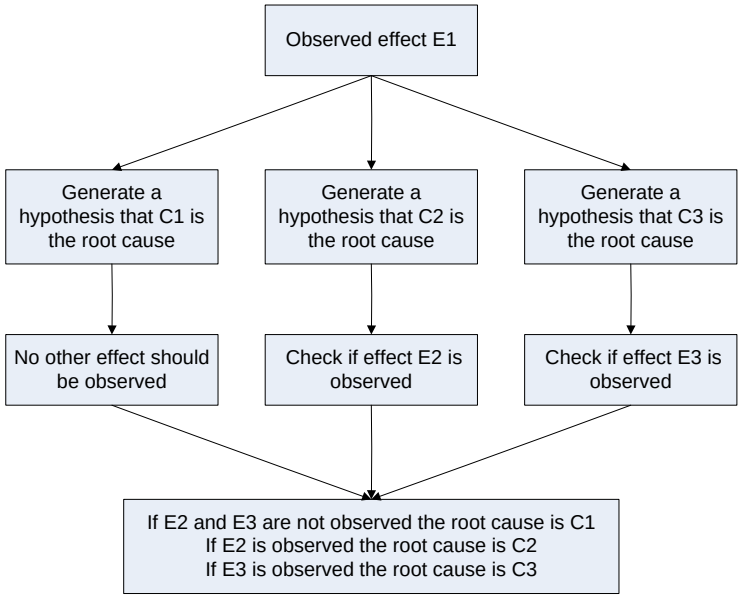


Figure 7-8: Workflow diagram for effect E1

If we carry out this workflow using the berry-picking information-seeking model, the three workflow branches distinguished by three different root causes hypothesis C1, C2 and C3 would be carried out individually. On the other hand, a workflow as such can be potentially processed by machines in parallel. Therefore, a net-casting model can illustrate such an RCAD process.

My vision is: to leverage the autonomic computing technology, processing each branch of the workflow simultaneously, and making the RCAD process for an individual investigated effect autonomic. In particular, the whole workflow described above should be managed by an autonomic element.

7.5 Automating the RCAD Process

Figure 7-9 depicts the concept of an autonomic element as introduced by IBM, which we can utilize to automate the RCAD process at a high abstraction/conceptual level. Messages passing within the element include *symptoms*, *change requests* and *change plans*.

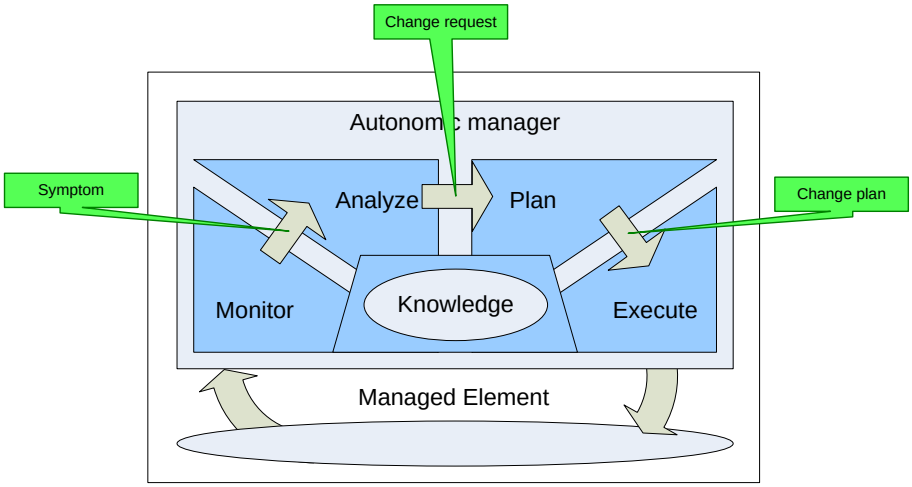


Figure 7-9: Messages within an autonomic element

At the core of each autonomic element is a closed-loop feedback control system. Its controller, also referred to as the autonomic manager, manages the process and the managed elements (could be other autonomic elements). The controller operates in four phases over a knowledge base to assess the current state of the managed elements. The monitor senses the managed process and its context, filters the accumulated sensor data, and stores relevant events in the knowledge base for future reference. The analyzer compares event data against patterns in the knowledge base to diagnose symptoms and stores the symptoms. The planner interprets the symptoms and devises a plan to execute the change in the managed process through the effectors.

It is noteworthy that the autonomic element proposed by IBM is specifically driven by an autonomic manager, which can be designed and implemented with either *polling architecture* or *Event-driven Architecture* (EDA). EDA with Complex Event Processing (CEP) technology in its core is the architecture we prefer to use. The reasons are listed below.

Section 1.2 lists the key problems we need to solve in this RCAD research, including:

- **Massive event clouds.** IT systems are widespread across large enterprises and the different components of enterprise systems generate many events that flow throughout the enterprise system layers.
- **Event flows.** The event-flow of an enterprise IT system becomes non-transparent and difficult to understand.
- **Latency.** When a problem or an opportunity arises it should be noticed right away and in real time to make sure the right action can be taken at the right moment.

All the above issues can be dealt with by our RCAD system. With CEP in its core, it is feasible for an EDA system to act in (near) real time and quickly respond to detected event variances in massive event clouds. Therefore, CEP technology is regarded as essential in the RCAD processes.

7.6 Applying Single Iteration of the Net-casting Model

By leveraging the autonomic computing technology, we are able to make the RCAD process for individual investigated-effect autonomic. Thus, the RCAD process searching for one effect, such as a symptom, can be described by a single iteration of the net-casting information-seeking model (cf. Figure 7-10).

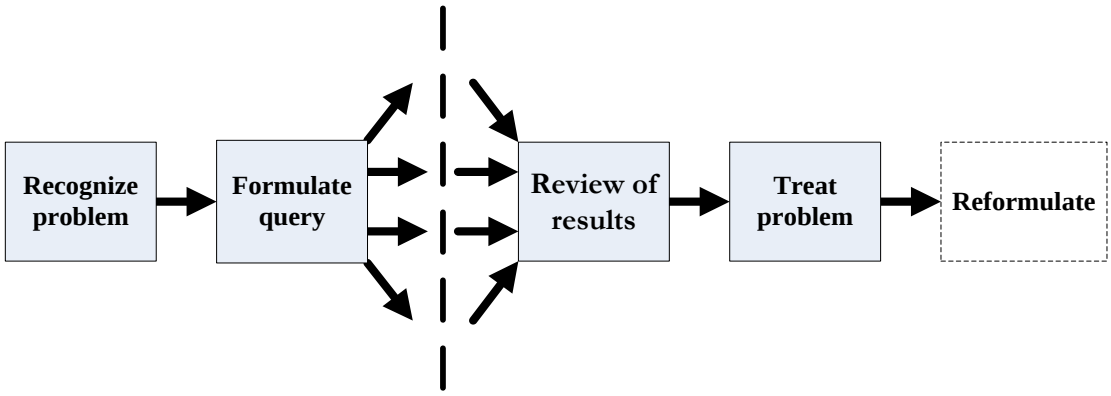


Figure 7-10: A single iteration of the net-casting model

According to the original description in Chapter 5, a net-casting model includes five stages: *recognize problem*, *formulate queries*, *review of results*, *treat problem* and *reformulate*. Since the last stage, *reformulate*, overlaps with the first stage, *recognize problem*, of the next/following iteration, we consider an iteration of a net-casting information-seeking model to include only the first four stages.

The four stages of a single iteration (i.e., *recognize problem*, *formulate queries*, *review of results*, *treat problem*) can be elaborated with *task*-level details provided in Table 17. The diagram of the single iteration is depicted in Figure 7-11. It will become the main building block of our RCAD process.

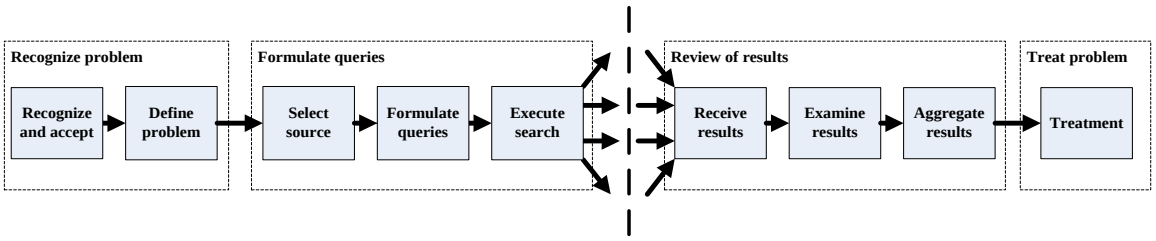


Figure 7-11: "Task" level diagram of a single iteration

Now we can list the *workflow details* of each *task* in Figure 7-11, as Table 20 shown. By following the *workflow details* in this table, potentially we are able to process each branch of the workflow in Figure 7-8 simultaneously. The details of "how" to make such an RCAD process autonomic will be explained in later chapters.

Table 20: Workflow details of diagnose effect E1

Tasks	Workflow details
Recognize and accept	Observed effect E1, aware that there is a symptom and decide to seek the root cause.
Define problem	Formulate hypotheses that C, C2 or C3 could be the root cause.
Select source:	Select where effect E2 and E3 can be observed as the sources of the search.
Formulate queries	Based on information needs, queries will be constructed by EPN
Execute search	Queries are submit within EPN
Receive results	Receive the results of whether effect E2 or E3is observed
Examine results	Either effect E2 or E3 is observed, it is sufficient for diagnosis
Aggregate results	Aggregate candidate diagnoses from different sources and make a conclusion: If E2 and E3 are not observed the root cause is C1; If E2 is observed the root cause is C2; If E3 is observed the root cause is C3.
Treatment	N/A

Summary

This chapter explains the notion of a fishbone diagram, and how auto-prompted fishbone diagrams can aid users to accomplish their RCAD tasks. We recognize that such kinds of RCAD processes can be represented by the Bates' Berry-picking model and the corresponding inference processes can be represented as workflow diagrams. This is the first contribution of the chapter.

Then, I propose that we can leverage the autonomic computing technology, processing each branch of the workflow simultaneously, and making the RCAD process for an individually-investigated effect autonomic. After elaborating the details of a workflow example in this chapter, we can clearly see how the RCAD process can be represented by a single iteration of the net-casting model. This is the second contribution of the chapter.

Chapter 8 Complex Event Processing in an Event Driven System

Event-driven architecture (EDA) is a software architecture style dealing with event streams using complex event processing (CEP) as the event processing model [Luck05]. In an event-driven architecture, the events we receive are not always tailored to the problems we try to solve. Therefore, we connect EPAs together to form an event processing network (EPN) to progress events in stages. In a simple EPN, events that pass through this EPN only flow in one direction and there is no loop within the EPN. To extend the existing CEP paradigm for a new type of EPN for RCAD, we define two groups of EPAs: (1) Monitor EPA: collects data and records some basic statistical information; and (2) Analyze EPA: simulates the inference process of the root cause diagnosis. In such an event-driven process, events not only drive monitor-EPAs, but also analyze-EPAs.

8.1 The Concept of an Event

An *event* is an action or occurrence detected by a software program; it is the fundamental concept of CEP systems. The Information Technology Infrastructure Library (ITIL) [Bon07] defines an event as "any detectable or discernable occurrence that has significance for the management of the IT infrastructure or the delivery of an IT service and evaluation of the impact a deviation might cause to the services." Typical sources of an event include the users (e.g., through a keystroke). Another source is a hardware device (e.g., through a timer).

Schulte clarified the difference between the concepts of a real-world event and a software event [Schu06]:

1. Type 1 event: *An event* [13](*ordinary event* or *real-world event*) is a state change (i.e., something that happened) which is significant to someone for some purpose. This implies that not everything that happens is an event. A state change is an event only if you treat it as one. Many state changes are too fine grained, unimportant or lacking in identity to be events.
2. Type 2 event: *An event* [14](*software event* or *event object*) is an object, usually in the form of a message that reports on an ordinary (Type 1) event.
3. Event processing means computing that uses events (i.e., Type 2, not Type 1).

Typically events (i.e., event objects) get transmitted as event messages, which are defined by an *event model*. An *event model* provides the definition of different types of events with a detailed description of the nature and structure for the events. Each event message has an event header and an event body. The event header normally contains meta-data to be used for quick reference, such as event specification ID, event name, event time stamp, event occurrence number, and event creator. The event body contains the full and detailed description of what has actually happened, for interested parties to interpret. So far, no industry-wide standard exists, although individual companies make the effort to standardise the event message structure, such as IBM's *common base event* (CBE) format [IBM10A].

8.2 Introduction of CEP

[13] Ordinary events (Type 1) may be [Schu06]

(i) Simple or atomic, if they contain no meaningful (Type 1) member events;

(ii) Complex, if they reflect the significance of two or more member events, each of which is meaningful (Type 1 events) in their own right [Schu06].

[14] An event object (Type 2) may report either a simple event or a complex event. A complex-event object (Type 2) may be [Schu06]:

(i) Raw or original, if it directly reports a complex event from the real world or

(ii) Is derived or synthesized, if it is generated by processing two or more Type 2 events.

Before the dawn of complex event processing (CEP), whenever the cause of an event in an enterprise system was to be determined (e.g., in a network layer), we had to search through a large amount of log files, which were usually located in different directories on different machines. There was no causal tracking to tell us what events have caused the happening of another event. Neither there was technology that would help users detect whether events at different locations at different times have a common cause. Nor there was technology that would let users know whether a complex pattern of events has happened globally is repetitive [Luck05].

CEP provides techniques and tools for tackling the above issues by defining and utilizing relationships among events. This technique allows users to define their own events as patterns of the events occurring in the systems and specify the events that are of interest at real time. Different kinds of events can be specified and monitored simultaneously. Many research projects and industrial solutions leverage complex event processing (CEP) and event stream processing (ESP). These approaches address the problem of processing large amounts of events to deliver (near) real-time monitoring, enable control loop decision-making and integrate data continuously. CEP engines, such as AmiT [AdEt04] and Esper [Espe10], have emerged as mature infrastructures for developing sophisticated CEP application systems.

There are several important concepts related to CEP systems:

Event: Each event (i.e., Type 2 events) contains general meta-data (i.e., event ID, time stamp) and event-specific information (e.g., server ID and selected data).

Event pattern: The main task of CEP is *event-pattern matching* (i.e., to identify the patterns of events that are significant for the application domain in an event cloud). In enterprise system management, thousands of raw events must be analyzed to discover event patterns signifying symptoms or other significant states. In some cases, event patterns take context and temporal dependencies between events into account.

Event processing: The first stage of event processing is to recognize relevant patterns of events in the event streams from the sources under monitoring. The second stage is to aggregate information from those events, to build up information that is needed to solve the current issues.

Event processing rule: It consists of two parts: (1) event patterns specify a certain situation of events, and (2) event actions are executed when the event pattern is fulfilled. In particular, new events can be generated within the event action part. Essentially, the event processing rules should transform simple events into more abstract and sophisticated complex events.

Event processing languages (EPL): Event processing rules are expressed by EPLs based on event algebras [ABW03] and processed by a corresponding rule engine. Basically, two different kinds of EPLs can be distinguished: event-condition-actions (ECA) rules [ZiUn99] or SQL-like continuous queries over event streams [ABW03].

Complex event processing (CEP) engine: CEP engine is a type of infrastructure software that provides a language and toolset for programmers to specify the processing of high-speed events, and the runtime server that executes the real-time analysis, either inside a packaged application or an enterprise [Laws10].

Sliding windows: The event processing has to cope with a large number of continuously arriving events from event clouds. For that reason, an appropriate rule language should allow the specification of sliding windows, where only the set of recently occurred events is considered.

Event processing agent: Pattern matching and event processing can be performed by so-called event processing agents (EPAs), which monitor the event streams. EPAs filter, split, aggregate, transform, and enrich events [DrSt05]. Following event processing rules and equipped with rule engines, they synthesize new complex events from simple events. Eventually, the generated complex events will trigger some appropriate event handling.

8.3 Fundamentals of Event-driven Architecture

Event-driven architecture (EDA) is widely applied in the design and implementation of applications or systems where events are transmitted among loosely coupled software components and services. The main focus of this architecture is event processing, if users use CEP as the general process model. Because most enterprise systems that are monitored emit log data continuously, they can be considered event-driven systems (conform to EDA) and are therefore most suited for CEP. Figure 8-1 illustrates the CEP based monitoring for event-driven systems. Event streams flowing from the event clouds, generated by sensors, contain a large volume of different events, which will be transformed, classified, aggregated, and evaluated to initiate suitable actions by the monitoring system. Such a monitoring system can be seen as a primitive form of a diagnostic system.

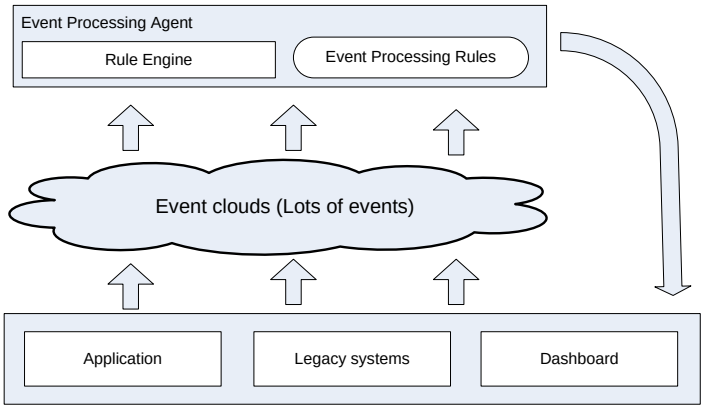


Figure 8-1: CEP based monitoring for event-driven systems

The key characteristic of a CEP system is its capability of handling complex event situations, detecting patterns, creating correlations, aggregating events and making use of time windows. Rozsnyai provides an overview of event model concepts and their implementation in various solutions [RSS07]. Event models have shown that ESP solutions usually treat events in the form of tuples, while CEP solutions make use of more complex data structures. CEP event model and pattern language vary from engine

to engine. They also have some common features. The following Table 21 lists the characteristics of nine popular CEP engines.

Table 21: Characteristics of nine popular CEP engines

Aleri[Aler10]	Aleri Streaming Platform offers two *event processing languages* (EPLs). Aleri SQL adds some extensions to standard SQL and Aleri XML is a set of elements and properties of event processing that can be defined in XML. Continuous Queries can use any of the following elements: • Joins: to correlate and combine data across multiple sources. • Filters: to filter data according to complex criteria. • Data Aggregation and Data Analysis: to summarize and group data sets by computing statistics (e.g. sum, count, average) across like elements. • Compute: to perform mathematical calculations, data transformation, etc., using a full range of operators and built in functions as well as the ability to register proprietary call-out functions.
AmiT [AdEt04]	An event stream engine whose goal is to provide high-performance situation-detection mechanisms. AMIT offers a sophisticated user interface for modelling business situations based on the following four types of entities: events, situations, lifespans, and keys. As a CEP engine, it requires a messaging platform, such as Websphere message broker, to operate upon.
Apama [Prog10]	Apama offers a rich *event processing language* (EPL) optimized for the concise expression of business and temporal logic. Apama's EPL is optimized for the concise expression of business and temporal logic. Available natively and in Java, Apama's EPL delivers a wide range of *complex event processing* (CEP) functionality.
Aptsoft [Apts10]	Aptsoft comes with inbuilt modules to support human interaction and dash board monitoring. It can run on any platform supporting a Java VM and can be clubbed with any standard relational database for persistence. AptSoft UI sets up parameters which the AptSoft CEP server incorporates into a Java application and executes. There is no CEP event processing language per se in the AptSoft platform.
Coral8 (CCL) [Aler10] (Aleri and Coral8	*Continuous computational language* (CCL) is an SQL based programming language. Basic complex event processing constructs, such as filters

merged in 2009)	(SELECT/WHERE), correlations (JOINS) and Aggregators (SUM or AVG), are structured in the same fashion as standard SQL. CCL offers integrated extensions such as windows, time series operations, output control, and pattern matching. CCL also offers numerous procedural language extensions, including: IF expressions, CASE expressions, Loops, Functions, and Variables. Coral8 Engine supports XML data type and CCL provides XML processing functions (similar to SQL/XML) which operate on XML data. Developers can easily extend CCL and the systems CCL can interact with through user-defined functions and plug-ins, and integrating Coral8 Engine into other applications.
Esper [Espe10]	An Open Source event stream processing solution for analyzing event streams. Esper supports conditional triggers on event patterns, event correlations and SQL queries for event streams. It has a lightweight processing engine and is currently available under GNU General Public License (GPL) (GPL v2) License. Esper is completely Java based and its light weight kernel is fully embeddable into any Java Process. It can run on any system which supports Java Virtual Machine. Esper *event processing language* (EPL) has been designed for similarity with the SQL but differs from SQL in its use of views rather than tables. EPL is based on the syntax of SQL but offers extensions for event stream processing, has a comprehensive complex event pattern matching language, and has object-oriented event processing capabilities leveraging the Java type system and XML.
RuleCore [Rule10]	An event-driven rule processing engine supporting *event condition action* (ECA) rules and providing a user interface for rule-building and composite event definitions. The algorithm has its roots in the active database research, such as Snoop [ChMi94].
StreamBase(StreamSQL) [Stre10]	StreamSQL is a next-generation query language for complex event-processing (CEP) applications. StreamSQL extends the industry-standard Structured Query Language (SQL) to empower the processing of real-time data streams. Just as the inherent value of SQL is its ability to issue queries against stored data, this same querying capability must also exist for data streams. The development of StreamSQL was funded by StreamBase Systems.
Tibco [Tibc10]	The TIBCO *business events processing language* (BEPL) is an extendable, feature and grammar rich, object-oriented Java-like EPL that allows users to define business rules on events, systems, services, or a combination of all of these and execute tasks. Built-in EPL functions include: Date and Date

	Functions, Engine.Locale Functions, Engine Functions, Event Functions, Instance.PropertyArray Functions, Instance.PropertyAtom Functions, Instance.StateMachine Functions, Instance Functions, Math Functions, Number Functions, String Functions, String.IO Functions, System Functions, Temporal.Calculus Functions, Temporal.History Functions, Temporal.Numeric Functions, Temporal Statistic Function, and Xpath Functions.

In Table 22 we also compare the EPL approach, the CEP platform and the usability/use type among these CEP engines, based on [AESW08]. Most of these CEP engines are available as commercial products. Among these nine popular CEP engines, Esper is the only one that provides APIs for public access as an open source project.

Table 22: Comparing the EPL Approach, the CEP platform and the usability/use type

EPL approach	CEP-platform	Usability / User Type
Pseudo-SQL	- Aleri, Coral8 (CCL) -StreamBase (StreamSQL) - Esper (EQL)	- skilled EPL programmers - not limited to the community of SQL-programmers
Special rules languages	- AmiT - Tibco	- skilled EPL programmers - unlikely to form a community
Java or other third-generation languages generated code	- Tibco - Apama - Aleri Studio	- community of Java programmers
GUI-based approaches (graphical editors) and code generation	- AptSoft - StreamBase	- Business users, managers, marketing employees - GUI as additional to a EPL

8.4 Simple Event Processing Network

In EDA, the events received are not always tailored to the problems people try to solve. Therefore, we need a technology that to enable progress in stages. For instance, we can connect EPAs to form an event processing network (EPN) [ShEt08]. Figure 8-2 shows the structure of an EPN described by Luckham [Luck05, p. 208]. At the adapter layer, the events are monitored from the IT layer and adapted to the CEP format. The next layer of CEPs is filters that eliminate irrelevant events from further processing. These are

lightweight, fast EPAs typically designed to deal with large event throughput. The filtered events are passed on to a layer of maps and constraints that apply the rules involved in a CEP application (e.g., monitoring event traffic for conformance to business policies).

Luckham point out: A rule engine (i.e., different from a CEP engine) would probably not organize its rules in this way (i.e., filters before maps and constraints). It puts pattern triggers of rules in a tree structure, according to similarities between the patterns. So if an event fails to match one pattern, further attempts to match patterns lower down that branch can be terminated, which reduce the number of matching attempts that cannot succeed. Thus, filtering cannot be applied [Luck05].

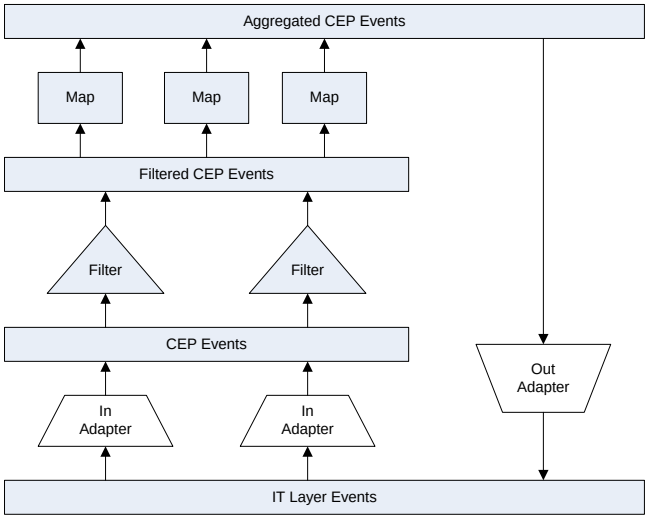


Figure 8-2: A simple EPN Example

Although the EPN, as shown in Figure 8-2, may deal with events that are widely distributed across a large system, and have many monitoring points on the IT layer, the events passing through this EPN only flow in one direction with no loop within the EPN. From a structural point of view, this is a simple EPN.

8.5 A simple EPN is not enough

Although a simple EPN can merge events from various sources to present a coherent global view of system activity, achieving a complete RCAD process for most of the use case scenarios becomes difficult. For example, suppose there is a warning event such as "there are three ping failures from one server within the past 30 seconds". According to the functional requirements of the system, "three ping failures within the past 30 seconds" should be significant enough to be regarded as a symptom related to that particular server. Now the question is, is this symptom significant enough to make a decision that the server has failed and hence reboot it or replace it? Experienced IT administrators would have a negative answer to that question. Instead, they would collect more evidence to obtain a better diagnosis.

To understand the situation better, it is helpful to compare the system diagnosis process with a doctor-patient scenario. There is an analogy between system diagnosis and medical diagnosis. From a medical diagnosis perspective, the doctor will conduct an initial exam or test when a patient walks into a clinic. According to the hypothesis of the potential disease the patient might have, the doctor prescribes a series of tests. When all the needed information is collected, the doctor will diagnose the patient's problem and recommend treatment accordingly. However, the diagnosis and/or treatment may be inadequate or wrong. The doctor will specify recommendations incrementally based on the feedback obtained from the patient, and diagnosis will be performed repetitively until the patient is cured.

From a system-diagnosis perspective, an EPN receives a single or a collection of event streams from event clouds, performs some computation on these events from different layers in a cascaded manner, and may not be able to produce a valid diagnosis. Thus, we need a group of EPAs to perform further data collection.

The main idea is: by extending the existing CEP paradigm, we define Monitor(I)-EPAs to generate hypothesis events according to the use case workflow or fishbone

diagram. Then Analyze(I)-EPAs will trigger Monitor(II)-EPAs to collect more diagnostic data to find more accurate root causes (by Analyze(II)-EPAs). In such an event-driven process, events not only drive monitor-EPAs, but also drive analyze-EPAs. Both groups of EPAs can be driven simultaneously.

These EPAs not only process the incoming event streams and generate diagnoses, but also act the role of autonomic manager at the same time. When different types of EPAs within an EPN are triggered one after another like dominos by a series of events, the inference process of RCAD is automated.

8.6 Events in a Real Use Case

In reality, a use case such as Use Case One will be executed by human administrators. Such types of use cases usually are not directly generated from requirement documentation (i.e., goal tresses). Rather, they are produced by system designers and gradually elaborated by human experts, including experienced system administrators. Often, those use cases are incomplete and might cause users to suffer from the tunnel vision problem.

A use case should record all the necessary steps of an RCAD process for detecting the root cause of a symptom / syndrome. Most incompleteness of a use case could be easily spotted from a use case workflow (a diagram). Thus, drawing a use case workflow for an individual use case is an essential step toward constructing a successful RCAD system. In Chapter 9, I will lay out the detailed steps of constructing the workflow diagram, for example Use Case One—I name it as Use-Case-One Workflow.

Theoretically, a use case workflow (cf. Figure 7-8: Workflow diagram for effect E1) can be represented by a set of fishbone diagrams (cf. Figure 7-4: Fishbone diagram FD1, FD2 and FD3) and vice versa. With a computer-aid RCAD tool such as the one

Kontogiannis and Wong built, we can overcome the tunnel vision problem by expanding the correlation horizon using auto-prompted fishbone diagrams.

To a real world use case like Use-Case-One Workflow, it will be much more efficient and effective to make the RCAD process autonomic. Each use case should be automated with an autonomic element and the corresponding autonomic manager should be event-driven. The event types relevant to Use-Case-One Workflow are depicted in Figure 8-3—the diagram of a single iteration of the net-casting model. The diagram of a single iteration could be tailored. Event types of Use Case One are defined in Chapter 9.

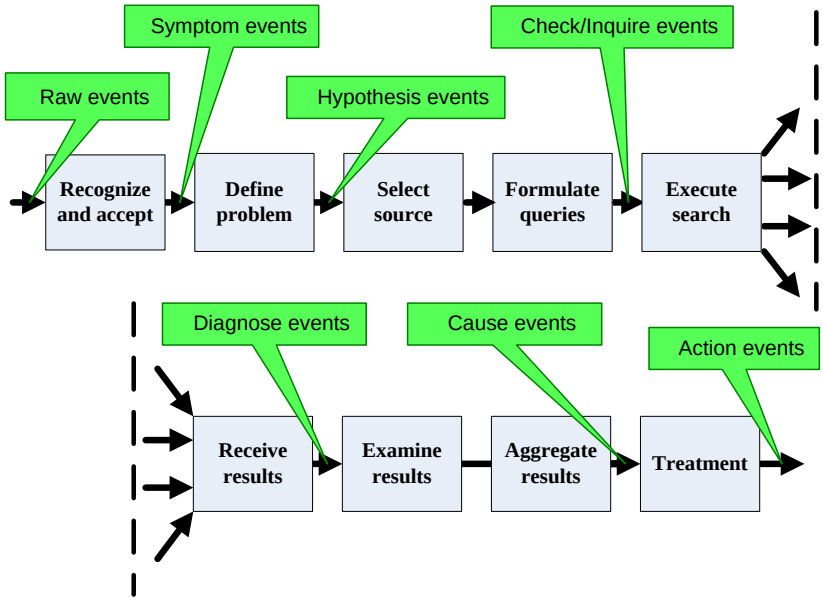


Figure 8-3: Event types of Use Case One in a single iteration of the net-casting model

8.7 An EDA-based RCAD System

Our RCAD system is an EDA-based system. It employs EPNs to track causality with the aforementioned mechanism. Figure 8-4 illustrates the transformation process that includes two different paths (one in yellow and the other in green). By following a single

iteration of the net-casting model described in Section 7.5, an RCAD system transforms raw (simple) events into more abstract and sophisticated synthesized events, such as symptoms. The whole process follows three steps listed below:

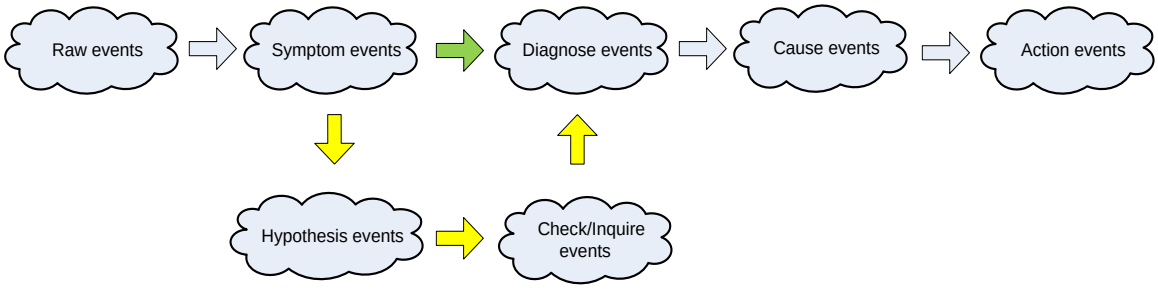


Figure 8-4: Event stream transformation process

(1) A stream of continuously arriving raw events are aggregated and transformed into a synthesized event stream. The raw events contain the information indicating some state changes of a system (i.e., ping-server signal is sent, ping-server response timeout). Most of the synthesized events indicate that the system is normal and that these state changes are desired. Some of the synthesized events are symptoms, which indicate that some change is undesired and there is a problem within the system.

(2) Subsequently, the stream of synthesized events must be analyzed to identify existing problems. This is achieved by applying event patterns on the synthesized event stream that diagnose system problems.

   (a) If a problem detection pattern matches a certain syndrome (i.e., a set of symptoms), a diagnose event is created.

   (b) If no perfect match exists, a hypothesis event is created based on the symptoms and more information is needed to specify the cause. Subsequently, a set of inquire events will be sent to different information resources and results are carried back by diagnose events.

(3) Once there is enough information (provided by diagnose events) to match a cause-detection pattern, a corresponding cause event is created as well. When a problem is identified, an action plan is created, which yields a sequence of appropriate

actions. An example of such an action is a network reroute package to bypass a malfunction router. In the following section we derive a more detailed event model for the RCAD system.

8.8 Event Hierarchies for RCAD

Event is the key concept of EDA as well as our RCAD system. Therefore, events processed by the RCAD system should be defined precisely by a formal event model. A formal event model provides a complete understanding of the different event types, its properties, constraints, and dependencies. Explicit event semantics is essential to an EDA. It is the indispensable basis to derive the software architecture of an EDA-based system.

The key issue of an event model is the layered hierarchy of all event types that are significant for the application domain. In the RCAD system, all events can be structured in a four-layered hierarchy [Dunk09].

1. Raw events are at the bottom of the event hierarchies. They are processed by the sensors, and aggregated and correlated to more meaningful events, called synthesized events.
2. Synthesized events are on the second layer. Some of them are symptom events.
3. If a symptom event appears in the synthesized event stream, the RCAD process will start. The RCAD system will generate a series of RCAD events (including hypothesis events, check/inquire events, diagnose events and cause events), which are on the third layer.
4. The outcome of the RCAD process will be a cause event, which indicates the root cause of the symptom. Finally, according to the “cause” of the symptom, appropriate action events will be sent to the managed system to remedy the situation. Action events are on the top level.

The hierarchy of events for RCAD in Figure 8-5 reflects the sequence of event processing steps.

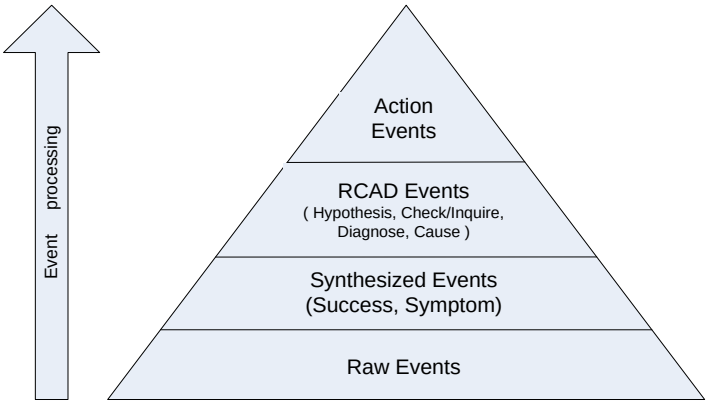


Figure 8-5: Event hierarchy for RCAD

Figure 8-6 shows the relevant event types of our RCAD system, which split/elaborate the event types in Figure 8-5 (the event hierarchy) into appropriate subtypes. The description of event types and subtypes are as follows:

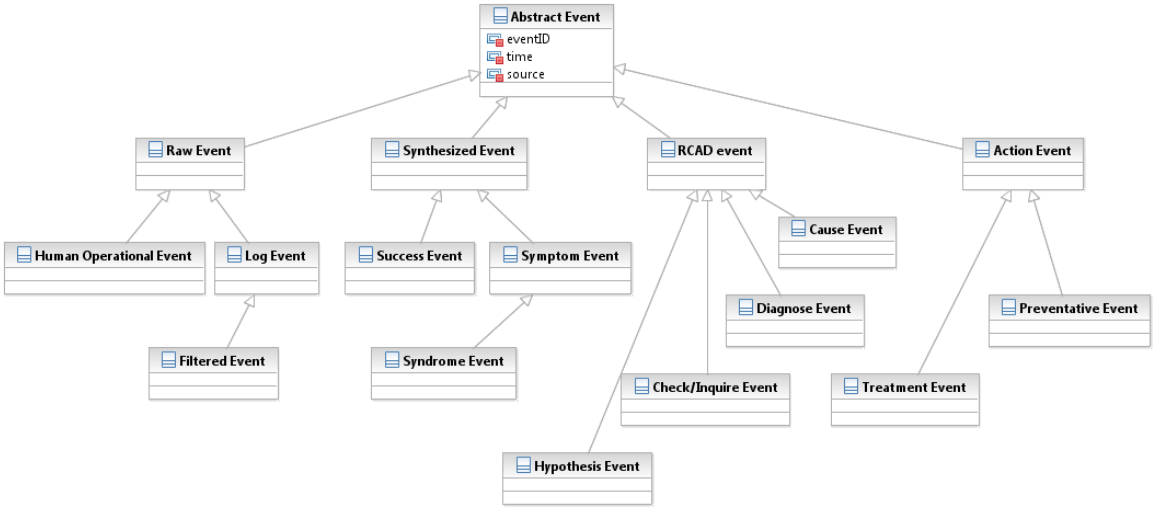


Figure 8-6: Fine grained event hierarchy for RCAD

Abstract event is an abstract event type (represented as an abstract class in Java) that contains meta-data, which is shared (inherited) by all other event types. Typically, it contains an event Id, the occurrence time and the event source. All other event types inherited from this class make use of these properties.

Raw events are the events directly produced by the different levels of sensors (event generators) incorporated in the system. These events contain the raw and unmodified sensor data. We can distinguish them with the following subtypes:

- **Human-operational event** is the event indicating a certain type of human operation acting upon the system.
- **Log event** is the general class of sensor events emitted from a sensor when it detects a state change that needs to be reported and recorded, such as a threshold violation. Depending on the sensor type, the events contain different data.
- **Filtered event** can be considered as the stream of log events reduced by all unnecessary events. To accelerate event processing, the high volume of log events should be reduced as early as possible. Therefore, all irrelevant events must be filtered. For instance, some sensors emit events periodically, such as a timer; others might emit some duplicated events reporting the same real-world event.

Synthesized event: A single raw event is not sufficient for deriving a meaningful state of the actual situation. In general, raw data is uncorrelated and too fine grained, and therefore must be aggregated and correlated.

- **Success event:** is generated and emitted when raw events are analyzed and a desired situation of the system is concluded. For example, a ping server signal is sent, and a corresponding response is received within a time limit.
- **Symptom event:** is created and emitted in case an undesirable situation occurs, while raw events are analyzed. Two categories of symptoms exist: one is threshold violation; another is abrupt change.
- **Syndrome event**: a systems administrator is often not interested in individual symptom readings in time or the location of individual symptom occurrence in the

system, but rather in correlating symptom events based on temporal and spatial observations. A temporal correlation aggregates the symptom at a certain location over a period of time for recording the duration of the problem; for example, "more and more clients finding responses from a particular server are getting slower." Spatial correlations combine the symptoms from different locations at a certain instant of time; for example, "a wide range of servers are out of service."

RCAD events: are generated and processed by EPAs of our RCAD system. RCAD events can be classified into four subtypes: hypothesis events, check/inquire events, diagnose events and cause events. These subtypes themselves include a list of bottom-level subtypes respectively when they are applied to a specific use case. The definition of these bottom-level subtypes is described in Section 9.3.

Action events: are also generated by an EPA of our RCAD system. They can be classified into two subtypes: treatment events and preventative events. Action events are sent to the monitored system to remedy certain situations. The definition of these subtypes of action events is covered in Section 9.3 as well.

8.9 EPAs within the EPN

The hierarchy of events reflects the whole RCAD process, which consists of a sequence of event processing steps. The system transforms the raw events into more abstract synthesized events, which are used to reflect the actual status of monitored system and to identify emerging problems described by symptom events. The symptom events are processed by corresponding event processing agents (EPA), which are connected to an event processing network (EPN). Each EPA consists of a rule engine with a set of event processing rules. The distribution of the processing rules on different agents has two advantages:

- Each EPA has a definitive task, with a relative small number of rules to achieve high cohesion.
- EPAs can be distributed in different locations to achieve a scalable and flexible architecture.

The event processing network connecting the event processing agents is shown in Figure 8-7. It is a substantial extension from the simple EPN example as illustrated in Figure 8-2.

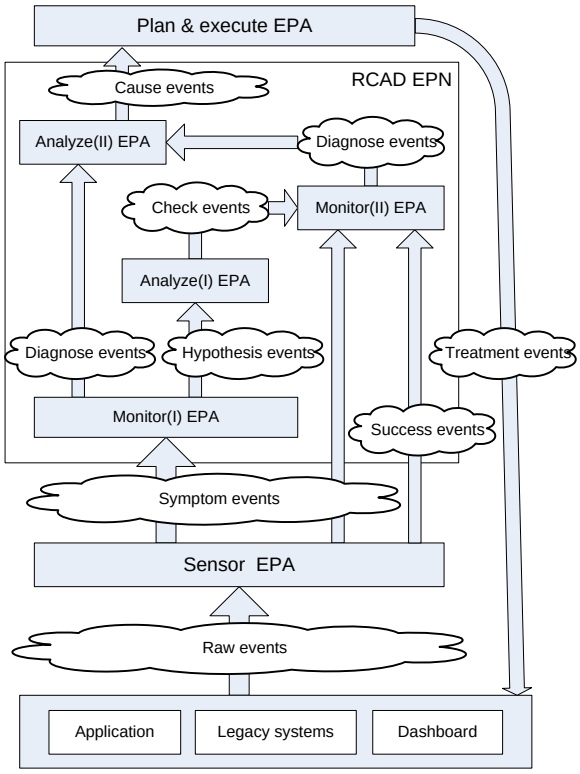


Figure 8-7: EPN for RCAD

Sensor EPA: They carries out the *recognize and accept* tasks in the net-casting model; once the system aware there is a symptom, it will start to seek the root cause of the symptom. It correlates the fine-grained raw events, and emits a stream of synthesized events that may contain success events and symptom events (i.e., when problems are detected). This process also maps numerical values to meaningful qualitative statements.

Numerical measures, (e.g., time period of 15 seconds), have no significance if not related to the characteristics of a monitored system. Instead, our RCAD system uses symbolic rather than numerical values to represent symptoms, (e.g., time out).

The **RCAD EPN** contains four types of EPAs:

- **Monitor(I)-EPA**: If a set of symptom events correlates with a certain type of syndrome on the basis of spatial or temporal characteristics, a corresponding diagnose event is created. If there is not a perfect match, there will be a list of potential root causes. Therefore, one or several hypothesis events will be created based on the hypothesis. This EPA carries out the *define problem* task in the net-casting model (cf. Figure 7-11 and Figure 8-3).
- **Analyze(I)-EPA**: According to the hypothesis, a set of check/inquire events will be sent to different information resources. This EPA carries out the *select source* and the *formulate queries* tasks in the net-casting model (cf. Figure 7-11 and Figure 8-3).
- **Monitor(II)-EPA**: Distributed at different locations, these EPAs will collect needed information according to the check events they receive. The resulting information will be processed and sent back through diagnose events. This EPA carries out the *execute search* and *receive results* tasks in the net-casting model (cf. Figure 7-11 and Figure 8-3).
- **Analyze(II)-EPA**: Once there is enough diagnose events to match a cause detection pattern, a corresponding cause event is created. This EPA carries out the *exam results* and *aggregate result* tasks in the net-casting model (cf. Figure 7-11 and Figure 8-3).

Plan and execute EPA: When a cause is identified, an action plan is created, which yields a sequence of appropriate treatment events or preventative events. This EPA carries out the *treatment* task in the net-casting model.

Summary

In this chapter we surveyed popular CEP engines and provided a brief comparison of the EPL Approach, the CEP platform and the usability/use types among these engines. We then discussed the concepts of event and event-driven architecture. After examining the structure of a simple EPN, we recognized that a simple EPN is not capable of achieving our RCAD tasks such as Use Case One. We then illustrated how event types of Use Case One are defined in a single iteration of the net-casting model. This is the first contribution of this chapter.

With the fine grained event hierarchy for RCAD, we proposed a new type of EPN for RCAD by extending the existing CEP paradigm. We defined Monitor(I)-EPAs to generate hypothesis events according to a use-case workflow/fishbone diagram. Then Analyze(I)-EPAs trigger Monitor(II)-EPAs to collect more diagnostic data and eventually those data enable Analyze(II)-EPAs to discover the root cause. In such an event-driven process, events not only drive monitor-EPAs, but also analyze-EPAs. This is the second contribution of this chapter.

Chapter 9 Designing an EPN for a Use-Case-Unit

Luckham describes a complete lifecycle for designing event-driven systems [Luck05]. It is based on the concept of *process architecture*. In this chapter we show how to apply this lifecycle for the design of our event processing networks (EPNs). A systematic scenario that how to design/construct a use-case-unit EPN is introduced through three real use cases:

1. Use Case One: Cannot ping server(s)
2. Use Case Two: Super Bowl Sunday (cf. Appendices B)
3. Use Case Three: SystemX is perceived to be slow (cf. Appendices C)

9.1 Architectural Diagram of an Event-driven System

Process architecture is a precise, high-level specification of the behaviour of each process in the system and the communication between the processes in the system [Luck05]. It is used as a standard to guide and constrains the development of a process system, throughout its lifecycle—from design to implementation, deployment, and modification. Process architecture has three components:

4. A graphical constituent, called the architecture diagram
5. A behavioural constituent, called the behavioural specification
6. A constraint constituent, called the design constraints

An architectural diagram as in Figure 9-1 graphically shows the processes in the system and the connectors between the processes. The diagram depicts the components of an event-driven system and their communication structure. A behavioural specification augments the diagram with a set of precise rules specifying how each process and connector behaves—that is, how a process or a connector reacts when it receives events by creating new events and data. The design constraints augment the diagram with a set of constraints that specify limitations on the process behaviours and on the

communication via connectors in the architecture. A constraint can, for example, define a time bound on a process activity [Luck05].

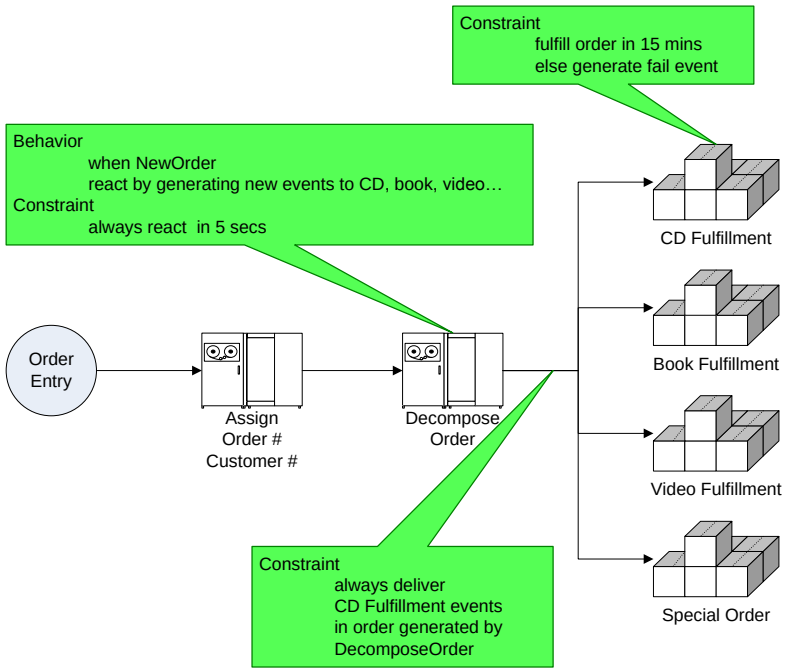


Figure 9-1: An architecture diagram with informal annotations [Luck05]

Behaviours and constraints are called annotations, which are usually expressed in a mathematical or computer language. In our RCAD system, all processes are implemented as *event processing agents* (EPAs). For each specific use case, all EPAs within an *event processing network* (EPN) can be described either in English as informal annotations or in *event pattern language* (EPL) as formal annotation (e.g., for the intrusion detection system introduced in Chapter 11). The process connectors can be implemented by any middleware which provides point-to-point message service, such as Java Message Service (JMS). The delivery mechanism of the message service is not a major concern of this research (i.e., events will not be altered when they are delivered by the message service) and thus the details related to connectors will not be covered in this dissertation.

When constructing an architectural diagram of an event-driven system whose operation depends upon receiving and sending events, two entities within the system must be defined clearly:

- Events: include event types, parameters and descriptions, and
- EPAs: include input event types, output event types and the descriptions of their behaviours.

Figure 9-2 depicts the interface of an event processing agent class. An EPA consists of event pattern rules and local variables whose values make up its state. Each rule has two parts: a pattern called the trigger and a body consisting of actions [Luck05]. In Esper for example, an EPA can be realized by an Esper engine instance.

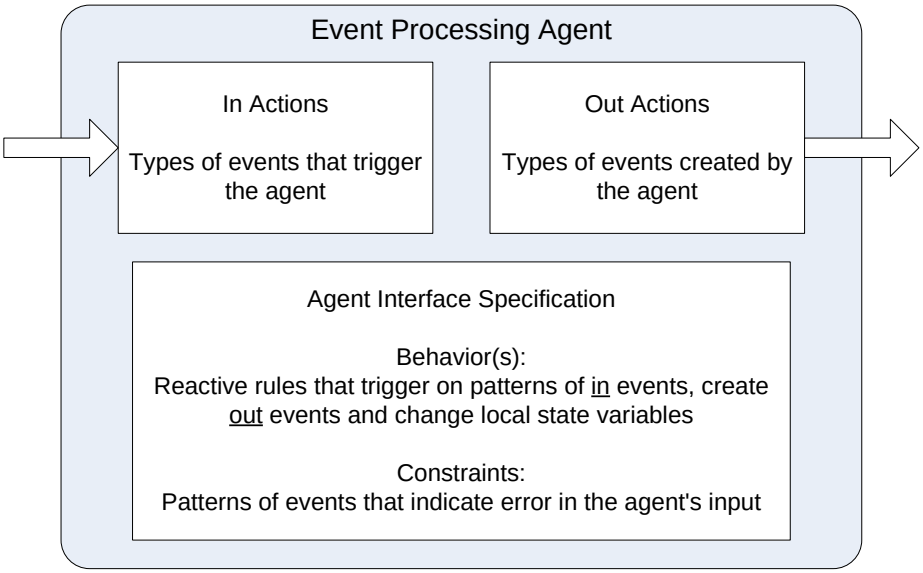


Figure 9-2: Interface of an event processing agent class [Luck05]

The concepts of EPA, EPN, and architecture diagram as described by Luckham [Luck05] can be leveraged in the process of designing an EPN for automating a specific use case.

To design an EPN for automating a specific use case such as Use Case One, we need to further refine those event subtypes of RCAD events (cf. Figure 8-6): *symptom*, *hypothesis*, *check/inquire*, *diagnosis/result*, *cause*, and *treatment*. The refined event subtypes are then defined as input event types and output event types of a set of EPAs. Those EPAs, which form an EPN for Use Case One, perform the tasks as outlined below:

- **Monitor(I)-EPA** generates hypothesis events according to symptom and use case scenarios
- **Analyze(I)-EPA** sends a set of check/inquire events to different information resources according to the hypothesis
- **Monitor(II)-EPA** triggers to collect more data needed to find a root cause
- **Analyze(II)-EPA** when there are enough diagnose events to match a cause detection pattern, a diagnosis is rendered.

The EPN that consists of the above EPAs is called a use-case-unit. In such an EPN, events not only drive monitor-EPAs, but also drive analyze-EPAs. All groups of EPAs can be run simultaneously. In this context, specific event types and individual EPAs based on Use Case One are defined.

9.2 Event Subtypes in Use Case One Workflow

Our EPN (a use-case-unit) is designed based on CEP technology and real use case workflows. A real use case provided by CA called Use Case One has been introduced in Figure 6-2. In fact, this is one of the three use cases (i.e., Use Case One, Two and Three) that are studied extensively in this dissertation. Due to space limitations, only Use Case One is presented in detail in this chapter. The other two are in Appendices B and C. The workflow of Use Case One is shown below in Figure 9-3 and Figure 9-4.

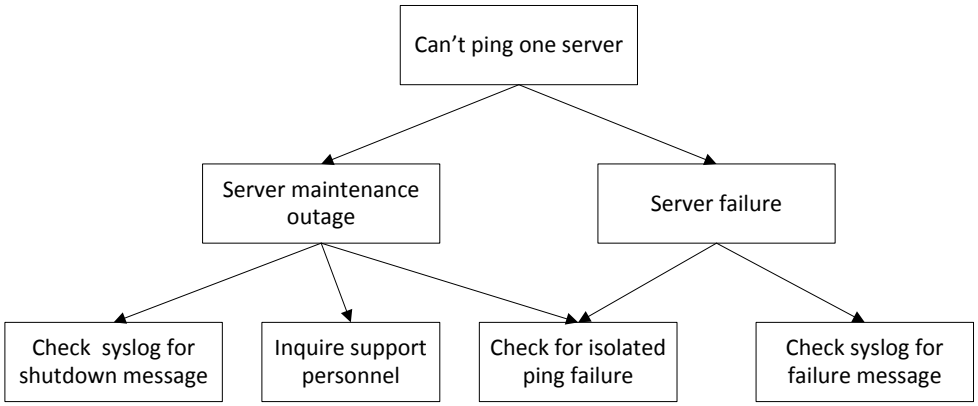


Figure 9-3: Workflow of Use Case One (Part 1)

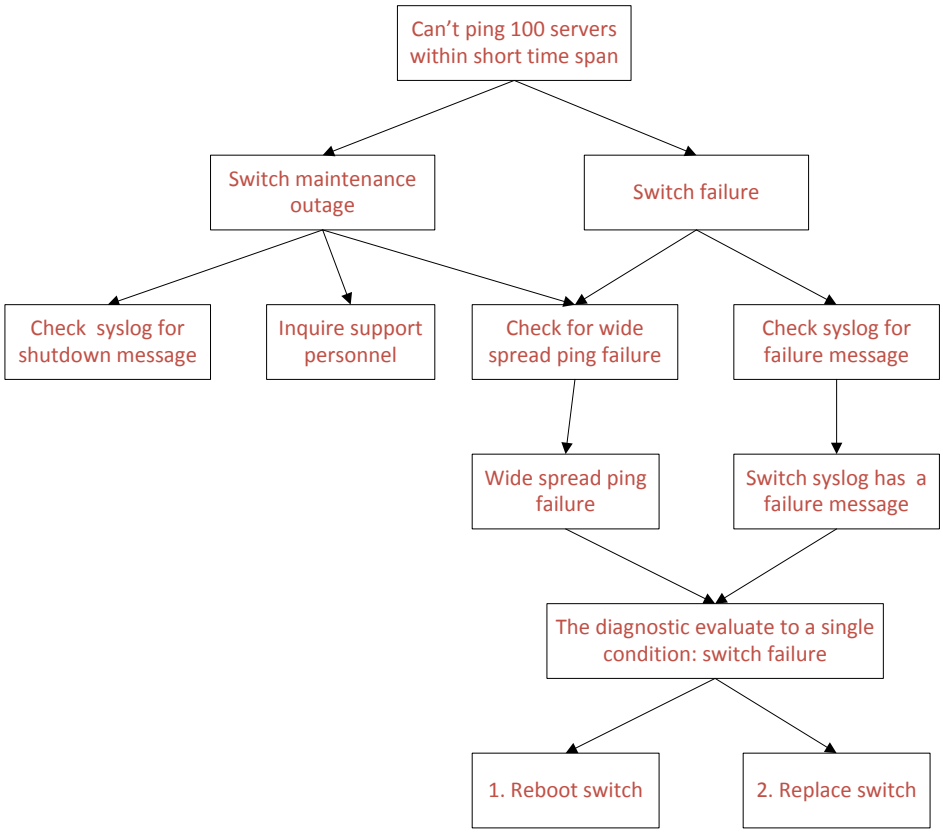


Figure 9-4: Workflow of Use Case One (Part 2)

In the above workflow, when a network administrator finds that he/she *can't ping one server*, a standard diagnosis and treatment process is followed. If there is only one server

that the administrator is unable to ping, then the potential root cause could be a *server maintenance outage* or *a server failure* (cf. Figure 9-3, black text in Figure 6-2). If the administrator *can't ping 100 servers within a short time span*, the possible root cause could be a *switch maintenance outage* or a *switch failure* (cf. Figure 9-4, brown text in Figure 6-2). Several servers failing at the same time is considered a highly unlikely (i.e., small probability) situation.

Therefore, the investigation process could lead to two separate investigation paths: one (cf. black text in Figure 6-2) being the investigation of the server along the workflow in Figure 9-3. The other (cf. brown text in Figure 6-2) is the investigation of the switch along the workflow in Figure 9-4. In either case, more data/evidence are required for diagnosis.

In Figure 9-3, if a network administrator discovers an isolated ping failure and there is a *shutdown* message in the *syslog*, the result of the diagnosis would be a *server maintenance outage*. The network administrator can also ask support personnel to find the real cause (about the server maintenance outage). If the administrator finds that the symptom is an *isolated ping failure* and there is a *failure* message in the *syslog*, the result of the diagnosis would be *server failure*.

In Figure 9-4, if the network administrator identifies a widespread ping failure and there is a shutdown message in the *syslog*, the diagnosis would be *switch maintenance outage*. The network administrator can also ask support personnel to find the actual defect (about the switch maintenance outage). If the administrator finds it is a *widespread ping failure* and there is a *failure* message in the *syslog*, the diagnosis would be *switch failure*. In this case the recommended treatments would be: *reboot switch* or *replace switch*.

Each box in the workflow diagram is considered as an activity and each activity represents a corresponding event. These events are categorized by the researcher and so far six event types have been identified through close examination of the three use cases

provided by CA Inc. Every event in the use cases can be classified into the following six types:

- **Symptom**: one or many alarms or alerts or warnings generated by some anomaly detector mechanism.
- **Hypothesis**: a suggested explanation for an observable symptom or of a reasoned proposal predicting a possible causal correlation among multiple symptoms
- **Check or Inquire**: request for more supportive information from machines or human for further investigation.
- **Diagnosis or Result**: identify the nature of the symptom, by process of elimination or other analytical methods. More correlated symptoms are gathered from machines or humans. The result would be a set of symptoms.
- **Cause**: the set of symptoms defines a syndrome, which in further defines cause(s). This cause(s) is the root cause(s) of the symptoms.
- **Treatment**: solution for the problem

Thus, the boxes/nodes of Use Case One workflow diagram can be clustered/classified by the six event types as stated above (cf. Figure 9-5).

Other noteworthy discoveries when we represent Use Case One into workflow diagrams include:

- Some parts of the workflow are incomplete. For example, the *diagnose*, *cause* and *treatment* steps are missing in *can't ping server* symptom (cf. black text in Use Case One).
- In the red text part of the Use Case One workflow, we found three steps—*check /inquire*, *diagnose*, and *cause*—match the three steps of a scientific method process—*hypothesis*, *experiment* and *observation*.

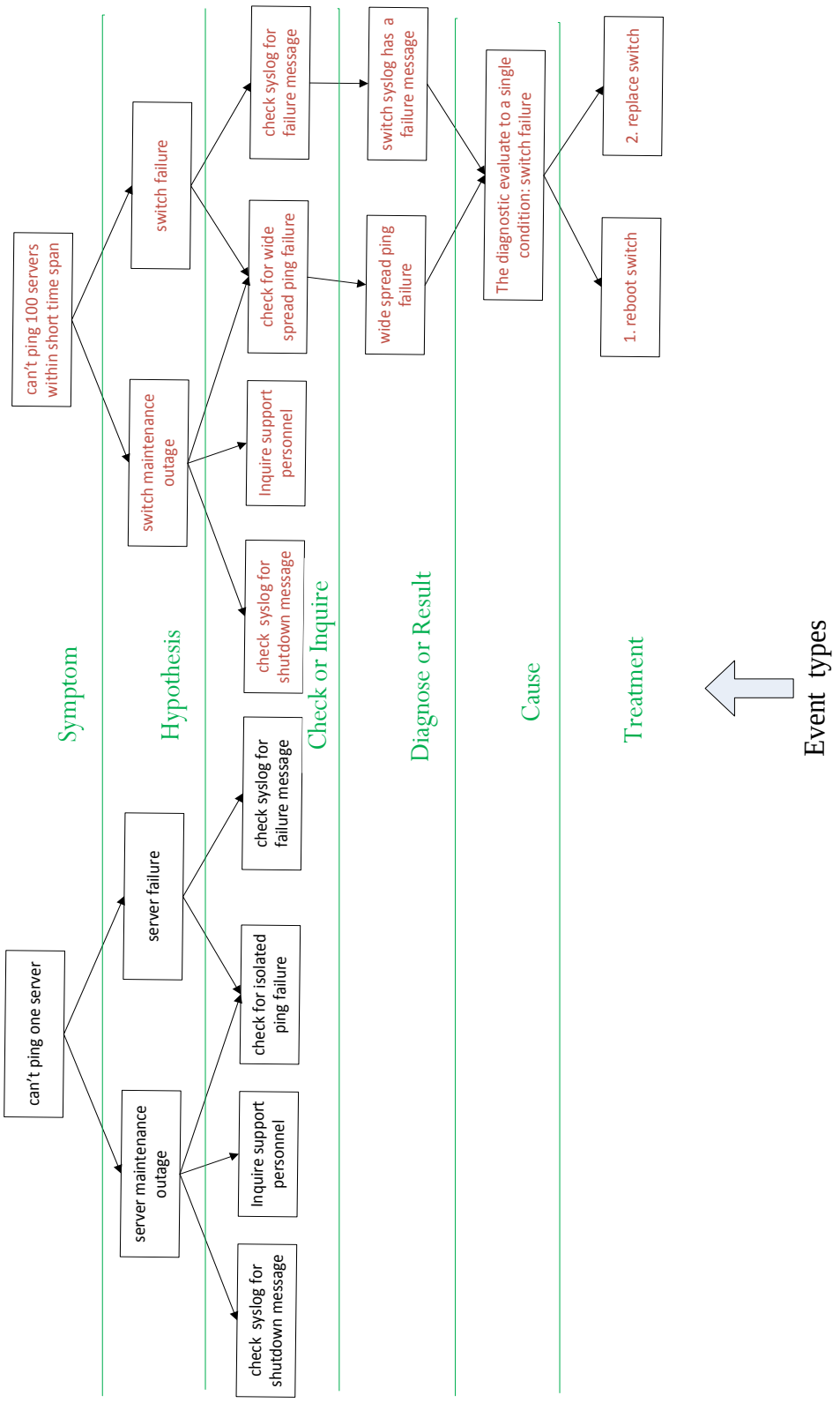


Figure 9-5: The six event types of Use Case One

From other use cases such as Use Case Two and Use Case Three (cf. Appendices B and C, as provided by CA Inc.), we discovered two useful facts:

- Use cases might not be documented in the workflow order.
- Since the scientific method is often unconsciously followed when those use cases have been documented, in some case, events of hypothesis type and cause type share the exact same name, which can cause some degree of confusion. We should distinguish them based on the fact that if they are verified. For example, *virus* would be a hypothesis before it was investigated and confirmed. Thus, hypothesis events should not be defined with a simple event type called *virus*. It would be better to start the event type with the prefix *Event_hypothesis_virus*.

9.3 A Use-Case-Unit EPN

When we consider every activity (i.e., labelled in boxes in Figure 9-5) in the workflow diagram as an event, the process between two consecutive events would be reacting to one type of event and generate another type of event. From an event processing point of view, it would be transforming one type of event into another, which can be accomplished by CEP technology, specifically using EPAs. Therefore, all six types of events that are introduced above can be processed by a collection of EPAs. For example, *symptom* is the input event type of a Monitor(I)-EPA and *hypothesis* is the corresponding output event type. Architecture diagram of a parallel, asynchronous diagnosis process (especially in Monitor(II)-EPAs) is shown in Figure 9-6, which consists of a set of EPAs. This set of EPAs forms an EPN which serves as the analysis engine of a use-case-unit.

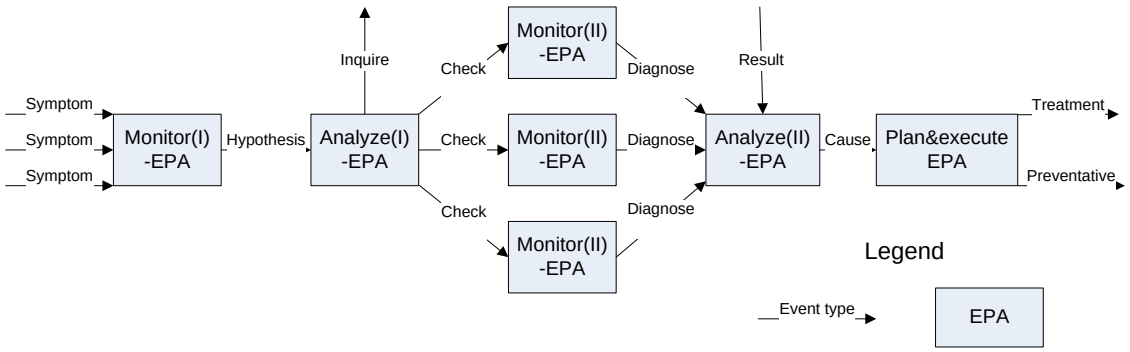


Figure 9-6: A generic use-case-unit EPN

We can cluster the EPAs of the use-case-unit EPN into three groups of EPAs: Monitor-EPA, Analyze-EPA, and Plan&Execute-EPA as depicted in Figure 9-7.

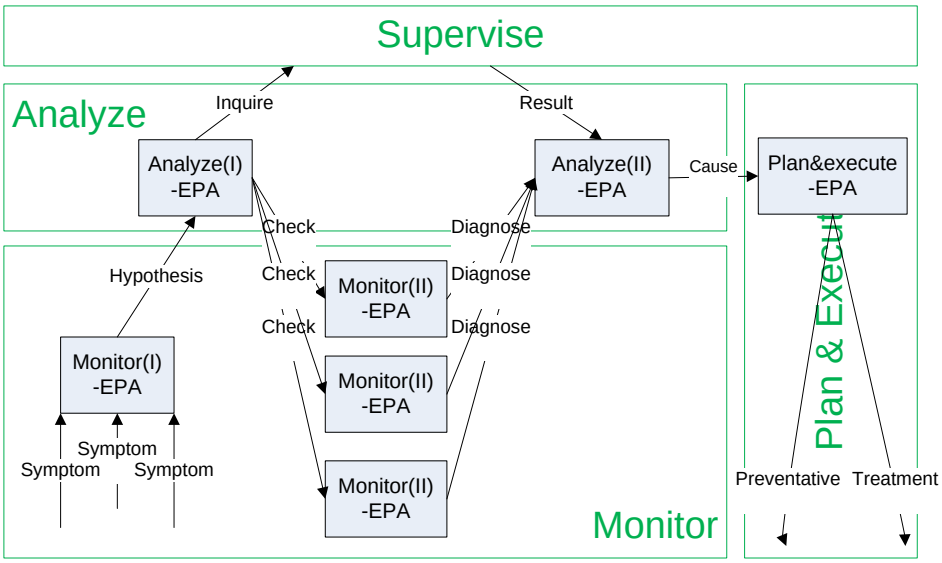


Figure 9-7: Three groups of EPAs for a generic use-case-unit EPN

When we instantiate the use-case-unit with a specific example, Use Case One Workflow, we can define a list of bottom-level subtypes for each one of the six event types (i.e., *symptom*, *hypothesis*, *check/inquire*, *diagnosis/result*, *cause*, and *treatment*) in the use-case-unit. The class diagrams depicted in Figure 9-8 include all the bottom-level subtypes for Use Case One.

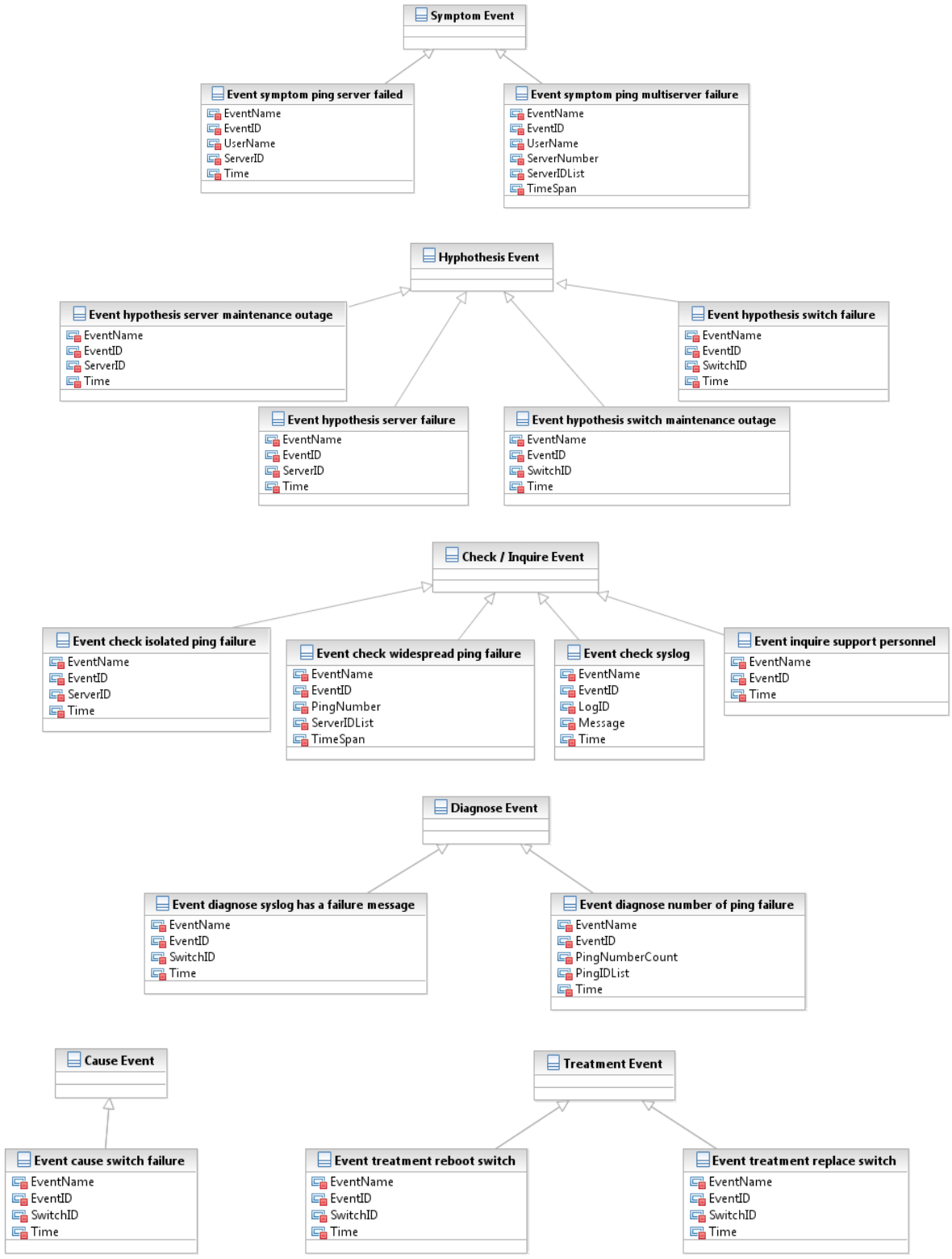


Figure 9-8: Event subtypes for Use Case One Unit

Table 23 shows the event subtypes we defined for Use Case One Unit. To make the event name easier to recognize by a human, we start the bottom-level-subtype names of events with *Event_* plus its subtype name, such as *Event_symptom*.

Table 23: Event types and event definitions in Use Case One Unit

Event type	Event subtype	Event description
Symptom	Event_symptom_ping_server_failed (EventName, EventID, UserName, ServerID, Time)	Symptom: can't ping one server
	Event_symptom_ping_multiserver_failure (EventName, EventID, UserName, ServerNumber, ServerIDList, TimeSpan)	Symptom: can't ping 100 servers within a short time span
Hypothesis	Event_hypothesis_server_maintenance_outage (EventName, EventID, ServerID, Time)	Hypothesis: server maintenance outage
	Event_hypothesis_server_failure (EventName, EventID, ServerID, Time)	Hypothesis: server failure
	Event_hypothesis_switch_maintenance_outage (EventName, EventID, SwitchID, Time)	Hypothesis: switch maintenance outage
	Event_ hypothesis_switch_failure (EventName, EventID, SwitchID, Time)	Hypothesis: switch failure
Check or Inquire	Event_check_isolated_ping_failure (EventName, EventID, ServerID, Time)	Check: isolated ping failure
	Event_check_widespread_ping_failure (EventName, EventID, PingNumber, ServerIDList, TimeSpan)	Check: widespread ping failure
	Event_check_syslog (EventName, EventID, LogID, "shutdown message",Time)	Check: syslog for shutdown message
	Event_check_syslog (EventName, EventID, LogID, "failure message", Time)	Check: syslog for failure message
	Event_inquire_support_personnel (EventName, EventID,Time)	Inquire: support personnel
Diagnosis or Result	Event_diagnose_syslog_has_a_failure_message (EventName, EventID,SwitchID,Time)	Diagnose result: syslog has a failure message
	Event_diagnose_number_of_ping_failure (EventName,EventID,PingNumberCount,PingIDList, Time)	Diagnose result: number of ping failure
Cause	Event_cause_switch_failure (EventName, EventID,SwitchID,Time)	Root cause: switch failure
Treatment	Event_treatment_reboot_switch (EventName, EventID,SwitchID,Time)	Treatment: reboot switch
	Event_treatment_replace_switch (EventName, EventID,SwitchID,Time)	Treatment: replace switch

Based on the event subtypes defined in Table 23, we can build the EPN for Use Case One One Unit (cf.

Figure 9-9) according to the Use Case One Workflow depicted in Figure 9-3 and Figure 9-4.

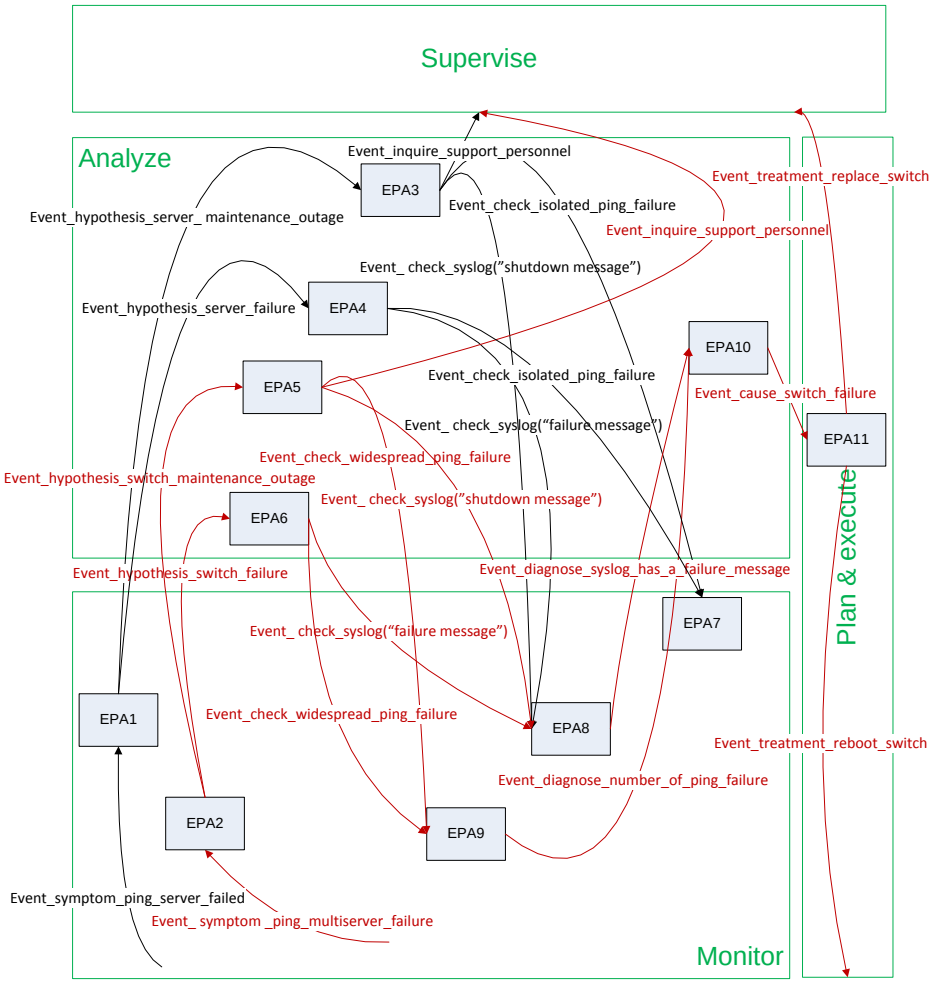


Figure 9-9: EPN of Use Case One Unit

The EPN of Use Case One Unit consists of eleven EPAs (EPA1 to EPA11) as defined in Table 24 and Table 25.

Table 24: EPAs used in Use Case One Unit (Part1)

EPA Type	EPA ID	Input event(s)	Output event(s)
Monitor (I)	1	Event_symptom_ping_server_failed (EventName, EventID, UserName, ServerID, Time)	Event_hypothesis_server_maintenance_outage (EventName, EventID, ServerID, Time); Event_hypothesis_server_failure (EventName, EventID, ServerID, Time)
	2	Event_symptom_ping_multiserver_failure (EventName, EventID, UserName, ServerNumber, ServerIDList, TimeSpan)	Event_hypothesis_switch_maintenance_outage (EventName, EventID, SwitchID, Time); Event_hypothesis_switch_failure (EventName, EventID, SwitchID, Time)
Analyze (I)	3	Event_hypothesis_server_maintenance_outage (EventName, EventID, ServerID, Time)	Event_check_isolated_ping_failure (EventName, EventID, ServerID, Time) Event_check_syslog (EventName, EventID, LogID, "shutdown message",Time); Event_inquire_support_personnel (EventName, EventID,Time)
	4	Event_hypothesis_server_failure (EventName, EventID, ServerID, Time)	Event_check_isolated_ping_failure (EventName, EventID, ServerID, Time); Event_check_syslog (EventName, EventID, LogID, "failure message",Time)
	5	Event_hypothesis_switch_maintenance_outage (EventName, EventID, SwitchID, Time)	Event_check_widespread_ping_failure (EventName, EventID, PingNumber, ServerIDList, TimeSpan); Event_check_syslog (EventName, EventID, LogID, "shutdown message",Time); Event_inquire_support_personnel (EventName, EventID,Time)
	6	Event_hypothesis_switch_failure (EventName, EventID, SwitchID, Time)	Event_check_widespread_ping_failure (EventName, EventID, PingNumber, ServerIDList, TimeSpan); Event_check_syslog (EventName, EventID, LogID, "failure message",Time)

Table 25: EPAs used in Use Case One Unit (Part2)

EPA Type	EPA ID	Input event(s)	Output event(s)
Monitor (II)	7	Event_check_isolated_ping_failure (EventName, EventID, ServerID, Time)	N/A (not provided by Use Case One)
	8	Event_check_syslog (EventName, EventID, LogID, "failure message",Time)	Event_diagnose_syslog_has_a_failure_message (EventName, EventID,SwitchID,Time)
	9	Event_check_widespread_ping_failure (EventName, EventID, PingNumber, ServerIDList, TimeSpan)	Event_diagnose_number_of_ping_failure (EventName,EventID,PingNumberCount, PingIDList, Time)
Analyze (II)	10	Event_diagnose_syslog_has_a_failure_message (EventName,EventID,SwitchID,Time); Event_diagnose_number_of_ping_failure (EventName,EventID,PingNumberCount,PingIDList, Time)	Event_cause_switch_failure (EventName, EventID,SwitchID,Time)
Plan & Execute	11	Event_cause_switch_failure (EventName,EventID,SwitchID,Time)	Event_treatment_reboot_switch (EventName, EventID,SwitchID,Time) OR Event_treatment_replace_switch (EventName, EventID,SwitchID,Time)

Through the Use Case One example, we illustrated a systematic scenario that constructing a use-case-unit EPN (i.e., an architecture diagram) from a real use case (i.e., in plain text such as Use Case One). For instance, we convert *cannot ping server* in Use Case One (cf. Figure 6-2) into an event subtype *Event_symptom_ping_server_failed* (cf. Table 23). Then, we define this event subtype as the input event type of EPA1. The output of EPA1 has two event types: *Event_check_server_maintenance_outage* and *Event_check_server_failure* (cf. Table 24). In further, these two output event types of EPA1 are defined as the input event types of EPA2 and EPA3, respectively. Therefore, all the EPAs are driven / triggered by events, just like dominos. When the first event *Event_symptom_ping_server_failed* come into EPA1, related EPAs within Use Case One EPN are triggered one after another by a series of events, the inference process of RCAD

is automated. If the switch in fact is failed, the expected output event of EPN of Use Case One Unit is *Event_treatment_reboot_switch* or *Event_treatment_replace_switch*, which can trigger the real treatment for the target system.

Figure 9-9 shows all the EPAs which form the use-case-unit EPN for Use Case One. We call this specific use-case-unit EPN for Use Case One as Use Case One Unit—an instance of generic use-case-unit EPN. This type of EPN constitutes the core part of our RCAD system.

The use-case-unit EPNs for Use Case Two and Three are described in Appendices B and C.

Summary

This chapter describes a systematic scenario that how to design / construct a use-case-unit EPN from a real use case. This means once a new use case similar as Use Case One is available, an RCAD system developer can design / construct a new use-case-unit EPN (e.g., an architecture diagram) by following this scenario. The design scenario is the first contribution of this chapter. The rationale behind this scenario is by replacing human operations with autonomic operations, we can relieve the system administrator's burden of some routinely performing RCAD tasks.

In this chapter, we identified six event types for Use Case One (also applied to Use Case Two and Three). All six event types are concluded from use case survey, which guarantees that these event types are proper and sufficient for all known use cases. All these events are processed by a collection of EPAs. This set of EPAs forms an EPN we defined as a generic use-case-unit. The generic use-case-unit EPN itself is the second contribution of this chapter.

To illustrate the approach, we instantiated the use-case-unit with a specific example: Use Case One Workflow. We defined a list of bottom-level subtypes for each one of the six event types (i.e., *symptom*, *hypothesis*, *check/inquire*, *diagnosis/result*, *cause*, and *treatment*) in the use-case-unit. Based on these event subtypes, we laid out the architecture diagram of our Use Case One EPN according to the Use Case One Workflow.

We validated our use-case-unit EPN approach by (1) describing a scenario that illustrates how to design/construct a use-case-unit EPN through a real use case and (2) providing a generic use-case-unit EPN. Correspondingly, we validated the use-case-unit EPN design scenario with existing real use cases. The steps of the design scenario for Use Case One are covered in Section 9.3. Use Case Two and Three are covered in Appendices B and C. We also validated the generic use-case-unit EPN through these real use cases at the architecture level. The use-case-unit EPN for Use Case One is depicted in

Figure 9-9. The EPNs for Use Case Two and Three are depicted in Appendices B and C.

In addition, we validated our new net-casting information seeking model. In Section 7.5, we described the "task" level details of a single iteration of our net-casting information seeking model (cf. Figure 7-11). In Section 8.6, we instantiated it with event types concluded from Use Case One, Two and Three (cf. Figure 8-3). Consequently, in this chapter, we validated the net-casting information seeking model with the generic use-case-unit EPN design (cf. Figure 9-6).

Chapter 10 Event Processing Autonomic Computing Reference Architecture

From an autonomic computing prospective, a use-case-unit EPN often achieves one single MAPE (Monitor, Analyze, Plan and Execute) loop with a slight variation in its loop characteristics—an extra loop between the Monitor and Analyzer. In other cases, a use-case-unit EPN can be simplified without such an extra loop when no further information is required. Based on the ACRA model presented by IBM, we introduce a variant of ACRA—*event processing autonomic computing reference architecture* (EPACRA). In the middle layer of the EPACRA there are use-case-unit EPNs. Equipped with CEP engines, these EPNs are able to process event flows from massive event clouds and automate RCAD processes with very little latency when a failure or symptom(s) appears. Compared to some computer-aided diagnoses such as the tool prototype built by Kontogiannis and Wong (cf. Figure 7-3), an RCAD tool based on EPACRA likes a "emergency room", which can offer urgent care when system has a failure. The shortcoming of such types of RCAD tools is they are not designed to identify all the symptoms. Those infrequent or unidentified failures are left to other computer-aided diagnose tools to process. Those computer-aided diagnose tools, in metaphor, are considered as "regular clinics".

10.1 Introducing an Extra Loop into the Autonomic Element

A use-case-unit EPN usually consists of a set of EPAs. Each EPA might be distributed over multiple locations in a network environment. Even though the autonomic element phases are not located necessarily on the same network component, we consider them as part of the same autonomic element.

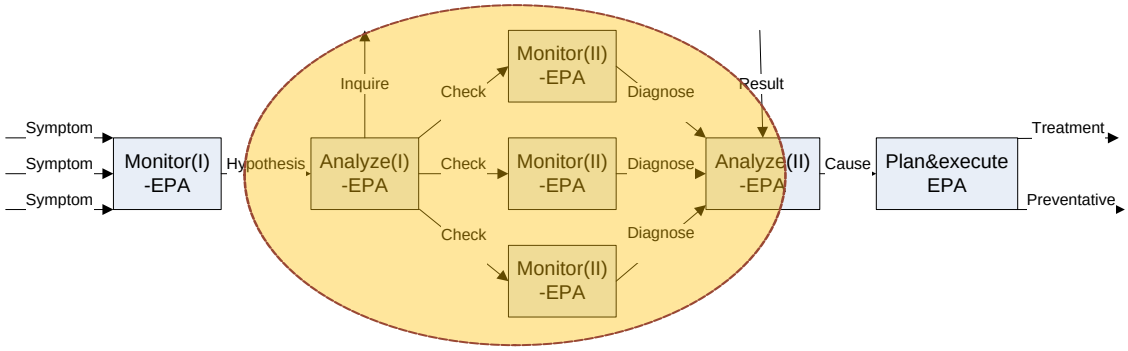


Figure 10-1: EPA level view of a use-case-unit EPN

EPAs in a use-case-unit EPN accept one of the following five types of events as input: *symptom*, *hypothesis*, *check/inquire*, *diagnose/result*, or *cause*. As a consequence, these EPAs generate one of the following five types of events as output: *hypothesis*, *check/inquire*, *diagnose*, *cause*, and *treatment*. The EPA-level view of a use-case-unit EPN is depicted in Figure 10-1. Table 26 lists the input event types and output event types for different types of EPAs:

Table 26: Input event types and output event types for EPAs

EPA type	Input event type	Output event type
Monitor(I)	Symptom event	Hypothesis event
Analyze(I)	Hypothesis event	Check or Inquire event
Monitor(II)	Check or Inquire event	Diagnose event
Analyze(II)	Diagnose or Result event	Cause event
Plan & execute	Cause event	Treatment event

By following the scientific method, usually a use-case-unit EPN will collect more information after *hypothesis* events have been generated, (i.e., EPA will send *check* events or *inquire* events as messages to search for more information that is necessary). The result is sent back as a *diagnose* event when the information that is requested for it is available or sent back as a *result* event by human operators.

From an autonomic computing prospective, a use-case-unit EPN achieves one single MAPE (Monitor, Analyze, Plan and Execute) loop with a slight variation in its loop characteristics. EPAs in the EPN of use-case-unit implement all four phases (i.e., Monitor, Analyze, Plan and Execute) of an autonomic element with an extra small loop between Monitor and Analyzer. The highlighted circle in Figure 10-1 includes those EPAs that compose the extra loop.

Figure 10-2 shows the extra loop in an autonomic element. This autonomic element represents a high-level view of a use-case-unit EPN. This view of the use-case-unit EPN is an abstraction of use-case workflows and does not show the level of EPAs. The highlighted circle in Figure 10-2 is an abstracted view of the highlighted circle in Figure 10-1, which emphasizes that repetitive diagnosis/inference process of a use-case workflow includes an extra loop between Monitor and Analyzer in the autonomic element.

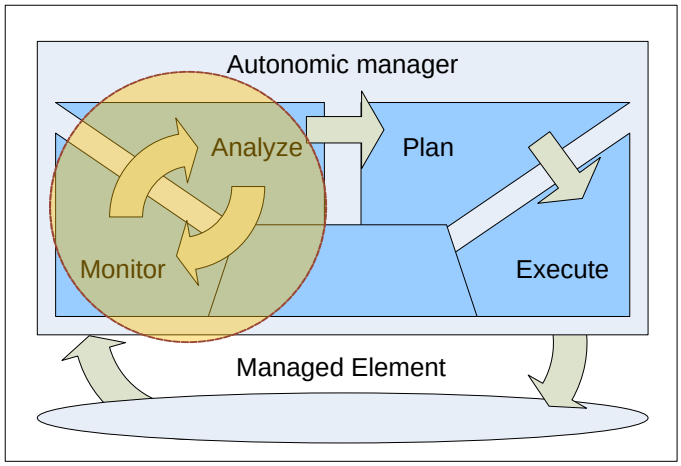


Figure 10-2: An extra loop between Monitor and Analyzer

10.2 Events in an Autonomic Element

Drilling down into the lower layers of a use-case-unit EPN, we notice that an autonomic manager is implemented with a combination of EPAs. In Figure 10-3, when

some anomaly is detected and a *symptom* event is generated, Monitor(I)-EPA will analyze the *symptom* event and produce a *hypothesis* event which will further generate a *check* or *inquire* event by Analyze(I)-EPA. *Check* events will then be sent out to Monitor(II)-EPA to collect additional information. An *inquire* event will be sent to human operators.

In the second round of monitoring, ideally all essential information will be collected by Monitor(II)-EPA and several *diagnose* events will be generated. Analyze(II)-EPA will analyze these *diagnose* events and discover the root cause. A *cause* event will be generated by Analyze(II)-EPA. The *cause* event will now be used by Plan&Execute EPA to produce a *treatment* event or an additional *preventative* event. Either event will be sent out to serve the self-healing or self-optimizing goal.

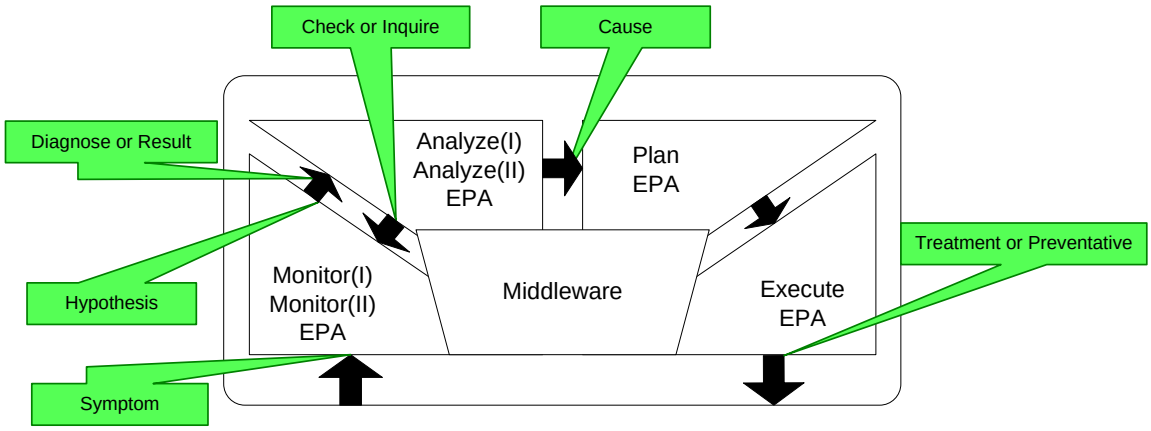


Figure 10-3: EPAs of a use-case-unit EPN comprise an autonomic element

By applying the event types as stated in Table 26 to some specific examples such as User Case One, we are able to define all the event types as outlined in Table 23 in detail. Figure 10-4 depicts all the EPA IDs and event types in the autonomic element elaborated from Figure 10-3.

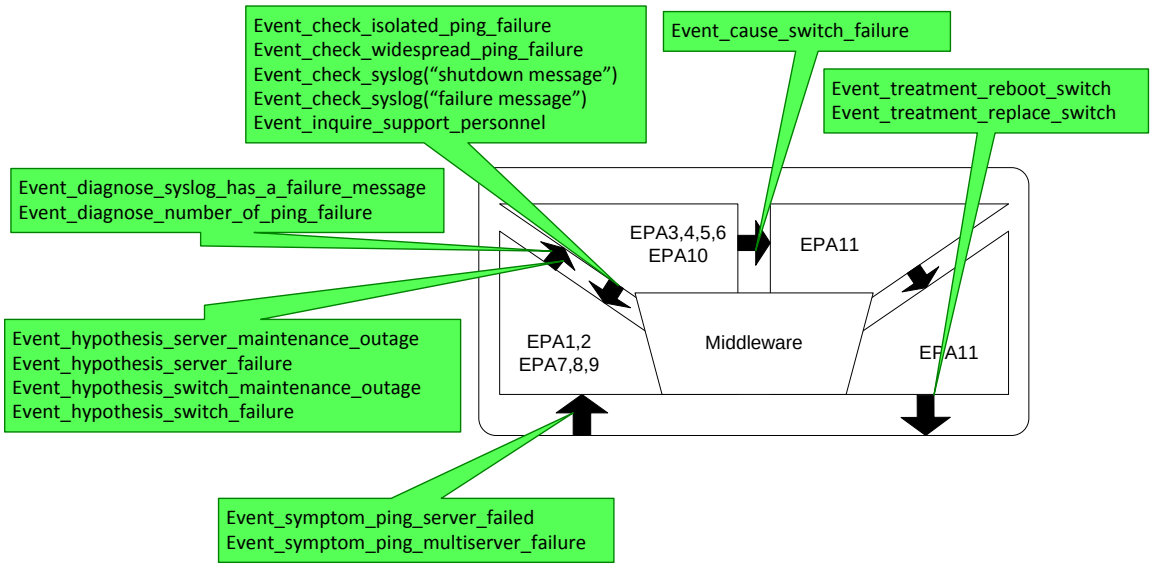


Figure 10-4: Event types in the autonomic element

From an architecture pattern prospective, if we consider that the Monitor-EPAs do not belong to the same autonomic element as the rest of the EPAs. The EPAs of User Case One can nicely fit into the chain-of-monitors architecture pattern (chain-of-responsibility variant). The scenario of the chain-of-monitors architecture pattern is as follows: when each monitor performs its part of the task and then passes the request to the next monitor until all the actions have been performed. Figure 10-5 depicts the chain-of-responsibility variant of the chain-of-monitors pattern.

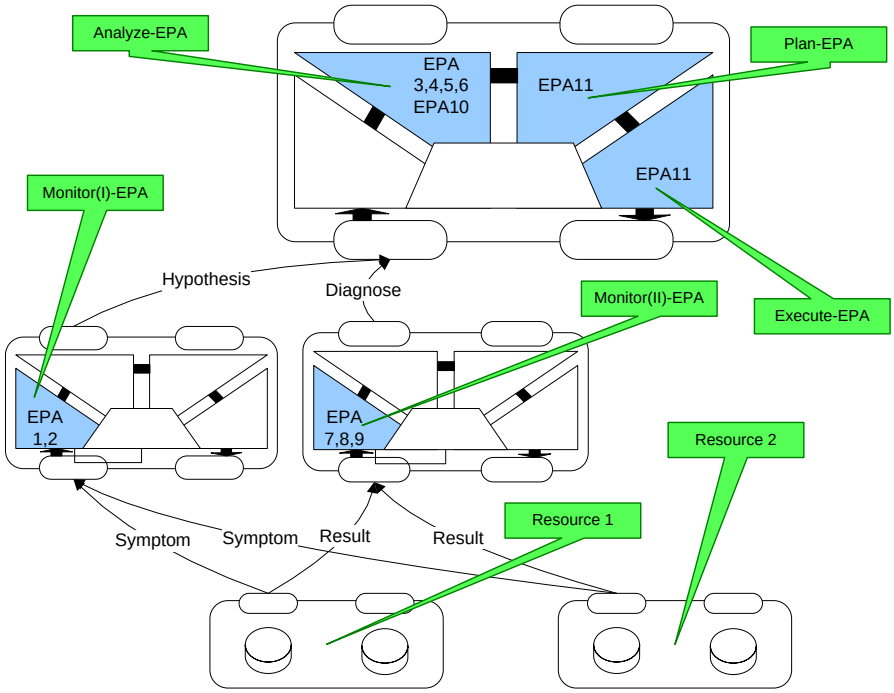


Figure 10-5: EPAs of a use-case-unit EPN to demonstrate the chain-of-monitors pattern

10.3 Simplified Use-Case-Unit EPN

In some cases, no extra information is required for Monitor(I)-EPA to identify the actual problem based on the given symptoms. In such a case, the corresponding EPN of a use-case-unit can be simplified. The Monitor(I)-EPA will generate a *diagnose* event instead of a *hypothesis* event. The simplified EPN only need to deal with three types of events: *symptom*, *diagnose/result*, and *cause*. Correspondingly, the EPN generates three types of events: *diagnose*, *cause*, and *treatment/preventative*. Figure 10-6 depicts a simplified use-case-unit EPN. Table 27 represents the corresponding list of input event types and output event types for the different types of EPAs in Figure 10-6.

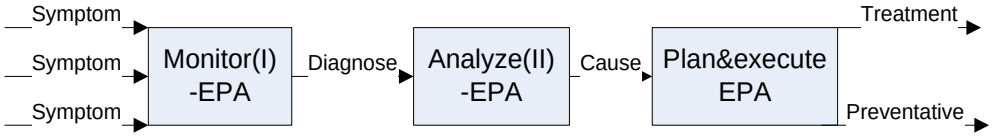


Figure 10-6: EPA level view of a simplified use-case-unit EPN

Table 27: Input event types and output event types for EPAs

EPA type	Input event type	Output event type
Monitor(I)	Symptom event	Hypothesis event
Analyze(II)	Diagnose or Result event	Cause event
Plan & execute	Cause event	Treatment event

From an Autonomic Computing prospective, a simplified use-case-unit EPN achieves one single MAPE loop exactly as the original one proposed by IBM [IBM06] (Figure 10-7). EPAs in the EPN of simplified use-case-unit EPN implement all four phases (Monitor, Analyze, Plan and Execute) of an autonomic manager without an extra loop between Monitor and Analyzer.

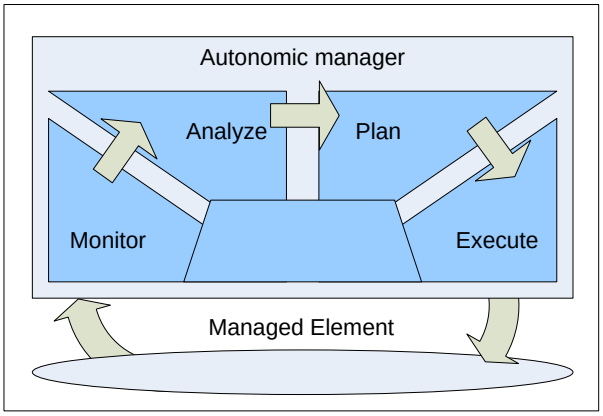


Figure 10-7: No extra loop between Monitor and Analyzer in an autonomic element

The following sequence of events happens in our RCAD process when the cause-effect model of a symptom is straightforward and no extra information is needed. In Figure 10-8, when some anomaly is detected and a *symptom* event is generated, Monitor(I)-EPA will analyze the *symptom* event and produce a *diagnose* event. Then, Analyze(II)-EPA will analyze the *diagnose* event and discover the root cause. As the result, the *cause* event will be generated by Analyze(II)-EPA, which will now be used by

Plan & Execute EPA to produce a *treatment* or *preventative* event. Either event will be sent out to achieve the self-healing or self-optimizing goal.

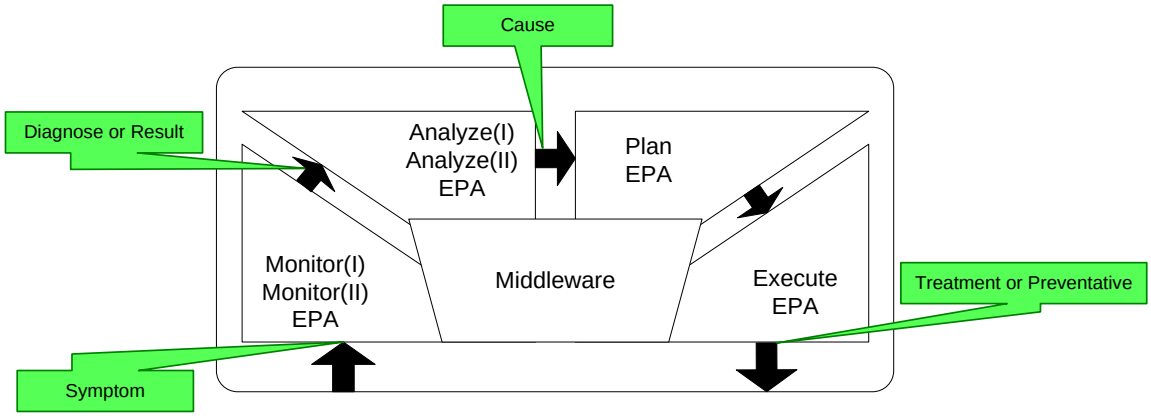


Figure 10-8: Autonomic manager consisting of EPAs for a simplified use case unit

10.4 Cause-and-effect Diagrams for Use Case One

Kontogiannis and Wong built a tool prototype (cf. Section 7.3) for computer-aided diagnoses using auto-prompted cause-and-effect diagrams (which are called fishbone diagrams in this context). For Use Case One, the scenario of finding the root cause *switch failure* might involve four fishbone diagrams: FD1, FD2, FD3, and FD4. All four fishbone diagrams are depicted in Figure 10-9.

In fishbone diagram FD1, effect *can't ping one server* (cf. fish head) has four causes: C1 (*server maintenance outage*), C2 (*server failure*), C3 (*switch maintenance outage*) and C4 (*switch failure*). In fishbone diagram FD2, effect *can't ping 100 servers within a short time span* has two causes: C1 (*switch maintenance outage*) and C2 (*switch failure*). In Fishbone diagram FD3, effect *wide spread ping failure* could have multiple causes. One of them is C1 (*switch failure*). In fishbone diagram FD4, effect *switch syslog has failure message* also could have multiple causes. One of them is C1 (*switch failure*).

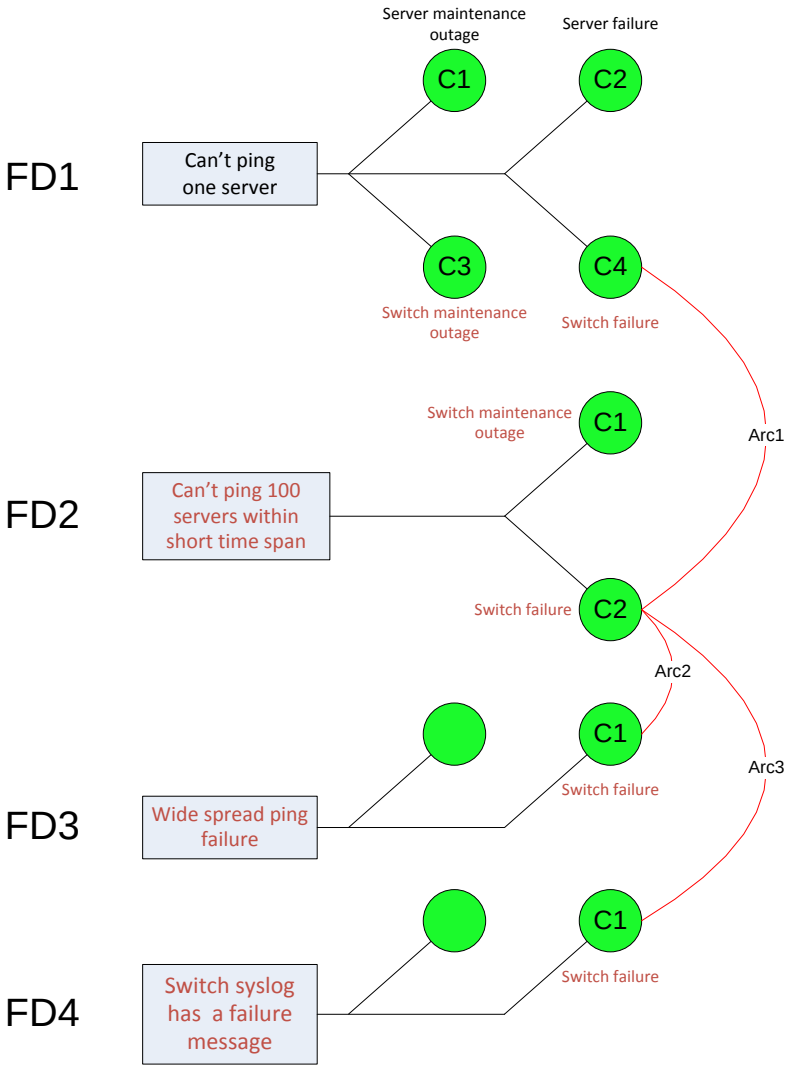


Figure 10-9: Cause-and-effect diagrams for Use Case One

When an IT administrator investigates an effect, he/she will check all the causes in FD1 with respect to the effect *can't ping one server*. Some causes will contribute to other effects, such as C4 (*switch failure*) in FD1. The tool will automatically identify the fishbone diagrams with the same effects. At this moment FD2 is auto-prompted. Since C2 (*switch failure*) in FD2 and C4 (*switch failure*) in FD1 actually the same cause, they are linked by a red color arc (i.e., Arc1).

If the effect *can't ping 100 servers within a short time span* in FD2 exists and when an IT administrator investigates C2 (*switch failure*) in FD2, the tool will provide two more fishbone diagrams FD3 and FD4, automatically.

Because the same cause *switch failure* contributes to the fishbone diagrams FD3 and FD4, C2 (*switch failure*) in FD2 is linked to C1 (*switch failure*) in FD3 by red color Arc2, and linked to C1 (*switch failure*) in FD4 by red color Arc3, respectively. If the administrator also observes the effects *wide spread ping failure* and *switch syslog has failure message*, the cause *switch failure* therefore is concluded as the root cause of the investigated effect *can't ping one server*. In such scenario, the RCAD process is performed semi-automatically with the assistance of this diagnostic tool.

10.5 Event Processing Autonomic Computing Reference Architecture

Luckham identified three key technological requirements to achieve the goal of autonomous parallel processing [Luck05].

1. The process should be completely automated and event driven.
2. The human will be taken out of the loop to make all the activities in the process fully autonomous.
3. The human will be kept in control over the processes by being provided with real-time viewing to make decisions.

Considering all use cases as RCAD processes, we propose to satisfy the above three key technological requirements in RCAD to achieve the goal of autonomic and parallel processing as follows:

1. By leveraging CEP technology, all use-case-unit EPNs should be completely automated and event-driven.

2. Humans are taken out of the loop from each use-case-unit EPN. Each EPA of the EPN can make decisions and communicate asynchronously. Thus, all the use-case-unit EPNs can be executed in parallel.

3. Humans are still in control over the process, which requires IT administrators to have understandable, personalized views of activity at every level. We need on-the-fly modifiable process rules, including exceptional situation handling, to modify and control processes.

Based on the ACRA model presented by IBM, we introduce a variant of ACRA—event processing autonomic computing reference architecture (EPACRA). The variation is based on CEP technology, real use-case workflows and our net-casting information seeking model. As autonomic resource managers, EPNs (i.e., a collection of EPAs) orchestrates managed resources in the middle layer of EPACRA (cf. Figure 10-10).

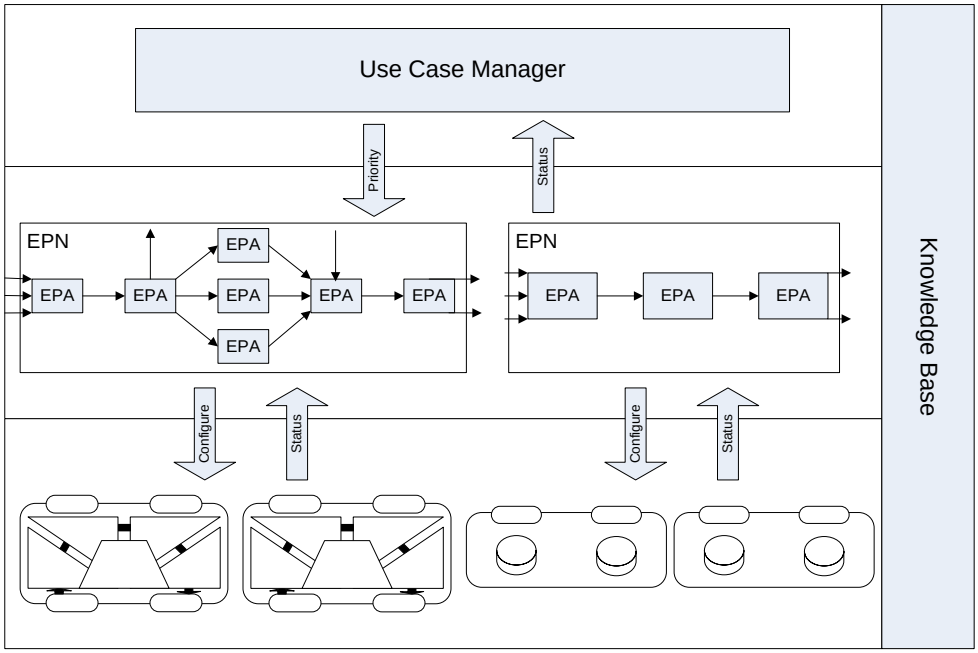


Figure 10-10: EPACRA model: EPA level view

The lowest layer contains the system components or managed-resources. These resources may or may not have embedded, self-managing attributes. The middle layer contains EPNs which can be categorized as self-healing or self-protecting. Individual resources may have one or more EPNs, each implementing a relevant RCAD use case. Compared to the conventional ACRA, the endpoints are simplified since the information that goes in and comes out of an autonomic element is just events, which have already been clearly defined. The top layer contains the use-case manager that orchestrates use-case-unit EPNs. These use-case managers can either be manual or autonomic by incorporating control loops that realize broad objectives of the overall IT infrastructure.

Figure 10-11 represents a high level view of EPACRA. This EPACRA view is an abstract representation in the form of autonomic elements without showing the level of EPAs. Please note that the model includes both a typical use-case workflow (requiring an extra loop between Monitor and Analyzer in the autonomic element) and a simplified use-case workflow (without a loop between Monitor and Analyzer in the autonomic element).

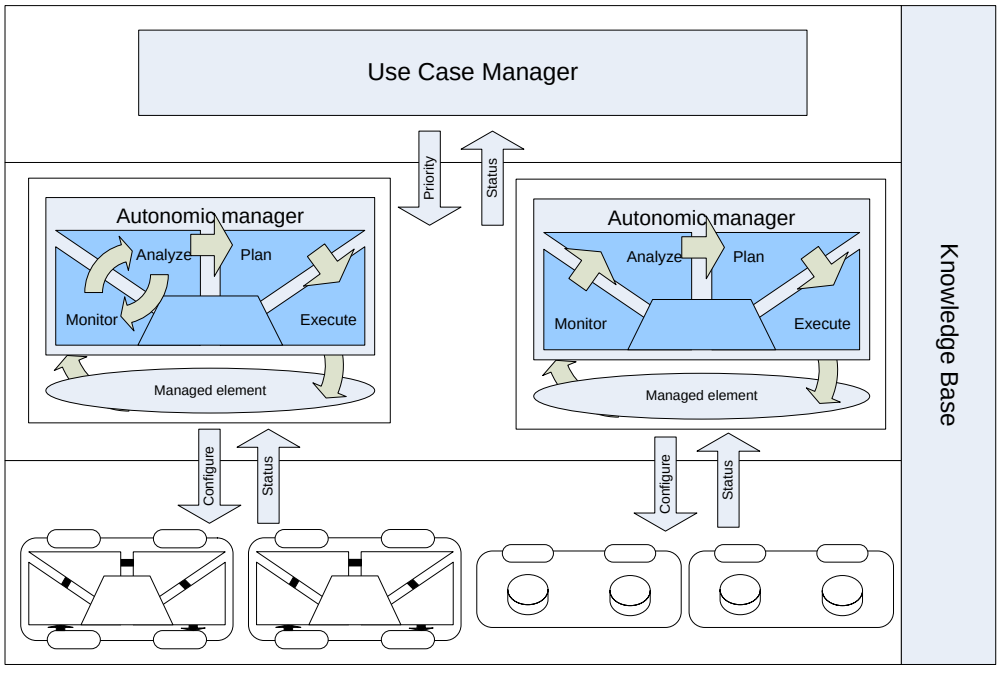


Figure 10-11: EPACRA model: Abstract view

Since EPNs are constructed with CEP technology, they will benefit from flexibilities provided by CEP engines such as [Luck05]:

- On-the-fly modifiable process rules, including exceptional situation handling, to modify and control processes; and
- On-the-fly evolution means the ability to modify a process without halting the rules engines or disrupting the execution of other processes.

Furthermore, different EPAs might be located on different machines in a distributed environment. So they are required to share the same middleware (e.g., such as a publish/subscribe middleware) to pass events. This type of architecture could be very dynamic. For example, a hypothesis event may decide which instance of Analyze(I)-EPA will be involved and how many instances of Analyze(I)-EPA will be involved. Corresponding instance of Analyze(I)-EPA can even be created when they are needed.

Summary

From an autonomic computing perspective, EPAs in the EPN of a use-case-unit often implement all four phases (i.e., Monitor, Analyze, Plan and Execute) of an autonomic element with an extra small loop between Monitor and Analyzer. In other cases, a use-case-unit EPN can be simplified without such an extra loop when no further information is required. This chapter illustrated how an EPN achieves the four phases of an autonomic element, and how events flow between these four phases, in both of these two cases. Matching use-case-unit EPNs to corresponding autonomic elements is the first contribution of this chapter.

We validated the new chain-of-monitor architecture pattern with the use-case-unit EPN design (cf. Section 10.2), since we did not have such an example at hand. In Section 10.2, the EPAs of a use-case-unit EPN comprise an autonomic element (cf. Figure 10-3) and demonstrate the chain-of-monitor pattern (cf. Figure 10-5).

Based on the ACRA model presented by IBM, we introduced a variant of ACRA—the *event processing autonomic computing reference architecture* (EPACRA)—which is the second first contribution of this chapter. The variation of ACRA, EPACRA, was derived from CEP technology, real use-case workflows and the net-casting information-seeking model.

From validation viewpoint, the use-case-unit EPN design, as an instance, validated the middle layer of the EPACRA model (cf. Figure 10-10)—a variant of ACRA model (Figure 2-3).

Chapter 11 A Case Study

Most diagnosis tools help users troubleshoot problems by making it easy to visualize and correlate data within and across systems. It is important to point out that obtaining a better view of the problem data is not the same as finding the root causes of a problem. Root causes usually hide deep inside systems and are often difficult to detect—like finding a needle in a haystack. Only experienced users with strong general troubleshooting skills, good knowledge of the application, and comfort with and access to the right tools will be able to accomplish the job effectively. This situation certainly leaves many opportunities for researchers to contribute and make improvements, especially improving the effectiveness and efficiency of computer-aided RCAD processes. The approach proposed in this dissertation is to leverage CEP technology and make the RCAD process autonomic through properly designed and implemented use-case-unit event processing networks (EPNs). This chapter validates the *generality* of the use-case-unit EPN approach through a case study in a university-laboratory setting (cf. Section 1.3. *Theory* is a set of statements that explain a set of phenomena. Ideally, the theory has predictive power—that means the theory has some degrees of *generality*).

11.1 Research Platform

The subject system we use for our case study is a Web application system developed by Lin [Lin10], who was an M.Sc student in our research group at the University of Victoria. It is a stock trading system called Daytrading system, which handles customer stock operations such as buying and selling, based on the requirements provided by a fictional company called Stocktrading Inc. The system architecture comprises seven components (cf. Figure 11-1): Firewall, Load Balancer, Web Server & Transaction Server Cluster, Account Manager, Cache, Database Cluster, and Monitor & Management Platform.

The Daytrading system is designed to support multiple users to trade stocks online simultaneously. The system is composed of several individual components: *web server*, *transaction server*, *cache server*, *database server* and the *quote server* (cf. Figure 11-1).

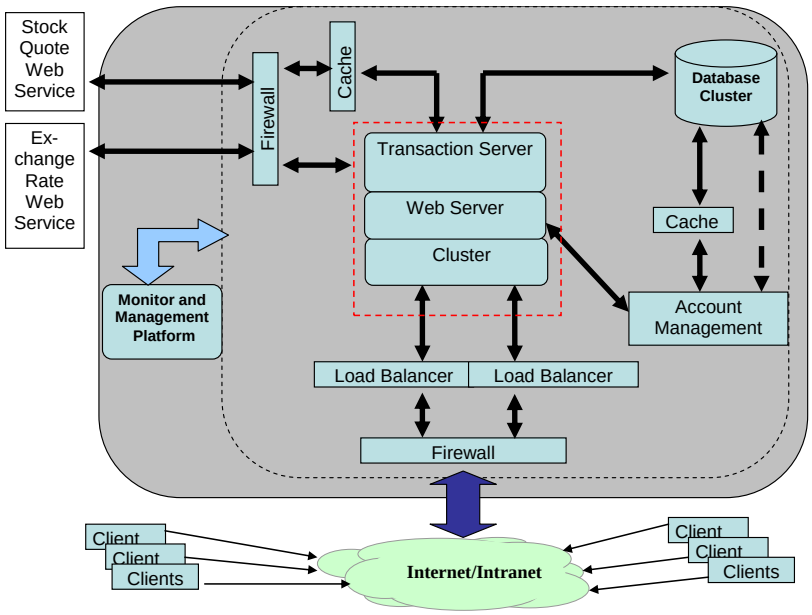


Figure 11-1: Architecture overview of the Daytrading system

The Daytrading program is written in Java and comprises three packages: *user interface*, *account management*, and *transaction management*. A high level architectural diagram of the Daytrading program is depicted in Figure 11-2 below.

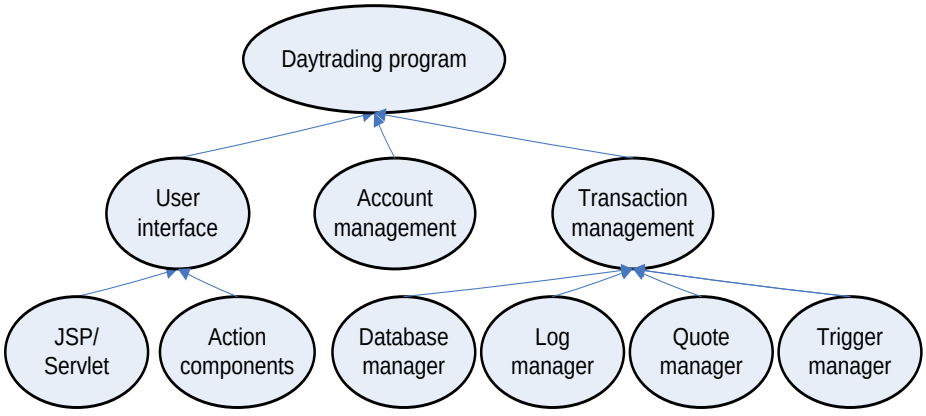


Figure 11-2: High level architecture of the Daytrading program

Besides the functional requirements provided by the fictional company StockTrading Inc., the goal of Lin's project [Lin10] was to build a scalable, secure, high availability, high performance, and low-cost system as an experimental platform for RCAD research.

11.2 Quality Criteria

Based on the ISO 9126 quality model, we considered the security factor of functionality, the recoverability and fault tolerance factors of reliability, the time behaviour (performance) factor of efficiency, and the changeability factor of maintainability [Zhu08]. These selected quality criteria are treated as preferred non-functional requirements in the system. Figure 11-3 depicts the goal tree (cf. Section 7.2) for the following non-functional requirements: *changeability*, *fault tolerance*, *recoverability*, *security* and *performance*. All sub-goals that fulfill those non-functional requirements are selected based on the recommendations of textbook *Distributed Systems: Concepts and Design* [CDK01] and our own experience. Table 28 lists selected non-functional requirements as soft goals and means to satisfy these goals.

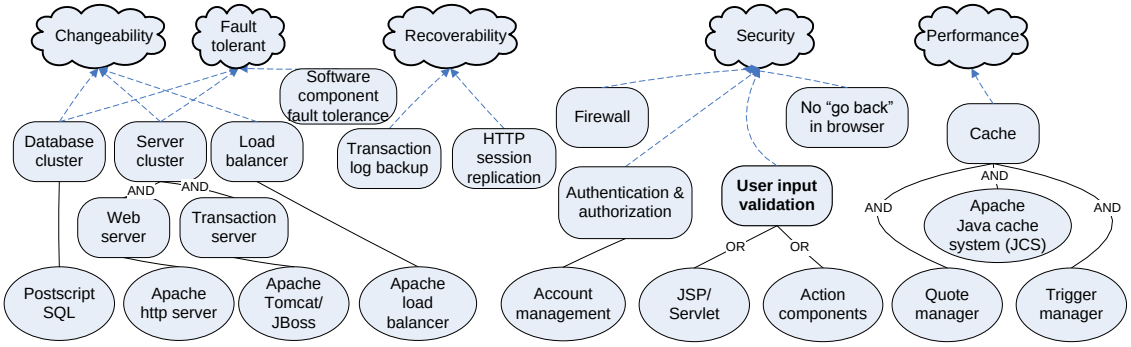


Figure 11-3: Scalability and availability goals

Table 28: Non-functional requirements and the means to satisfy them

Non-functional requirements	Means to satisfy these soft goals
Changeability	A simple Web application system only needs a Web server, a transaction server, and a database server. In order to accommodate growth in the number of clients, a Web application system should be designed to meet the changing capacity requirements through upgrading or adding components. Using load balancing and clustering technologies, it is easy to add a server node into the cluster to increase system capacity.
Fault tolerance	Although one Linux machine is enough to support the basic services required by our fictional StockTrading Inc., the system cannot be considered as fault tolerant, even though a Windows/Linux system installed in a server usually achieves 99% availability. Using load balancing and clustering technologies in the system, each server in the system would be a fault-tolerant component. The service will not be completely shut down if only one of the servers in the cluster is malfunctioning. In addition, the transaction logs are critical for a financial institute to recover from a disaster. They should be duplicated in the database and file system.
Security	The platform is designed to be a secure system that protects itself from malicious attacks. From the administration perspective, the attacks may come from both external and internal sources. The most prominent approach to prevent outside attacks is to set up hardware firewalls. All traffic is screened when a firewall is encountered. The internal software firewall on each machine can be used to block illegal connections. We are not only concerned with the security on the server side, but also on the client side. A Web page is the only mechanism with which a client can send information to the system. It is important to keep invalid messages from the system, particularly messages which include attack information. We can set up strict policies to validate the user input for preventing invalid messages and selected attacks.
Performance	Our Daytrading system uses quote server and trigger manager to shorten the average response time for each request and optimize the overall performance of the system.

11.3 Case Study: Prototype of Intrusion Detection

There are two types of Web application system requirements failures: incorrect inputs that generate correct outputs (i.e., intrusions), and correct inputs that generate incorrect outputs (i.e., system malfunctions). We chose the former type to construct a prototype implementation of the analytical tool to demonstrate the automation of the RCAD process. Therefore, we implemented an intrusion detection system, which is able to

recognize two types of malicious attacks on web application systems, and then take actions to prevent those intrusion attempts.

The prototype performs the following actions:

- It diagnoses whether the monitored Web application is under malicious attack, such as Cross-site Scripting (XSS) or SQL Injection.
- When an abnormal request (from client to web server) is detected, a hypothesis of *intrusion might occur* is generated.
- The response (from the Web server) is checked. If the response includes abnormal (protected) information, then an intrusion has occurred. Actions are then undertaken to prevent those intrusion attempts. If the response is normal, then intrusion did not occur.

With this prototype we validated both the use-case-unit EPN design scenario and the generic use-case-unit EPN itself. In particular, we showed how to automate the RCAD process for this case study. Please be aware that the goal of this case study is *not* building a sophisticated intrusion detection tool.

11.3.1 Intrusion Detection on Daytrading System

Based on the Daytrading system, we built an intrusion detection system to recognize the two different types of malicious attacks on web application systems, Cross-site Scripting (XSS) and SQL Injection. In case the Daytrading system has no capability to prevent those two types of intrusions, the intrusion detection system will take actions to prevent those intrusions from occurring.

Hermosillo et al. designed a security aspect called AproSec for detecting Cross-site Scripting and SQL Injection [HGSD07]. There are two drawbacks to their approach though. The first is its efficiency. The AproSec aspect validates and filters whatever HTTP request arrives. This process is unnecessary for many applications that already

have the capability to prevent certain types of intrusions. Our design uses AspectJ to check both incoming requests (from clients) and outgoing responses (from the server) for the intrusion attempt and then performs actions accordingly. The process reflects our inquiry-based RCAD approach. The second issue is scalability: we use a database rather than an XML format configuration file to store validation information (i.e., what tags should be checked in requests). Our prototype externalizes application logic into rule engines (one is JBoss rule engine, another is CEP technology)—a design strategy that not only leads to improved maintainability, but also to knowledge-sharing among incoming and outgoing message checking. The design and implementation of JBoss rule engine approach is discussed in Lin's thesis [Lin10]. This dissertation only covers the CEP technology-based approach.

11.3.2 Intrusion Detection System Design

Our intrusion detection system based on CEP technology consists of seven components: Sensor EPA (symptom generator), Monitor(I)-EPA (hypothesis generator), Analyze(I)-EPA (check generator), Monitor(II)-EPA (diagnose generator), Analyze(II)-EPA (cause generator), Plan EPA (treatment generator), and Execute EPA (intrusion preventer).

- **Sensor EPA** monitors the HTTP requests which users send from a Web browser through the Internet. HTTP requests will be monitored and identified if they are normal or abnormal according to the rules defined by the CEP agent. While normal requests are passed to the Web Applications without intervention, abnormal requests are screened and a symptom event will be generated accordingly.
- **Monitor(I)-EPA** will generate a hypothesis of the root cause after it receives a symptom event from sensor EPA.
- **Analyze(I)-EPA** will generate and send a set of check events to **Monitor(II)-EPA** to fetch further information related to the current hypothesis.

- **Monitor(II)-EPA** monitors the HTTP responses which the Web applications send back to the client. Similar to HTTP requests, responses will be monitored and identified as normal or abnormal according to the rules defined by the CEP agent.
- **Analyze(II)-EPA** will confirm or deny the hypothesis according to the diagnose event sent from **Monitor(II)-EPA**.
- **Plan & Execute EPA** will decide if Intrusion prevention should be activated. While normal responses are passed to clients without intervention, abnormal requests will be filtered by this component.

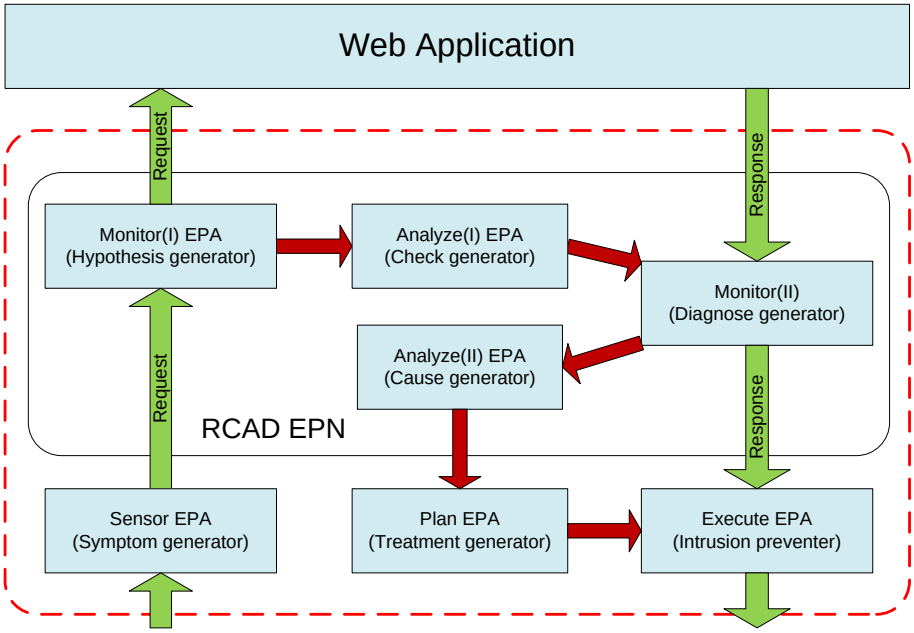


Figure 11-4: EPAs of our intrusion detection system

In component level, Sensor EPA (Symptom generator) and Execute EPA are implemented based on the AspectJ monitoring and executing technology which resides on the Web server. Other components, including the Monitor(I)-EPA, Monitor(II)-EPA, Analyze(I)-EPA, and Analyze(II)-EPA, are implemented based on the CEP technology.

11.3.3 Intrusion Detection Work Flow

Figure 11-5 depicts the workflow of an intrusion-detection process. When Sensor EPA of our intrusion detection system *finds an abnormal request* from the client, so a *hypothesis* event of *intrusion might occur* will be generated by Monitor(I)-EPA. This results in a *check* event of *check response format* being generated from Analyze(I)-EPA and sent to the Monitor(II)-EPA. If the returning *diagnose* event shows the response message is in *normal response format*, it indicates that an *intrusion has not occurred*. If the returning *diagnose* event shows the response message is in *abnormal response format*, it indicates that an *intrusion has occurred*. A corresponding *cause* event of *intrusion has occurred* is generated by Analyze(II)-EPA and sent to the Plan EPA. Then the Plan EPA will generate a *treatment* event of *start intrusion prevention component* and send it to the Execute EPA, from which a treatment process will be undertaken.

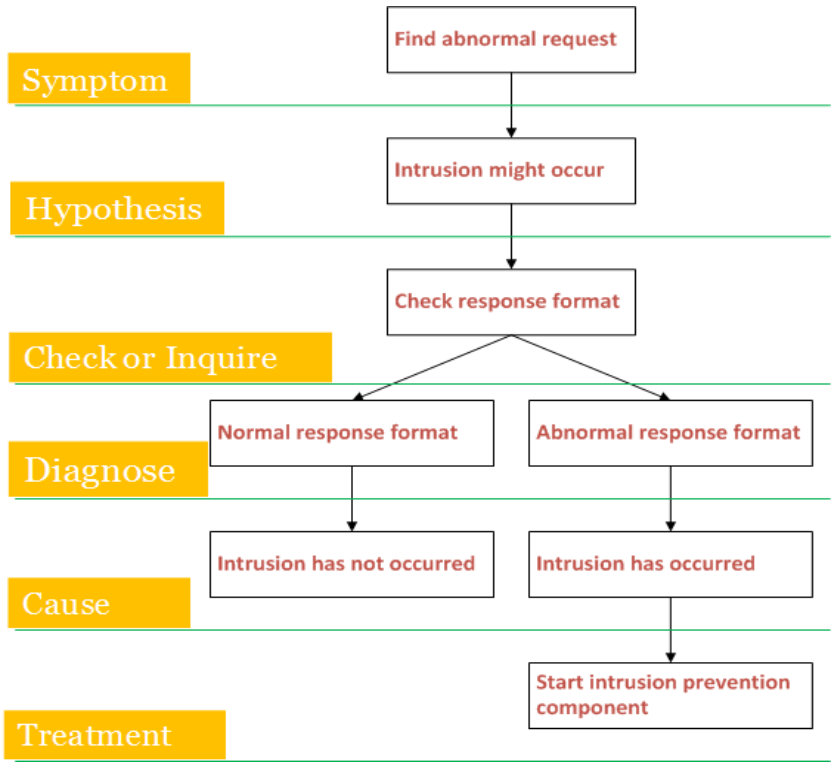


Figure 11-5: Workflow of the intrusion-detection process

11.4 Design Realization

Properly defining complex events and event patterns using CEP engines and EPLs, respectively, is crucial to achieve RCAD goals. We chose Esper to automate this process (cf. Chapter 8). Appendix D provides technical details on how we use the Esper framework. Appendix E provides implementation details of the Esper engine instance.

Summary

This chapter introduced a research platform for adaptive RCAD called Daytrading system. Besides architecture and software components, non-functional requirements of this system, especially some security-related issues were discussed. Based on the security-related requirements, we constructed a prototype implementation of the analytical tool to demonstrate the automation of the RCAD process. Specifically, an intrusion-detection system that is able to recognize and prevent two types of malicious attacks on web application systems was implemented. As a case study, it validates both the chain-of-monitors pattern and the net-casting model. It also validated the *generality* of the use-case-unit EPN design scenario and the use-case-unit EPN through the design of an intrusion detection system (cf. Figure 11-4 and Figure 11-5). At last, the implementation of the intrusion detection system validated the feasibility of the EPN approach with Esper.

Chapter 12 Conclusions

After summarizing and comparing the four research methodology types as categorized by Creswell [Cres02] and discussed by Easterbrook [ESSD07] in Section 1.3, we classified our research methodologies to fall into the *positivism* or the *pragmatic* (i.e., *mixed-method*) categories. Once we clarified the concepts of theory, model, hypothesis, validation vs. verification, we presented and validated the following *theories* throughout this dissertation:

- A set of architecture patterns
- The lattice of autonomic architecture patterns
- The new Chain-of-monitor architecture pattern
- Net-casting information seeking model
- The use-case-unit EPN designing scenario
- The generic use-case-unit EPN
- The feasibility of EPN approach
- The middle layer of the EPACRA model

12.1 Contributions

This section articulates the major contributions of this dissertation.

Autonomic computing and three layer architecture survey: Chapter 2 introduced the concepts of autonomic computing, autonomic element, as well as IBM's ACRA model. It surveyed industry standards related to autonomic computing technologies and reference implementations. It also covered popular open source projects widely available on the Web. Chapter 2 reveals that in our research context, the three-level hierarchy of self-* properties identified by Salehie [Sale09] can be achieved by an ACRA-model-like three layer architecture widely shared by many robotic systems, control systems and autonomic

computing software systems. Based on this discovery, we recognized that the main effort of this dissertation is to concentrate on how to construct *resource managers*—essentially, autonomic elements in the middle level of the ACRA model.

Lattice of autonomic computing architecture patterns: Chapter 3 identified a set of autonomic computing architecture patterns based on the formation of the control loops. Control loops are critical to control systems in general and autonomic computing systems, such as our RCAD system, in particular. Such control (or feedback) loops can operate independently, form a coherent hierarchy, or work collaboratively towards common goals. Our research identified six patterns of autonomic elements [ZLKM08]. However, that paper did not address how feedback loops form individual autonomic elements and how autonomic elements interact with each other. Sweitzer and Draper presented eight application patterns (i.e., *not* architecture patterns) and demonstrated how those patterns should be applied in autonomic systems to solve specific low-level problems [SwDr07]—for example, how to use partial autonomic managers as building blocks to compose a full autonomic manager (e.g., Application Pattern 3a discussed in Chapter 3). Based on the six out of the eight application patterns (cf. Pattern 1a, 1b, 2, 3a, 3b and 4 discussed in Chapter 3) identified by them, Chapter 3 clearly illustrates for the first time a set of autonomic computing architecture patterns by structure diagrams. After illustrating these architecture patterns, we observed the ambiguity in the aggregator-escalator-peer pattern. The ambiguity problem was solved by adding a new architecture pattern called *chain-of-monitors* into the updated lattice of autonomic computing architecture patterns.

Net-casting information seeking model: Bates introduced an information-seeking model called *Berry-picking* to represent evolving search processes in real-life manual source searches, which is analogous to the action of picking huckleberries or blueberries in a forest. From a human-computer interaction point of view, information-seeking processes range from heavily interactive processes with intensive human intervention to light interactive processes with little human intervention (i.e., autonomic information seeking process). In the latter case, a large portion of the workload (i.e., especially some

routinely performed tasks) shift from human operation to autonomic operation. A new information seeking model was needed to depict this type of information-seeking process, which we call the *net-casting process*. Based on the survey of traditional information seeking models, including Shneiderman's model [SBC98], Marchionini's model [Mar97] and Hearst's model [Hear99] conducted in Chapter 4, Chapter 5 proposed the five-stage net-casting model includes: (1) Recognize the problem, (2) Formulate queries, (3) Review the results, (4) Treat the problem, and (5) Reformulate. These stages are named and defined based upon the study and comparison of traditional information seeking models we made in Chapter 4.

Event processing autonomic computing reference architecture (EPACRA): EPACRA is proposed based on (1) *complex event processing* (CEP) technology introduced in Chapter 8, (2) real use-case workflows introduce in Chapter 7, and (3) the net-casting information-seeking model introduced in Chapter 5. EPACRA should be seen as a variant of the IBM ACRA model, aided with CEP to deal with large event clouds in real-time environments. Instead of one centralized *event processing agent* (EPA), the use-case-unit of RCAD features a distributed *event processing network* (EPN) which often consists of many EPAs to provide more adaptability and flexibility. The realization of EPACRA helps to relieve the burden of system administrator of routine *root cause analysis and diagnosis* (RCAD) tasks (cf. Chapter 6) when they manage numerous resources (i.e., self-managed and atomic resources) in a heterogeneous and real-time environment.

A generic use-case-unit EPN and a systematic scenario for designing a use-case-unit EPN: Conforming to ACRA, EPACRA includes an innovative architecture—a generic use-case-unit EPN—for RCAD in the middle layer of the reference model. Each use-case-unit EPN reflects our automation approach, including identification of events from use cases and classifying those events into six event types. Chapter 6 introduces a real world RCAD scenario *Use Case One* as provided by CA. Chapter 8 defined individual EPAs to process those different event types identified from Use Case One. It also

depicted a specific use-case-unit EPN—the EPN for Use Case One at architecture diagram level.

A case study for adaptive RCAD: Chapter 11 introduced a prototype of a Web application intrusion detection tool to demonstrate the autonomic mechanism of the RCAD process. Specifically, this tool recognizes two types of malicious attacks on web application systems, and then takes action to prevent those intrusion attempts. This case study validates both the chain-of-monitors pattern and the net-casting model. It also validates our use-case-unit EPN approach, including the generality of the use-case-unit EPN design scenario and the use-case-unit EPN, as an innovative approach to accomplish an RCAD workflow. The implementation of the intrusion detection system validated the feasibility of EPN approach with Esper. This case study might also be beneficial in the future for other RCAD projects and researchers who have similar interests. Thus, the prototype itself can be considered as a contribution of this research.

12.2 Future Work

This section identifies several possible future researches as follows.

Validation of EPACRA: EPACRA is a variant of the ACRA model which strives to integrate CEP technology into autonomic computing systems to aid RCAD processes. Such an effort sets the first step toward the goal of creating a new adaptive RCAD architecture. As a new architecture model, EPACRA must be implemented for various scenarios and be improved and validated iteratively over long periods of time. My research accounts only for the initial steps of such a research process which consists of related model investigation, model creation and model validation. Besides the case study of the intrusion detection system, more empirical studies are needed in the future. These empirical studies, either case studies or user studies, will further elaborate model details and validate the model under specific circumstances, particularly in real-world settings and with large amounts of event data.

Elaborating autonomic computing architecture patterns and the net-casting information seeking model: Both the autonomic computing architecture patterns and the net-casting information seeking model presented in this dissertation are based on the literature, including our previous publications, and preliminary observations. We point the way to reveal (1) the inherent relationships among the different autonomic computing architecture patterns, and (2) the interactions among different autonomic elements in each individual pattern. We expect that the addition of architecture patterns and net-casting model related applications will improve these patterns and model by providing more specific details. On the other hand, advocating the application of these patterns and the model impose equal importance to articulate them in the sense of making them widely recognized as a real "pattern" or "model" .

Developing RCAD tools: Designing and implementing a research tool with few flaws often takes a fairly long period of time. The following case studies or user studies based on that tool conducted in a real setting probably takes even more research and work. Once a fair amount of use cases are ready and EPNs of those use-case units are implemented, the adaptive RCAD could be constructed dynamically to confirm its feasibility and adaptability (our current prototype of intrusion detection system can only modify rules off-line). For example selecting frequently-executed use cases and converting them into EPNs is one example of adaptive RCAD mechanism.

Integrated with CA products: Our adaptive RCAD tool should be realized with CA Inc. methods and tools. Once the RCAD tool becomes mature, legacy and new RCAD tools can be intergraded under one instigative workflow in the domain of CA's *system management* and *service availability management* (SAM). As a "emergency room" like tools, the adaptive RCAD tool is able to relieve the system administrator's burden of routinely performing RCAD tasks in a heterogeneous environment by generating symptom related hypotheses and gathering additional evidences to further substantiate or deny them automatically. It could work with other "regular clinics" like computer-aided RCAD tools such as the one built by Kontogiannis and Wong (cf. Figure 7-3). Those tools are more suitable for diagnosing infrequent or unidentified failures (symptoms). CA's

Catalyst, as an integration platform based on Web services and *service oriented architecture* (SOA), provides another opportunity to implement the RCAD tool based on Web services (e.g., for symptom collection and event passing). Such a tool can perform RCAD tasks upon applications that shared the same platform of CA's Catalyst.

Bibliography

[AAGK04] M. K. Agarwal, K. Appleby, M. Gupta, G. Kar, A. Neogi and A. Sailer, "Problem Determination Using Dependency Graphs and Run-Time Behavior Models," In: Proceedings 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2004), pp. 171-182, Davis, CA Springer, 2004.

[ABW03] A. Arasu, S. Babu and J. Widom, "CQL A Language for Continuous Queries Over Streams and Relations," In: *Proceedings 9th international Conference on Data Base Programming Languages (DBPL 2003)*, pp.1-19, Potsdam, Germany, 2003.

[ACTK05] IBM Corporation, "Autonomic Computing Toolkit User's Guide (3rd Edition)," http://www.ibm.com/developerworks/autonomic/sublink5.html, 2005.

[AdEt04] A. Adi and O. Etzion, "AMIT—The Situation Manager," *The International Journal on Very Large Data Bases* 13(2):177–203, 2004.

[AESW08] R. v. Ammon, C. Emmersberger, F. Springer and C. Wolff, "Event-Driven Business Process Management taking the example of DHL", FIS 2008 / 1st International Workshop on Complex Event Processing for Future Internet—Realizing Reactive Future Internet, Vienna, Austria, http://www.cittonline.de/downloads/FIS08_AmmonSpringer.pps, last accessed 2010.

[AFFG01] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D.P. Pazel, J. Pershing and B. Rochwerger, "Oceano-SLA Based Management of a Computing Utility," In: *Proceedings IFIP/IEEE International Symposium on Integrated Network Management*, pp. 855-868, Seattle, WA, USA, 2001.

[Aler10] Aleri Inc., "Complex Event Processing—Aleri," http://www.aleri.com, last accessed 2010.

[ANA10] ANA Project, "ANA—autonomic network architecture," http://www.ana-project.org/, last accessed 2010.

[AnFa00] B. Andersen and T. Fagerhaug, *Root Cause Analysis: Simplified Tools and Techniques*, ASQ Quality Press, Milwaukee, 2000.

[Apts10] AptSoft Corporation, "Internet Business Technology—Aptsoft," http://www.aptsoft.net, last accessed 2010.

[Arca97] J. S. Arcaro, *TQM Facilitator's Guide*, St. Lucie Press, Boca Raton, Florida, 1997.

[BaCa90] Y. Barlas and S. Carpenter, "Philosophical Roots of Model Validation: Two Paradigms," *System Dynamics Review* 6(2):148–166, 1990.

[Bat89] M. J. Bates, "The Design of Browsing and Berrypicking Techniques for the Online Search Interface," *Online Review* 13(5):407–424, 1989.

[BBDL10] M. Baskey, A. Black, M. Drescher, P. Lipton, Y. Matsumoto, M. Perazolo, A. Salahshour, D. Snelling and J. Vaught, "Symptoms Framework White Paper V 1.0", CA, IBM and Fujitsu, 2010.

[BBDL09] M. Baskey, A. Black, M. Drescher, P. Lipton, Y. Matsumoto, M. Perazolo, A. Salahshour, D. Snelling and J. Vaught, "Symptoms Autonomic Framework Specification V 1.0", CA, IBM and Fujitsu, 2009.

[BDIM04] P. Barham, A. Donnelly, R. Isaacs and R. Mortier, "Using Magpie for Request Extraction and Workload Modeling," In: *Proceedings 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, San Francisco, CA, USA, 2004.

[BDS07] S. Basu, J. Dunagan and G. Smith, "Why Did My PC Suddenly Slow Down?" In: *Proceedings 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques*, Article No.4, Cambridge, MA, USA, 2007.

[BKJJ98] R. P. Bonasso, R. Kerr, K. Jenks and G. Johnson, "Using the 3T Architecture for Tracking Shuttle RMS Procedures," In: *Proceedings IEEE International Joint Symposia on Intelligence and Systems*, pp.180-187, Rockville, MD, USA, 1998.

[BKK01] A. Brown, G. Kar and A. Keller, "An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in Distributed Environment," In: *Proceedings IFIP/IEEE International Symposium on Integrated Network Management*, pp. 377-390. Los Alamitos, USA, 2001.

[BLMY08] S. M. Belknap, H. Moore, S. A. Lanzotti, P. R. Yarnold, M. Getz, D. L. Deitrick, A. Peterson, J. Akeson, T. Maurer, R. C. Soltysik, G. A. Storm and I. Brooks, "Application of software design principles and debugging methods to an analgesia prescription reduces risk of severe injury from medical use of opioids," *Clinical Pharmacology & Therapeutics* 84 (3):385–92, 2008.

[Bon07] J. van Bon, *IT Service Management Based on ITIL V3: A Pocket Guide*, Volume 3, Van Haren Publishing, 2007.

[BoSu08] R. P. Jagadeesh Chandra Bose and U. Suresh, "Root Cause Analysis using Sequence Alignment and Latent Semantic Indexing," In: *Proceedings 19th Australian Conference on Software Engineering (ASWEC 2008)*, pp.367-376, Perth, Western Australia, 2008.

[BRMO03] M. Brodie, I. Rish, S. Ma and N. Odintsova, "Active Probing Strategies for Problem Diagnosis in Distributed Systems," In: *Proceedings 18th International Joint Conference on Artificial Intelligence*, pp. 1337-1338, Acapulco, Mexico, 2003.

[Broo86] R. A. Brooks, "A Robust Layered Control System for a Mobile Robot," IEEE Journal on Robotics and Automation RA-2(1):14–23, March 1986.

[BrRi94] M. Brassard and D. Ritter, *The Memory Jogger II: A Pocket Guide of Tools for Continuous Improvement and Effective Planning*, GOAL/QPC, Salem, NH, USA, 1994.

[BRS02] M. Brodie, I. Rish and Sheng Ma, "Intelligent Probing: a Cost-Effective Approach to Fault Diagnosis in Computer Networks," *IBM Systems Journal* 41(3):372–385, 2002.

[CA10] CA Inc., "CA Wily Introscope," http://www.ca.com/us/application-management.aspx, last accessed 2010.

[Calh95] C. Calhoun, *Critical Social Theory: Culture, History, and the Challenge of Difference*. Blackwell Publishers Inc.,1995.

[CASC2010] CASCADAS project, "CASCADAS," http://acetoolkit.sourceforge.net/cascadas/index.php, last accessed 2010.

[CCF04] G. Candea, J. Cutler and A. Fox, "Improving Availability with Recursive Microreboots: A Soft-State System Case Study," *Performance Evaluation* 56(1-4):213–248, 2004.

[CDK01] G. Coulouris, J. Dollimore and T. Kindberg, *Distributed Systems: Concepts and Design (3rd Edition),* Addison-Wesley Publishers Ltd., 2001.

[ChAd08] S. Chakravarthy and R. Adaikkalavan, "Events and Streams: Harnessing and Unleashing Their Synergy!," In: *Proceedings 2nd IEEE/ACM International Conference on Distributed Event-Based Systems(DEBS 2008),* pp. 1-12, Rome, Italy, 2008.

[ChCh03] H. Chan and T. C. Chieu, "An Approach to Monitor Application Status for Self-Managing (Autonomic) Systems," In: *Proceedings 18th ACM Annual SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003),* pp. 312-313, Anaheim, USA, Oct. 2003.

[ChMi94] S. Chakravarthy and D. Mishra, "Snoop: An Expressive Event Specification Language for Active Databases," *Data & Knowledge Engineering* 14(1):1–26, 1994.

[CKFF02] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox and E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services," In: *Proceedings IEEE 2002 International Conference on Dependable Systems and Networks (DSN 2002),* pp. 595-604, Bethesda, MD, USA, 2002.

[CKKF06] G. Candea, E. Kiciman, S. Kawamoto and A. Fox, "Autonomous Recovery in Componentized Internet Applications," *Cluster Computing* 9(1):175–190, 2006.

[CNYM99] L. Chung, B. A. Nixon, E. Yu and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers, 1999.

[CoSp98] J. F. Cox III and M. S. Spencer, *The Constraints Management Handbook*, St. Lucie Press, Boca Raton, FL, USA, 1998.

[Cres02] J. W. Creswell, *Research Design: Qualitative, Quantitative, and Mixed Method*, SAGE Publications, 2002.

[CRW01] A. Carzaniga, D. S. Rosenblum and A. L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Transactions on Computer Systems* 19(3):332–383, 2001.

[CZLJ04] M. Chen, A. X. Zheng, J. Lloyd, M. Jordan and E. Brewer, "Failure Diagnosis Using Decision Tree," In: *Proceedings 1st IEEE International Conference on Autonomic Computing (ICAC 2004)*, New York, USA, 2004.

[DBM09] S. Duan, S. Babu and K. Munagala, "Fa: A System for Automating Failure Diagnosis," In: *Proceedings 2009 IEEE International Conference on Data Engineering (ICDE 2009)*, pp. 1012-1023, Shanghai, China 2009.

[DDFD06] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt and F. Zambonelli, "A Survey of Autonomic Communications," *ACM Transaction on Autonomic and Adaptive Systems* 1(2):223–259, 2006.

[DDKM08] D. Dawson, R. Desmarais, H. M. Kienle and H. A. Müller, "Monitoring in Adaptive Systems Using Reflection," In: *ICSE 2008 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008)*, pp. 81-88, Leipzig, Germany, 2008.

[Dekk07] P. Dekkers, *Complex Event Processing*, Master Thesis Computer Science, Radboud University Nijmegen, 2007.

[Denn07] P. J. Denning, "Computing is a Natural Science," *Communications of the ACM* 50(7):13–18, 2007.

[DHRS08] X. Ding, H. Huang, Y. Ruan, A. Shaikh and X. Zhang, "Automatic Software Fault Diagnosis by Exploiting Application Signatures," In: *Proceedings 22nd Large Installation System Administration Conference (USENIX LISA 2008)*, San Diego, CA, USA, 2008.

[DLP01] W. Dickinson, D. Leon and A. Podgurski, "Finding Failures by Cluster Analysis of Execution Profiles," In: *Proceedings 23rd International Conference on Software Engineering (ICSE 2001)*, Toronto, Ontario, Canada, 2001.

[DMTF10A] Distributed Management Task Force Inc., "Web-Based Enterprise Management Standard," http://www.dmtf.org/standards/wbem/, last accessed 2010.

[DMTF10B] Distributed Management Task Force Inc., "Common Information Model Standard," http://www.dmtf.org/standards/cim/, last accessed 2010.

[DoCa04] J. Dowling and V. Cahill, "Self-Managed Decentralised Systems Using K Components and Collaborative Reinforcement Learning," In: *Proceedings ACM Workshop on Self-Managed Systems*, pp. 39-43, 2004.

[Dogg05] M. Doggett, "Root Cause Analysis: A Framework for Tool Selection," *Quality Management Journal* 12(4):34–45, 2005.

[Dowl04] J. Dowling, *The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems*, Ph.D. thesis, Department of Computer Science, Trinity College Dublin, 2004.

[DrSt05] D. Dresner and P. Stone, "Multiagent Traffic Management: An Improved Intersection Control Mechanism," In: *Proceedings 4th International Joint Conference on Autonomous Agents and Multiagent Systems(AAMAS 2005)*, pp. 471–477, Utrecht, The Netherlands, 2005.

[Dsso10] R. Dssouli, "Validation vs. Verification," COEN 345 lecture slides, http://users.encs.concordia.ca/~dssouli/Chap1-2pp.pdf, last accessed 2010.

[Duga00] J. B. Dugan, "Galileo: A Tool for Dynamic Fault Tree Analysis," In: *Proceedings 11th International Conference on Computer Performance Evaluation, Modeling Techniques and Tools (TOOLS 2000)*, pp. 328-331, Schaumburg, IL, USA, 2000.

[Dunk09] J. Dunkel, "On Complex Event Processing for Sensor Networks," In: *Proceedings 9th IEEE International Symposium on Autonomous Decentralized Systems (ISADS 2009)*, pp. 249–255, Athens, Greece, 2009.

[ElTa01] C. Elsaesser and M. C. Tanner, "Automated Diagnosis for Computer Forensics," Technical Report, The MITRE Corporation, 2001.

[EMC09] EMC, "Automating Root-Cause Analysis: EMC Inoix Codebook Correlation Technology vs. Rules-based Analysis," white paper, http://www.emc.com, 2009.

[Espe10] EsperTech, Esper Reference Documentation version 3.4.0, http://esper.codehaus.org/esper/documentation/documentation.html, 2010.

[East07]S. M. Easterbrook, "Empirical Research Methods in Requirements Engineering," Tutorial in: 15th IEEE International Conference on Requirements Engineering (RE 2007), New Delhi, India, October 2007, http://www.cs.toronto.edu/~sme/presentations/re07tutorial-vPrint.pdf, last accessed 2010.

[ESSD07] S. M. Easterbrook, J. Singer, M. Storey and D. Damian, "Selecting Empirical Methods for Software Engineering Research," In F. Shull and J. Singer (eds): *Guide to Advanced Empirical Software Engineering*, Springer Verlag, 2007.

[EuGu01] P. T. Eugster and R. Guerraoui, "Content-Based Publish/Subscribe with Strucutural Reflection," In: *Proceeding 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS 2001)*, San Antonio, Texas, USA, 2001.

[FHSE06] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund and E. Gjorven, "Using Architecture Models for Runtime Adaptability," *IEEE Software*, pp. 62-70, 2006.

[Firb89] R. J. Firby, *Adaptive Execution in Dynamic Domains*, Ph.D. Thesis, Computer Science Department, Yale University, 1989.

[Gane07] A. Ganek, "Overview of Autonomic Computing: Origins, Evolution, Direction," In M. Parashar and S.Hariri (eds): *Autonomic Computing Concepts, Infrastructure, and Applications*, CRC Press, 2007.

[GaSc02] D. Garlan and B. Schmerl, "Model-Based Adaptation for Self-Healing Systems," In: *Proceedings Workshop on Self-healing Systems*, pp. 27-32, 2002.

[Gat91] E. Gat, *Reliable Goal-directed Reactive Control for Real-world Autonomous Mobile Robots*, Ph.D. Thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1991.

[Gat97] E. Gat, "Three-layer Architectures," *Artificial Intelligence and Mobile Robots*, MIT/AAAI Press, 1997.

[GCHS04] D. Garlan, S. W. Cheng, A. C. Huang, B. Schmerl and P. Steenkiste, "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," *IEEE Computer* 37(10):46–54, 2004.

[GDZK10] P. Gupta, R. Desmarais, Q. Zhu, H. M. Kienle and H. A. Müller, "Static and Dynamic Architectural Views to Characterize Self-managing Systems," *Rigi Group Technical Report* (unpublished), Dept. of Computer Science, University of Victoria, 2010.

[Glas10] Glassbox Corp., "Glassbox—Just Tell You What Broke," http://www.glassbox.com/glassbox/Project.html, last accessed 2010.

[Gold90] E. M. Goldratt, *What is this Thing Called Theory of Constraints and how should it be Implemented?* North River Press, New York, 1990.

[Gold94] E. M. Goldratt, *It's Not Luck*, North River Press, Great Barrington, Massachusetts, 1994.

[GrRo08] P. Graubmann and M. Roshchin, "Adaptive SOA Infrastructure Based on Variability Management," *Book 4 Advanced Studies in Software and Knowledge Engineering*, Institute of Information Theories and Applications FOI ITHEA, pp70-75, 2008.

[HaPe10] M. J. Hawthorne and D. E. Perry, "Architectural Styles for Adaptable Self-healing Dependable Systems," http://users.ece.utexas.edu/~perry/work/papers/MH-05-Styles.pdf, last accessed 2010.

[Hase10] K. Haselden (Microsoft), "SQLIS: What is Validation?" http://www.sqlis.com/sqlis/post/What-is-Validation.aspx, last accessed 2010.

[Hear99] M. A. Hearst, "User Interface and Visualization," In R. Baeza-Yates and B. Ribeiro-Neto (eds): *Modern Information Retrieval*, Addison Wesley, 1999.

[Hern09] M. E. Hernandez, *Visualization for Seeing and Comparing Clinical Trials*, Ph.D. Thesis, Department of Computer Science, University of Victoria, 2010.

[HGSD07] G. Hermosillo, R. Gomez, L. Seinturier and L. Duchien, "AproSec: an Aspect for Programming Secure Web Applications," In: *Proceedings 2nd IEEE International Conference on Availability, Reliability and Security (ARES 2007)*, pp. 1026-1033, Vienna, Austria, 2007.

[HiSt06] M. G. Hinchey and R. Sterritt, "Self-managing Software," *IEEE Computer* 39(2):107–109, 2006.

[HJRS07] H. Huang, R. Jennings III, Y. P. Ruan, R. Sahoo, S. Sahu and A. Shaikh, "PDA: A Tool for Automated Problem Determination," In: *Proceedings USENIX Large Installation System Administration Conference (LISA07)*, pp. 153-166, Dallas, TX, USA, 2007.

[Horn01] P. Horn, "Autonomic Computing: IBM's Perspective on the State of Information Technology," Technical Report, IBM Corporation, http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf, 2001.

[HP10] Hewlett-Packard Co., "Adaptive enterprise solutions at the HP Solution Centers," http://www.hp.com/products1/solutioncenters/adaptive_enterprise/index.html, last accessed 2010.

[HuMc08] M. C. Huebscher and J. A. McCann, "A Survey of Autonomic Computing—Degrees, Models, and Applications," *ACM Computing Surveys* 40(3):7:1–28, 2008.

[IBM05] IBM Corporation, *Autonomic Computing Toolkit: Developers guide*, Technical Report SC30-4083-03, 2005.

[IBM06] IBM Corporation, "An Architectural Blueprint for Autonomic Computing, White Paper (4th Edition)," 2006, http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf

[IBM10A] IBM Corporation, "Common Base Event," http://www.ibm.com/developerworks/library/specification/ws-cbe/, last accessed 2010.

[IBM10B] IBM Corporation, "Autonomic Computing Toolkit User's Guide (3rd Edition)," http://www.ibm.com/developerworks/autonomic/sublink5.html, last accessed 2010.

[IBM10C] IBM Corporation, "Making ITIL Actionable in an IT Service Management Environment," http://www-306.ibm.com/software/tivoli/resource-center/overall/eb-itil-it-serv-mgmt.jsp, last accessed 2010.

[IETF10] Internet Engineering Task Force, "Simple Network Management Protocol—SNMP," http://www.ietf.org/, last accessed 2010.

[Ishi82] K. Ishikawa, *Guide to Quality Control* (2nd edition), Asian Productivity Organization, Tokyo, 1982.

[Iwan02] G. Iwan; "History-based Diagnosis Templates in the Framework of the Situation Calculus." *AI Communications* 15(1):31–45, Special issue on KI-2001, 2002.

[Jade10] Jade Organization, "Jade: a framework for autonomic management," http://sardes.inrialpes.fr/~depalma/JADEWEB/jade.html, last accessed 2010.

[Java10] Java Community Process, "Java Management Extensions—JMX," http://jcp.org/en/jsr/detail?id=3, last accessed 2010.

[JBLN06] M. Jelasity, O. Babaoglu, R. Laddaga, R. Nagpal, F. Zambonelli, E. G. Sirer, H. Chaouchi and M. Smirnov, "Interdisciplinary Research: Roles for Self-Organization," *IEEE Intelligent Systems* 21(2):50–58, 2006.

[JHS01] J. A. Jones, M. J. Harrold and J. T. Stasko, "Visualization for Fault Localization," In: *IEEE/ACM International Conference on Software Engineering (ICSE 2001) Workshop on Software Visualization*, Toronto, Ontario, Canada, 2001.

[JHS02] J. A. Jones, M. J. Harrold and J. T. Stasko, "Visualization of Test Information to Assist Fault Localization," In: *Proceedings IEEE/ACM International Conference on Software Engineering (ICSE 2002)*, Orlando, FL, USA, 2002.

[Josu07] N. M. Josuttis, *SOA in Practice: The Art of Distributed System Design*, O'Reilly, 2007.

[Juli03] K. Julisch, *Using Root Cause Analysis to Handle Intrusion Detection Alarms*, Ph.D. Thesis, University of Dortmund, Germany, 2003.

[KaGe97] S. Katker and K. Geihs, "A Generic Model for Fault Isolation in Integrated Management Systems," *Journal of Network and Systems Management* 5(2):109–130, 1997.

[KBE99] M. M. Kokar, K. Baclawski and Y. A. Eracar, "Control Theory Based Foundations of Self-Controlling Software," *IEEE Intelligent Systems* 14(3):37–45, 1999.

[KCCE07] V. Kumar, B. F. Cooper, Z. Cai, G. Eisenhauer and K. Schwan, "Middleware for Enterprise Scale Data Stream Management Using Utility-Driven Self-Adaptive Information Flows," *Cluster Computing* 10(4):443–455, 2007.

[KeCh03] J. O. Kephart and D. Chess, "The Vision of Autonomic Computing," *IEEE Computer* 36(1): 41–50, 2003.

[KiFo05] E. Kiciman and A. Fox, "Detecting Application-Level Failures in Component-based Internet Services," *IEEE Transactions on Neural Networks* 16(5):1027-1041, 2005.

[KiHa07] B. Kim and S. Hariri, "Anomaly-based Fault Detection System in Distributed System," In: *Proceedings 5th* ACIS *International Conference on Software Engineering Research, Management & Applications (SERA 2007)*, pp. 782-789, Busan, Korea, 2007.

[KlMy99] H. K. Klein and M. D. Myers, "A Set of Principles for Conducting and Evaluating Interpretive Field Studies in Information Systems," *MIS Quarterly* 23(1):67–93, 1999.

[KMR92] J. de. Kleer, A. K. Mackworth and R. Reiter; "Characterizing Diagnosis and Systems," In W. Hamscher, L. Console and J. de Kleer (eds): *Readings in Model-Based Diagnosis*, pp. 54-65, Morgan Kaufmann Publishers Inc., 1992.

[KPGV03] G. E. Kaiser, J. Parekh, P. Gross and G. Valetto, "Kinesthetics Extreme: An External Infrastructure for Monitoring Distributed Legacy Systems," In: *Proceedings Active Middleware Services*, pp. 22-31, 2003.

[KrMa07] J. Kramer and J. Magee, "Self-managed Systems: An Architectural Challenge," In: *Proceedings IEEE International Conference on Software Engineering, Future of Software Engineering (FoSE 2007)*, pp. 259-268, 2007.

[KrMa96] J. Kramer and J. Magee, "Dynamic Structure in Software Architectures," *ACM SIGSOFT Software Engineering Notes* 21(6):3–14, 1996.

[Ladd06] R. Laddaga, "Self Adaptive Software Problems and Projects," In: *Proceedings IEEE Workshop on Software Evolvability*, pp. 3-10, 2006.

[LaLa02] R. J. Latino and K. Latino, *Root Cause Analysis: Improving Performance for Bottom Line Results,* CRC Press, LLC, 2002.

[Laws10] L. Lawson, IT Business Edge, "The Ins and Outs of Complex Event Processing Solutions," http://www.itbusinessedge.com/cm/community/features/interviews/blog/the-ins-and-outs-of-complex-event-processing-solutions/?cs=22472, last accessed 2010.

[LBSZ98] J. P. Loyall, D. E. Bakken, R. E. Schantz, J. A. Zinky, D. A. Karr, R. Vanegas, and K. R. Anderson, "Qos Aspect Languages and Their Runtime Integration," In: *Proceedings International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pp. 303-318, 1998.

[LeFi02] R. de Lemos and J. L. Fiadeiro "An Architectural Support for Self-Adaptive Software for Treating Faults," In: *Proceedings Workshop on Self-healing Systems*, pp.39-42, 2002.

[Leve04] N. Leveson, "A New Accident Model for Engineering Safer Systems," *Safety Science* 42(4): 237–270, 2004.

[Lin10] L. Lin, *Aspect-Based Intrusion Detection Framework for Java Web Applications*, Master Thesis, Department of Computer Science, University of Victoria, July, 2010.

[LJP01] A.D. Livingston, G. Jackson, and K. Priestley, *Root Causes Analysis: Literature Review*, HSE Books, 2001.

[LLMY05] A. Lapouchnian, S. Liaskos, J. Mylopoulos and Y. Yu, "Towards Requirements-Driven Autonomic Systems Design," In: *Proceedings Workshop on Design and Evolution of Autonomic Application Software*, pp.1-7, 2005.

[LPH04] H. Liu, M. Parashar and S. Hariri, "A Component-Based Programming Model for Autonomic Applications," In: *Proceedings International Conference on Autonomic Computing*, pp.10-17, 2004.

[LPS00] M. Leszak, D. E. Perry and D. Stoll, "A Case Study in Root Cause Defect Analysis," In: *Proceedings 22nd international conference on Software engineering (ICSE2000)*, pp.428-437, Limerick, Ireland, 2000.

[Luck05] D. Luckham, *The Power of Events*, Addison Wesley, 2005.

[LWMW04] N. Lao, J. R. Wen, W. Y. Ma and Y. M. Wang, "Combining High Level Symptom Descriptions and Low Level State Information for Configuration Fault Diagnosis," In: *Proceedings 18th Conference on Systems Administration (LISA 2004)*, pp.151-158, Atlanta, USA, 2004.

[LYLM06] A. Lapouchnian, Y. Yu, S. Liaskos and J. Mylopoulos, "Requirements-Driven Design of Autonomic Application Software," In: *Proceedings IBM/ACM 2006 conference of the Center for Advanced Studies on Collaborative Research (CASCON 2006)*, pp. 80-93, Toronto, Ontario, Canada, 2006.

[MABB08] Y. Magid, A. Adi, M. Barnea, D. Botzer and E. Rabinovich, "Application Generation Framework for Real-Time Complex Event Processing," In: *Proceedings 32nd IEEE Annual International Computer Software and Applications Conference (COMPSAC 2008)*, pp. 1162-1167, Turku, Finland, 2008.

[Mar97] G. Marchionini, *Information Seeking in Electronic Environments, Series: Cambridge Series on Human-Computer Interaction, No. 9*, Cambridge University Press, 1997.

[MBM08] M. R. N. Mendes, P. Bizarro and P. Marques, "A Framework for Performance Evaluation of Complex Event Processing Systems," In: *Proceedings 2nd IEEE/ACM International Conference on Distributed Event-Based Systems (DEBS 2008)*, pp. 313-316, Rome, Italy, 2008.

[Mcll98] S. A. Mcllraith, "Explanatory Diagnosis: Conjecturing Actions to Explain Observations," In: *Proceedings ACM Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pp. 167-177, Trento, Italy, 1998.

[McMa10] K. McManus, "How Do You Find Root Causes?" http://www.greatsystems.com/rootcause.htm, last accessed 2010.

[Mena97] L. Menand, *Pragmatism: A Reader*, Vintage Press, 1997.

[Mich06] B. M. Michelson, "Event-Driven Architecture Overview," Patricia Seybold Group, 2006.

[Micr10] Microsoft Corporation, "Microsoft Dynamic Systems Initiative Overview," http://www.microsoft.com/servers/home.mspx, last accessed 2010.

[MiMi08] A. V. Mirgorodskiy and B. P. Miller, "Diagnosing Distributed Systems with Self-propelled Instrumentation," In: *Proceedings 9th ACM/IFIP/USENIX International Conference on Middleware (Middleware 2008)*, pp. 82-103, Leuven, Belgium, 2008.

[Mizu88] S. Mizuno, *Management for Quality Improvement: The Seven New QC Tools*, Productivity Press, Cambridge, 1988.

[MoKi06] R. Mortier and E. Kiciman, "Autonomic Network Management: Some Pragmatic Considerations," In: *Proceedings 2006 SIGCOMM Workshop on Internet Network Management*, pp. 89-93, Pisa, Italy, 2006.

[MPS08] H. A. Müller, M. Pezzè and M. Shaw, "Visibility of Control in Adaptive System," In: *Proceedings ACM/IEEE International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008)*, Leipzig, Germany, May 2008.

[MSN07] J. Mickens, M. Szummer and D. Narayanan, "Snitch: Interactive Decision Trees For Troubleshooting Misconfigurations," In: *Proceedings 2007 Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML 2007)*, Cambridge, MA, USA, 2007.

[MuGl05] A. Mukhija and M. Glinz, "Runtime Adaptation of Applications through Dynamic Recomposition of Components," In: *Proceedings International Conference on Architecture of Computing Systems*, pp. 124-138, 2005.

[Musa04] J. D. Musa, Software reliability engineering: more reliable software, faster and cheaper (2nd edition), AuthorHouse, 2004.

[Nelm07] C. R. Nelms, "The Problem with Root Cause Analysis," In: *Proceedings Joint 8th IEEE Conference on Human Factors and Power Plants and 13th Annual Conference on Human Performance / Root Cause / Trending / Operating Experience / Self Assessment (Joint 8th IEEE HFPP / 13th HPRCT)*, pp. 253-258, Monterey, CA, USA, 2007.

[NeMü07] S. Neti and H. A. Müller, "Quality Criteria and an Analysis Framework for Self-Healing Systems," In: *Proceedings ACM/IEEE International Workshop on Software Engineering for Adaptive and Self-Managing Systems(SEAMS 2007)*, pp. 39-48, Minneapolis, Minnesota, USA, 2007.

[NHDM09] S. Nadi, R. Holt, I. Davis and S. Mankovskii, "DRACA: Decision Support for Root Cause Analysis and Change Impact Analysis for CMDBs," In: *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research (CASCON 2009)*, pp. 1-11, Toronto, Ontario, Canada, 2009.

[NKHL08] Y. Al-Nashif, A. A. Kumar, S. Hariri, Y. Luo, F. Szidarovsky and G. Qu, "Multi-Level Intrusion Detection System (ML-IDS)," In: *Proceedings International Conference on Autonomic Computing*, pp. 131-140, Chicago, IL, USA, 2008.

[NRI09] NRI, *MORT User's Manual for Use with the Management Oversight & Risk Tree Analytical Logic Diagram*, Second Edition, The Noordwijk Risk Initiative Foundation, ISBN 978-90-77284-08-7, 2009.

[ODJe93] V. L. O'Day and R. Jeffries, "Orienteering in an Information Landscape: How Information Seekers Get from Here to There," In: *Proceedings INTERACT '93 and CHI*

'93 conference on Human Factors in Computing Systems, pp. 438-445, Amsterdam, The Netherlands, 1993.

[OGTH99] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum and A. L.Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems* 14(3):54–62, 1999.

[Open10] The Open Group, "Application Response Measurement—ARM," http://www.opengroup.org/tech/management/arm/, last accessed 2010.

[PaBu88] M. Paradies and D. Busch, "Root Cause Analysis at Savannah River Plant," In: *Proceedings of the IEEE Conference on Human Factors and Power Plants*, pp. 479–483, Monterey, CA, USA, 1988.

[PaHa05] M. Parashar and S. Hariri, "Autonomic Computing: An Overview," *Hot Topics, Lecture Notes in Computer Science* 3566:247–259, 2005.

[PBH02] P. R. Pietzuch and J. M. Bacon, "Hermes: A Distributed Event-Based Middleware Architecture," IN: *Proceedings 22nd IEEE International Conference on Distributed Computing Systems (ICDCSW 2002)*, pp. 611-618, Vienna, Austria, 2002.

[PEBA00] K. Pedersen, J. Emblemsvåg, R. Bailey, J. K. Allen and F. Mistree, "The Validation Square: Validating Design Methods," *ASME Design Theory and Methodology Conference*, Paper No. DETC2000/DTM-14579, Baltimore, MD, 2000.

[PGN07] S. Pertet, R. Gandhi and P. Narasimhan, "Fingerpointing Correlated Failures in Replicated Systems," In: *Proceedings 2nd USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*, Article No. 9, Cambridge, MA, 2007.

[Piro07] P. Pirolli, *Information Foraging Theory: Adaptive Interaction with Information*, Oxford University Press, 2007.

[PPL07] S. Piao, J. Park and E. Lee, "Root Cause Analysis and Proactive Problem Prediction for Self-Healing," In: *Proceedings 1st IEEE International Conference on Convergence Information Technology (ICCIT 2007)*, pp. 2085-2090, Gyeongju, Korea, 2007.

[Prog10] Progress Software Co., "Apama Capital Markets and Complex Event Processing ," http://web.progress.com/en/apama, last accessed 2010.

[PSE04] D. E. Perry, S. E. Sim and S. M. Easterbrook, "Case Studies for Software Engineers," In: *Proceedings IEEE/ACM 26th International Conference on Software Engineering (ICSE 2004)*, pp. 736-738, Edinburgh, Scotland, 2004.

[RBOM04] I. Rish, M. Brodie, N. Odintsova, S. Ma and G. Grabarnik, "Real-time Problem Determination in Distributed Systems using Active Probing," In: *Proceedings 2004 IEEE/IFIP Network Operations and Management Symposium (NOMS 2004)*, Seoul, Korea, 2004.

[RDRE00] I. Rouvellou, L. Degenaro, K. Rasmus, D. Ehnebuske and B. McKee, "Extending Business Objects with Business Rules," In: *Proceedings 33rd ACM International Conference on Technology of Object-Oriented Languages and Systems (OOPSLA 2000)*, pp. 238-249, St. Malo, France, 2000.

[RoLa05] P. Robertson and R. Laddaga, "Model Based Diagnosis and Contexts in Self-adaptive Software," *Lecture Notes in Computer Science Volume 3460, Self-star Properties in Complex Information Systems*, pp.112-127, Springer Verlag, 2005.

[RSS07] S. Rozsnyai, J. Schiefer and A. Schatten, "Concepts and Models for Typing Events for Event-Based Systems," In: *Proceedings 2007 Inaugural International Conference on Distributed Event-Based Systems(DEBS 2007)*, pp. 62-70, Toronto, Ontario, Canada, 2007.

[Rule10] Rulecore.com, "RuleCore-Esper-Complex Event Processing Server," http://www.rulecore.com/, last accessed 2010.

[Sale09] M. Salehie, A *Quality-Driven Approach to Enable Decision-Making in Self-Adaptive Software*, Ph.D. Thesis, University of Waterloo, 2009.

[SaMc04] S. M. Sadjadi and P. K. McKinley, "ACT: An Adaptive CORBA Template To Support Unanticipated Adaptation," In: *Proceedings International Conference on Distributed Computing Systems*, pp.74-83, 2004.

[SaTa05] M. Salehie and L. Tahvildari, "Autonomic Computing: Emerging Trends and Open Problems," In: *Proceedings Workshop on Design and Evolution of Autonomic Application Software*, pp. 82-88, 2005.

[SBC98] B. Shneiderman, D. Byrd and B. Croft, "Sorting out Searching: A User Interface Framework for Text Searches," *Communications of the ACM* 41(4):95–98, 1998.

[Schu06] W. R. Schulte, "Event Processing in Business Applications", http://complexevents.com/slides/Gartner_Schulte.ppt, 2006.

[Seve10] Seventh Framework Programme, "The Socrates Project," http://www.fp7-socrates.org/?q=node/1, last accessed 2010.

[SFHK03] G. D. M. Serugendo, N. Foukia, S. Hassas, A. Karageorgos, S. K. Mostéfaoui, O. F. Rana, M. Ulieru, P. Valckenaers and C. van Aart, "Self-Organisation: Paradigms

And Application," In: *Proceedings of Engineering Self-Organizing Application Workshop*, pp. 1-19, 2003.

[ShEt08] G. Sharon and O. Etizon, "Event-processing network model and implementation," *IBM Systems Journal* 47(2):321–334, 2008.

[ShFu94] T. Shibata and T. Fukuda, "Hierarchical Intelligent Control for Robotic Motion", IEEE Transaction on Neural Networks 5(5): 823–832, 1994.

[SMCS04] S. M. Sadjadi, P. K. McKinley, B. H. C. Cheng and R. E. K. Stirewalt, "TRAP/J:Transparent Generation of Adaptable Java Programs," *Lecture Notes in Computer Science* 3291:1243–1261, 2004.

[Sour10] Sourceforge, "ACE Autonomic Toolkit," http://sourceforge.net/projects/acetoolkit/files/, last accessed 2010.

[SPTU05] R. Sterritt, M. Parashar, H. Tianfield and R. Unland, "A Concise Introduction to Autonomic Computing," Advanced Engineering Informatics 19:181–187, Jul 2005.

[St02] R. Stemtt, "Towards Autonomic computing:Effective Event Management," In: *Proceedings of the 27th Annual IEEFINASA Software Engineering Workshop*, pp 40-47, Greenbelf, MD, USA, 2002.

[St03] R. Sterritt, "Autonomic Computing: The Natural Fusion of Soft Computing and Hard Computing," In: *Proceedings IEEE International Conference on Systems, Man and Cybernetics*, volume 5, pages 4754-4759, Washington, D.C., USA, 2003.

[Stre10] StreamBase Systems, Inc., "Complex Event Processing, Event Stream Processing, StreamBase," http://www.streambase.com, last accessed 2010.

[SuFl09] Y. Y. Su and J. Flinn, "Automatically Generating Predicates and Solutions for Configuration Troubleshooting," In: *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, USA, 2009.

[Sun10] Sun, "JVM Tool Interface—JVMTI," http://download.oracle.com/docs/cd/E17476_01/javase/1.5.0/docs/guide/jvmti/index.html, last accessed 2010

[SwDr07] J. W. Sweitzer and C. Draper, "Architecture Overview for Autonomic Computing," In M. Parashar and S. Hariri(des): *Autonomic Computing Concepts, Infrastructure, and Applications*, CRC Press, 2007.

[Tagu05] N. R. Tague, "Seven Basic Quality Tools," *The Quality Toolbox*, American Society for Quality, Quality Press, 2005.

[TBHS03] S. Tuttle, V. Batchellor, M. B. Hansen and M. Sethuraman, "Centralized Risk Management Using Tivoli Risk Manager 4.2," Technical report, IBM Tivoli Software, December 2003.

[Temp10] B. Temple. "Using Aspects to Autonomic-Enable Legacy Applications," http://www.ibm.com/developerworks/library/ac-aspects/index.html, last accessed 2010.

[TeMi06] V. Tewari and M. Milenkovic, "Standards for Autonomic Computing," Intel Technology Journal 10(4):275-284, 2006.

[Tibc10] TIBCO Software Inc., "TIBCO BusinessEvents," http://www.tibco.com/software/complex-event-processing/businessevents/default.jsp, last accessed 2010.

[Tiff02] M. Tiffany, "A Survey of Event Correlation Techniques and Related Topics," 2002.
http://www.tiffman.com/netman/netman.pdf, last accessed 2010.

[TLHX06] J. Tucek, S. Lu, C. D. Huang, S. Xanthos and Y. Y. Zhou, "Automatic On-line Failure Diagnosis at the End-User Site," In: *Proceedings of the 2nd conference on Hot Topics in System Dependability*, pp. 4-4, Seattle, WA, USA, 2006.

[VaKa03] G. Valetto and G. Kaiser, "Using Process Technology to Control and Coordinate Software Adaptation," In: *Proceedings International Conference on Software Engineering*, pp. 262-273, 2003.

[Wale02] A.Walenstein, *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*, Ph.D. Thesis, Simon Fraser University, 2002.

[WaLi04] E.J. Wagner and H. Lieberman, "Supporting User Hypotheses in Problem Diagnosis on the Web and Elsewhere," In: *Proceedings 9th ACM International Conference on Intelligent User Interfaces (IUI 2004)*, Funchal, Madeira, Portugal, 2004.

[WCG04] A. Whitaker, R. S. Cox and S. D. Gribble, "Configuration Debugging as Search: Finding the Needle in the Haystack," In: *Proceedings 6th Symposium on Operating Systems Design and Implementation*, pp.77-90, San Francisco, CA, USA, 2004.

[Weed07] D. L. Weed, "Nature and Necessity of Scientific Judgment," *Journal of Law and Policy* 15(1):135–164, 2007.

[WMYM07] Y. Wang, S. Mcllraith, Y. Yu, and J. Mylopoulos. "An Automated Approach to Monitoring and Diagnosing Requirements," In: *Proceedings 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE2007)*, pp. 293-302, Atlanta, USA, 2007.

[WSG05] J. White, D. C. Schmidt and A. S. Gokhale, "Simplifying Autonomic Enterprise Java Bean Applications via Model-Driven Development: A Case Study," In: *Proceedings International Conference on Model Driven Engineering Languages and Systems*, pp.601-615, 2005.

[YLLM08] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos and J. Leite, "From Goals to High-Variability Software Design," *Lecture Notes in Computer Science* 4994:1, 2008.

[YLWL06] C. Yuan, N. Lao, J. R. Wen, J. W. Li, Z. Zhang, Y. M. Wang and W. Y. Ma, "Automated Known Problem Diagnosis with Event Traces," In: *Proceedings ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys 2006)*, pp. 375-388, New York, NY, USA, 2006.

[ZDAM07] Q. Zhu, D. Dawson, P. Agrawal and H. A. Müller, "Leveraging Conceptual Models of Trust in Automated Systems to Promote 'Appropriate Trust' in Autonomic Systems," In: *Proceedings IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM 2007)*, pp. 473-476, Victoria, Canada, 2007.

[Zell06] A. Zeller, *Why Programs Fail*: *A Guide to Systematic Debugging*, Morgan Kaufmann Publishers, 2006.

[Zhu07] Q. Zhu, "An Experimental Platform for Root Cause Diagnosis Research," In: *Proceedings 14th IEEE Working Conference on Reverse Engineering (WCRE 2007)*, pp. 293-296, Vancouver, Canada, 2007.

[Zhu08] Q. Zhu, "Goal Trees and Fault Trees for Root Cause Analysis," In: *Proceedings IEEE International Conference on Software Maintenance (ICSM 2008)*, pp. 436-439, Beijing, China, 2008.

[ZiUn99] D. Zimmer and R. Unland, "On the semantics of complex events in active database management systems," In: *Proceedings 15th IEEE International Conference on Data Engineering (ICDE 1999)*, pp. 392–399, Sydney, Austrialia, 1999.

[ZLKM08] Q. Zhu, L. Lin, H. M. Kienle and H. A. Müller, "Characterizing Maintainability Concerns in Autonomic Element Design," In: *Proceedings IEEE International Conference on Software Maintenance (ICSM 2008)*, pp. 197-206, Beijing, China, 2008.

Appendix A: Glossary

ABAS	Attribute-Based Architectural Styles
AC	Autonomic Computing
ACRA	Autonomic Computing Reference Architecture
API	Application Programming Interface
AOP	Aspect-Oriented Programming
AspectJ	An aspect-oriented extension created at Palo Alto Research Center Incorporated for the Java programming language
Autonomic Computing Application Pattern	A set of templates for combining building blocks into specific situations commonly found in real deployments
Autonomic Computing Architecture Pattern	A set of templates for describing how autonomic elements may be integrated together to support the functional and non-functional requirements delivered by an autonomic computing system
BEPL	Business Events Processing Language
CCL	Continuous Computational Language
CED	Cause-and-effect Diagram
CEP	Complex Event Processing
CIM	Common Information Model
CMDB	Configuration Management Database
CRT	Current Reality Tree
CSS	Cross Site Scripting
Daytrading	A web-based stock trading system that handles the customer stock operations such as buying and selling stocks on-line
DMTF	Distributed Management Task Force
DOM	Document Object Model
ECA	Event Condition Action
EDA	Event-Driven Architecture
EPA	Event Processing Agent
EPACRA	Event Processing Autonomic Computing Reference Architecture
EPL	Event Processing Language
EPN	Event Processing Network
ESB	Enterprise Service Bus
Esper	A CEP engine
GPL	GNU General Public License
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
ID	(1) Interrelationship Diagram; (2) Identification
IPMI	Intelligent Platform Management Interface
IT	Information Technology

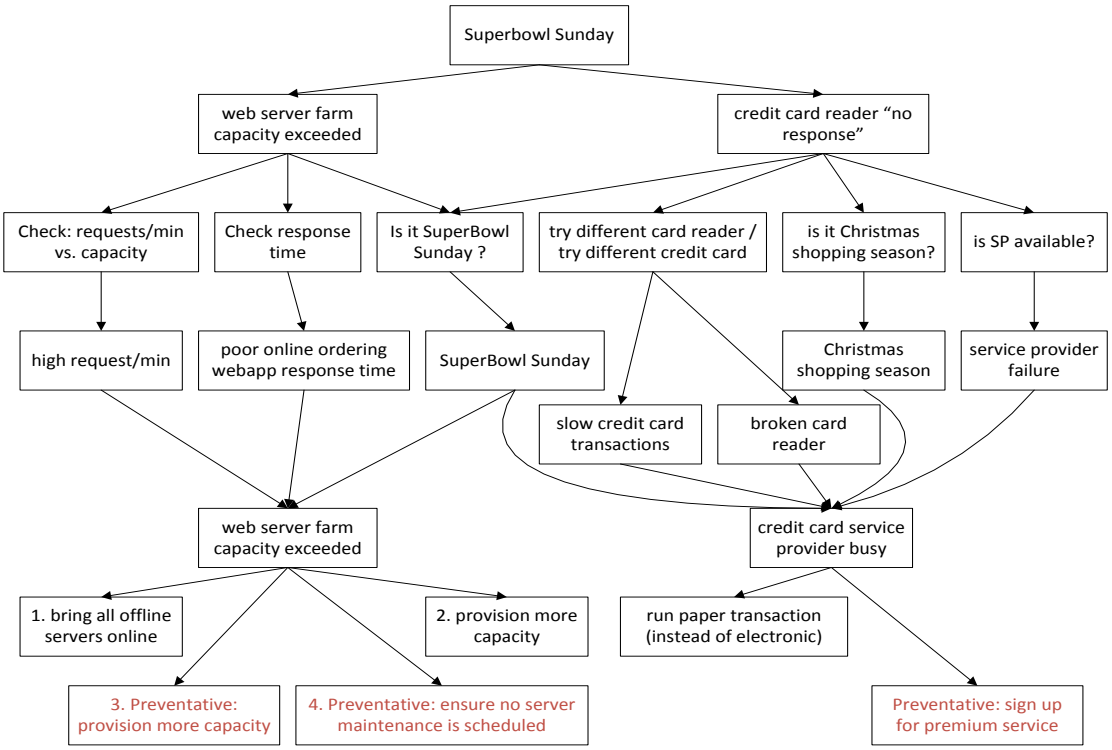
ITIL	Information Technology Infrastructure Library
JMS	Java Message Service
JMX	Java Management Extensions
MAPE	Monitor, Analyze, Plan and Execute
MIB	Management Information Base
MOF	Managed Object Format
MOWS	Management of Web Services
MUWS	Management Using Web Services
OASIS	Organization for the Advancement of Structured Information Standards
RCA	Root Cause Analysis
RCAD	Root Cause Analysis and Diagnosis
SLA	Service Level Agreement
SML	Service Modeling Language
SNMP	Simple Network Management Protocol
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SPL	Simplified Policy Language
SQL	Structured Query Language
SSL	Secure Socket Layer
TCP/IP	Transmission Control Protocol / Internet Protocol
TLS	Transport Layer Security
UDE	Undesirable Effects
UDP	User Datagram Protocol
UML	Unified Modeling Language
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium. It is the main international standards organization for the World Wide Web
WS-Addressing	A W3C recommendation and defines a transport neutral mechanism to address Web services and messages
WS-CIM	Web Services CIM. This DMTF standard defines a translation from the MOF format for CIM into XML schema
WS-Enumeration	A specification intended to support enumeration of data sources that cannot practically fit into a single SOAP message
WS-Eventing	Defines a protocol for web services to subscribe to another web service, or to accept a subscription from another web service.
WS-EventNotification	Describes how to support distributed events and notifications, as in publish and subscribe scenarios, independent of transport protocol. It is a work-in-progress that represents the unification of other competing specifications: WS-Notification and WS-Eventing
WS-Management	A Web services protocol standardized by the DMTF for

	managing devices, services, and systems. It is built on three fundamental specifications: WS-Transfer, WS-Eventing, and WS-Enumeration
WS-Notification	A group of specifications related to the WS-Resource Framework that allows event driven programming between web services
WS-Policy	Provides a general purpose mechanism for describing and communicating policies relating to a Web service
WS-Resource Framework	A family of OASIS published specifications for web services. It provides a set of operations that web services may implement to become stateful; web service clients communicate with resource services which allow data to be stored and retrieved
WS-Resource Transfer	Standardize how to use Web services for resource creation, access, manipulation, destruction and as well as managing its lifecycle
WS-Secure Conversation	A specification defines extensions for WS-Security and WS-Trust for establishing and sharing security contexts
WS-Security	A comprehensive standard from OASIS that provides confidentiality, integrity, and authentication for SOAP-based messages
WS-Security Policy	A specification defines the policy assertions for use with WS-Policy that relate to WS-Trust, WS-Security, and WS-SecureConversation.
WS-Transfer	A specification defining the transfer of an XML representation of a WS-addressable resource, as well as creating and deleting such resources
WS-Trust	A specification defines extensions to WS-Security to provide a framework for requesting and issuing security tokens, and to broker trust relationships.
WSDL	Web Services Description Language
WSDM	Web Services Distributed Management. A management standard from OASIS which consists of two components, MUWS and MOWS.
XML	Extensible Markup Language
XSD	XML Schema Definition
XSS	Cross Site Scripting

Appendix B: Use Case Two

- **Prevention – Domino's pizza wants to take preventative measures to ensure systems function efficiently during SuperBowl Sunday.**
  - Signs & Symptoms:
    - SuperBowl Sunday
  - Diagnostician engine gets signs/symptoms, initiates case history and begins diagnosis.
  - The diagnosis( ) relates the symptoms/signs to Conditions
    - SuperBowl Sunday; Poor online ordering webapp response time; high requests/min → Web server farm capacity exceeded
    - Christmas shopping season; SuperBowl Sunday; Slow credit card transactions; Broken card reader; Service provider failure → Credit card reader "no response"
  - Diagnostician engine recommends Diagnostics of type "preventative".
    - Web server farm capacity exceeded → Check requests/min vs capacity; Is it SuperBowl Sunday?; Check response time
    - Credit card reader "no response" → Try different card reader/Try different credit card; Is it Christmas shopping season?; Is it SuperBowl Sunday?; Is SP available?
  - Practitioner applies Diagnostics of type "preventative" (none of type 'preventative' above, so just continue)
  - The Diagnostician recommends Treatments of type "preventative" (based upon "Possible Treatments" in the Condition Obj)
    - Web server farm capacity exceeded → 1. Bring all offline servers online; 2. Provision more capacity; 3. preventative: Pre-provision more capacity; 4. preventative: ensure no server maintenance is scheduled
    - Credit card service provider busy → 1. Run paper transaction (instead of electronic) 2. preventative: Sign-up for premium service
  - Practitioner applies Treatments (in red above)
- **Lessons learned: Prevention will require Diagnostics & Treatments to have the notion of a "preventative" tag**

Use Case Two from CA Inc.



Workflow of Use Case Two

When taking a close look upon Use Case Two, it is found that this is a workflow which describes the scenario of diagnosing a *Superbowl Sunday* problem in an enterprise application environment. The workflow can be interpreted as follows:

(1) When a network administrator finds that *Superbowl Sunday* (as a sign) arrives, he/she should make a hypothesis that *Web server farm capacity exceeded* might happens. We labelled it as *hypothesis one.*

(2) When a network administrator finds that *Christmas shopping season* (as a sign) arrives, he/she should make a hypothesis that *Credit card reader "no response"* might happens. We labelled it as *hypothesis two.*

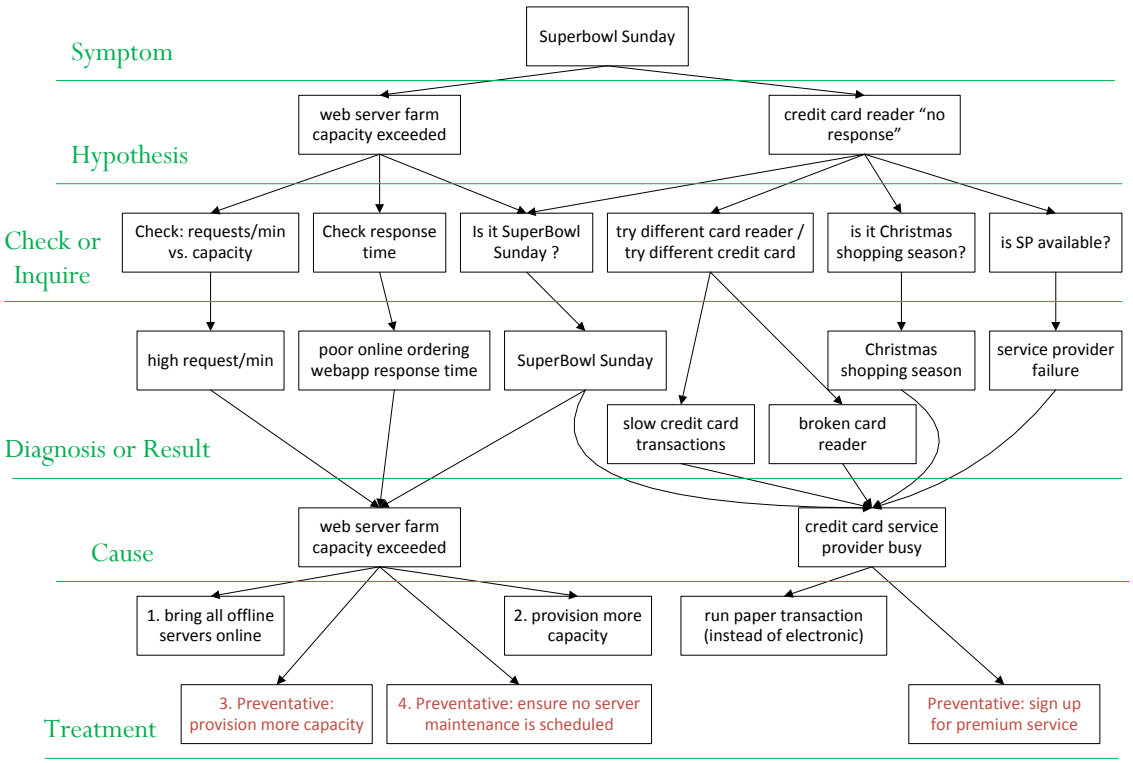
(3) With *hypothesis one: Web server farm capacity exceeded*, the administrator needs to *Check requests/min vs. capacity; Is it Superbowl Sunday?; Check response time.*

(4) With *hypothesis two: Credit card reader "no response"*, the administrator needs to *Try different card reader/Try different credit card; Is it Christmas shopping season?; Is it Superbowl Sunday?;Is SP available?*

(5) If *hypothesis one* is true—*Web server farm capacity exceeded,* He/she may take following actions:
   (a) Bring all offline servers online;
   (b) Provision more capacity;
   (c) Preventative: provision more capacity;
   (d) Preventative: ensure no server maintenance is scheduled.

(6) If *hypothesis two* is true—*Credit card provider is busy,* He/she may take following actions:
   (a) Run paper transaction (instead of electronic);
   (b) Preventative: Sign-up for premium service.

Each box in the workflow diagram is considered as a process and each process leads to a corresponding event. Every event in the use cases can be classified into the following six types: *Symptom*, *Hypothesis*, *Check or Inquire*, *Diagnosis or Result*, *Cause*, and *Treatment*. Thus, the boxes / nodes of Use Case Two workflow diagram can be clustered / classified by the six event types as below.

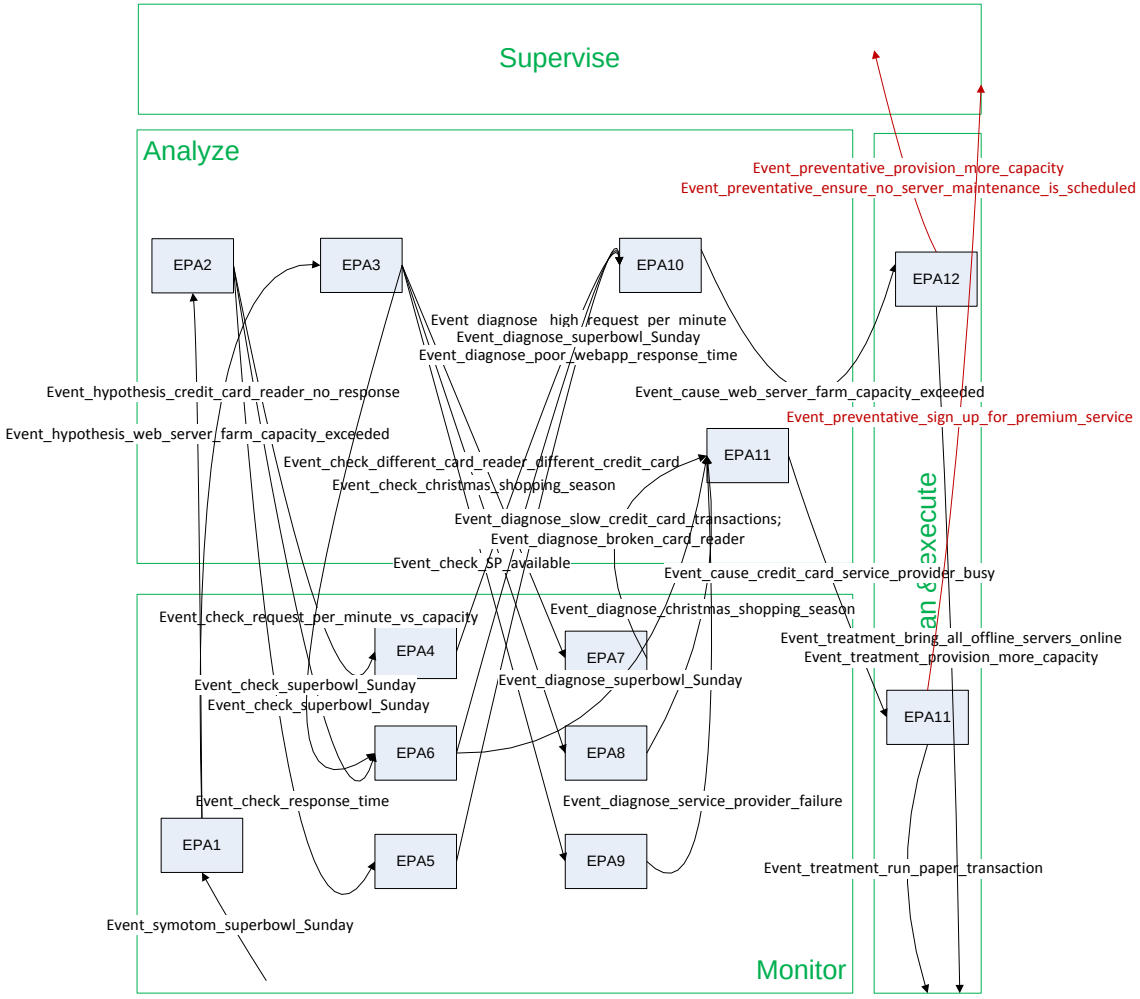


The following table shows the event subtypes we defined in the Use Case Two Unit:

Event type	**Event subtype**	**Event description**
Symptom	Event_symptom_superbowl_sunday	Sign: SuperBowl Sunday
Hypothesis	Event_hypothesis_web_server_farm_capacity_exceeded	Hypothesis: web server farm capacity exceeded
	Event_hypothesis_credit_card_reader_no_response	Hypothesis: credit card reader "no response"
Check or Inquire	Event_check_request_per_minute_vs_capacity	Check: request / minute vs. capacity
	Event_check_response_time	Check: response time

	Event_check_superbowl_sunday	Check: is it SuperBowl Sunday?
	Event_check_different_card_reader_different_credit_card	Check: try different card reader / try different credit card
	Event_check_christmas_shopping_season	Check: is it Christmas shopping season?
	Event_check_SP_available	Check: is service provider available?
Diagnosis or Result	Event_diagnose _high_request_per_minute	Diagnose result: high request/min
	Event_diagnose_poor_webapp_response_time	Diagnose result: poor online ordering webapp response time
	Event_diagnose_superbowl_sunday	Diagnose result: SuperBowl Sunday
	Event_diagnose_slow_credit_card_transactions	Diagnose result: slow credit card transactions
	Event_diagnose_broken_card_reader	Diagnose result: broken card reader
	Event_diagnose_christmas_shopping_season	Diagnose result: Christmas shopping season
	Event_diagnose_service_provider_failure	Diagnose result: service provider failure
Cause	Event_cause_web_server_farm_capacity_exceeded	Root cause: web server farm capacity exceeded
	Event_cause_credit_card_service_provider_busy	Root cause: credit card service provider busy
Treatment	Event_treatment_bring_all_offline_servers_online	Treatment: bring all offline servers online
	Event_treatment_provision_more_capacity	Treatment: provision more capacity
	Event_preventative_provision_more_capacity	Preventative: provision more capacity
	Event_preventative_ensure_no_server_maintenance_is_scheduled	Preventative: ensure no server maintenance is scheduled
	Event_treatment_run_paper_transaction	Treatment: run paper transaction
	Event_preventative_sign_up_for_premium_service	Preventative: sign up for premium service

Based on the event subtypes defined in above table, we can build the EPN of Use Case Two Unit according to the Use Case Two Workflow.



The EPN of Use Case Two Unit consists of eleven EPAs (EPA1 toEPA13). They are defined in following tables:

EPA Type	EPA ID	Input event(s)	Output event(s)
Monitor (I)	1	Event_symptom_superbowl_sunday	Event_hypothesis_web_server_farm_capacity_exceeded; Event_hypothesis_credit_card_reader_no_response
Analyze (I)	2	Event_hypothesis_web_server_farm_capacity_exceeded	Event_check_request_per_minute_vs_capacity; Event_check_response_time; Event_check_superbowl_sunday
	3	Event_hypothesis_credit_card_reader_no_response	Event_check_superbowl_Sunday; Event_check_different_card_reader_different_credit_card; Event_check_christmas_shopping_se

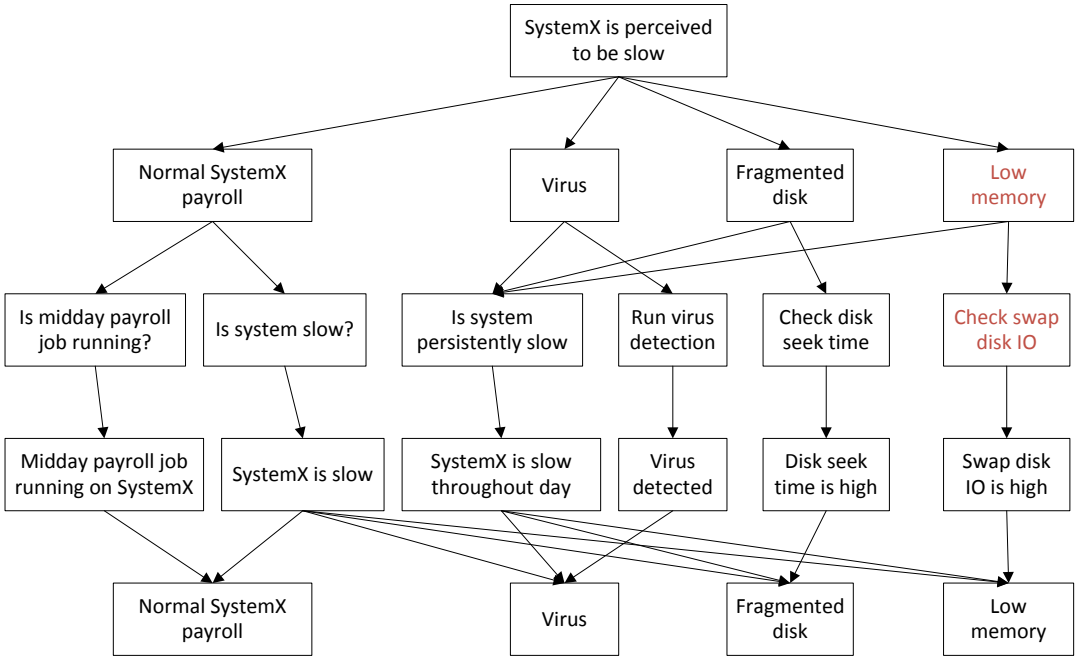
			ason; Event_check_SP_available

EPA Type	EPA ID	Input event(s)	Output event(s)
Monitor (II)	4	Event_check_request_per_minute_vs_capacity	Event_diagnose_high_request_per_minute
	5	Event_check_response_time	Event_diagnose_poor_webapp_response_time
	6	Event_check_superbowl_sunday	Event_diagnose_superbowl_sunday
	7	Event_check_different_card_reader_different_credit_card	Event_diagnose_slow_credit_card_transactions; Event_diagnose_broken_card_reader
	8	Event_check_christmas_shopping_season	Event_diagnose_christmas_shopping_season
	9	Event_check_SP_available	Event_diagnose_service_provider_failure
Analyze (II)	10	Event_diagnose_high_request_per_minute; Event_diagnose_poor_webapp_response_time; Event_diagnose_superbowl_sunday	Event_cause_web_server_farm_capacity_exceeded
	11	Event_diagnose_superbowl_Sunday; Event_diagnose_slow_credit_card_transactions; Event_diagnose_broken_card_reader; Event_diagnose_christmas_shopping_season; Event_diagnose_service_provider_failure	Event_cause_credit_card_service_provider_busy
Plan & execute	12	Event_cause_web_server_farm_capacity_exceeded	Event_treatment_bring_all_offline_servers_online; Event_treatment_provision_more_capacity; Event_preventative_provision_more_capacity; Event_preventative_ensure_no_server_maintenance_is_scheduled
	13	Event_cause_credit_card_service_provider_busy	Event_treatment_run_paper_transaction; Event_preventative_sign_up_for_premium_service

Appendix C: Use Case Three

- Diagnostic Side Effects – Critical system is slow due to low memory, but recommended diagnostic slows it even further
  - Signs & Symptoms:
    - SystemX is perceived to be slow
  - Diagnostician engine gets signs/symptoms, initiates case history and begins diagnosis.
  - The diagnosis( ) relates the symptoms/signs to Conditions
    - SystemX is slow; Midday payroll job running on SystemX → Normal SystemX Payroll condition
    - SystemX is slow; SystemX is slow throughout day; Virus detected → Virus
    - SystemX is slow; SystemX is slow throughout day; Disk seek time is high → Fragmented disk
    - SystemX is slow; SystemX is slow throughout day; Swap disk IO is high → Low memory
  - Diagnostician engine recommends Diagnostics (based upon "Confirming Diagnostics" in the Condition Obj)
    - Normal SystemX Payroll → Is midday payroll job running?; Is System slow?
    - Virus → Is system persistently slow?; Run virus detection
    - Fragmented disk → Is system persistently slow?; Check disk seek time
    - Low memory → Is system persistently slow?; Check swap disk IO
  - Practitioner applies Diagnostics (all above)
  - The Diagnostics process generates new Signs/Symptoms
    - System is persistently slow
    - No virus detected
    - Disk seek time is high
    - Swap disk IO is high
    - Side effect: Virus scanner consumes more memory, disk IO, etc... causing system to slow even further
- Similarly, Treatments can also produce side effects. See next slide for an example.
- Lessons learned: Diagnostics & Treatments can be complex workflows (that would likely limit the side effects)??

Use Case Three from CA Inc.



Workflow of Use Case Three

When taking a close look upon Use Case Three, it is found that this is a workflow which describes the scenario of diagnosing a *SystemX is perceived to be slow* problem in an enterprise application environment. The workflow can be interpreted as follows:

(1) When a network administrator finds that *SystemX is perceived to be slow* (as a sign), he/she should make following four hypotheses:
   (a) *Normal SystemX payroll;* We labelled it as *hypothesis one.*
   (b) *Virus;* We labelled it as *hypothesis two.*
   (c) *Fragmented disk;* We labelled it as *hypothesis three.*
   (d) *Low memory;* We labelled it as *hypothesis four.*

(2) With *hypothesis one: Normal SystemX payroll,* the administrator needs to check: *Is midday payroll job running?; Is System slow?.*

(3) With *hypothesis two: Virus,* the administrator needs to check: *Is system persistently slow?; Run virus detection*

(4) With *hypothesis three: Fragmented disk,* the administrator needs to check: *Is system persistently slow?; Check disk seek time*

(5) With *hypothesis four: Low memory,* the administrator needs to check: *Is system persistently slow?; Check swap disk IO*

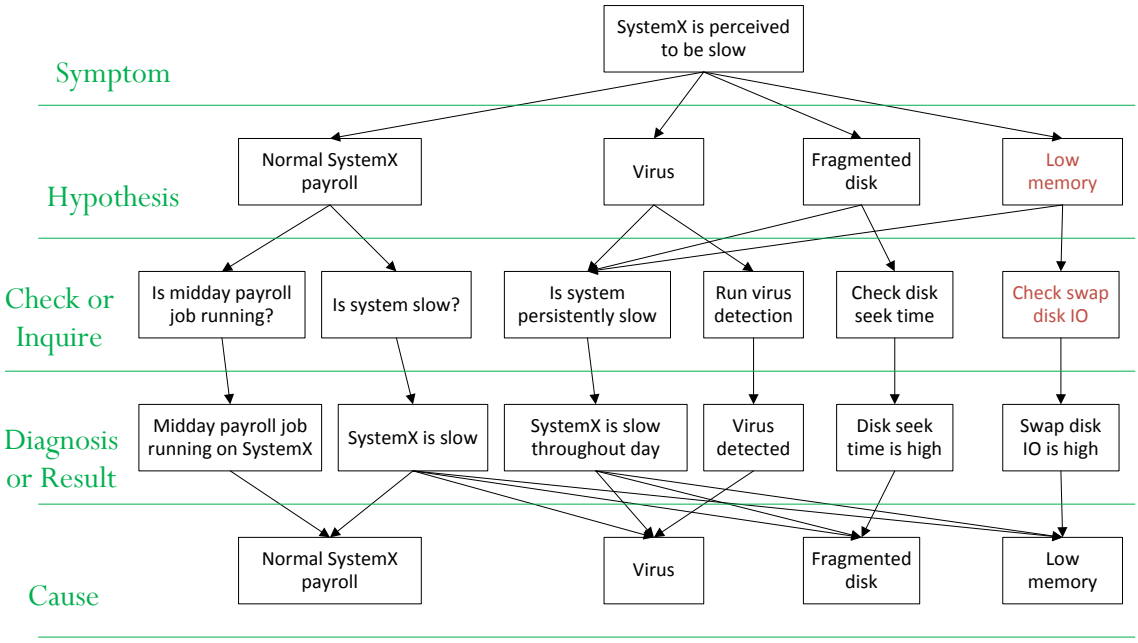
(6) If *SystemX is slow*; *Midday payroll job running on SystemX,* then *hypothesis one* is true—*Normal SystemX payroll condition.*

(7) If *SystemX is slow*; *SystemX is slow throughout day; Virus detected,* then *hypothesis two* is true—*Virus*

(8) If *SystemX is slow; SystemX is slow throughout day; Disk seek time is high,* then hypothesis three is true—*Fragmented disk*

(9) If *SystemX is slow; SystemX is slow throughout day; Swap disk IO is high*, then hypothesis three is true—*Low memory*

Each box in the workflow diagram is considered as a process and each process leads to a corresponding event. Every event in the use cases can be classified into the following five types: *Symptom, Hypothesis, Check or Inquire, Diagnosis or Result, and Cause*. *Treatment* is absent in this case. Thus, the boxes / nodes of Use Case Two workflow diagram can be clustered / classified by the five event types as below.

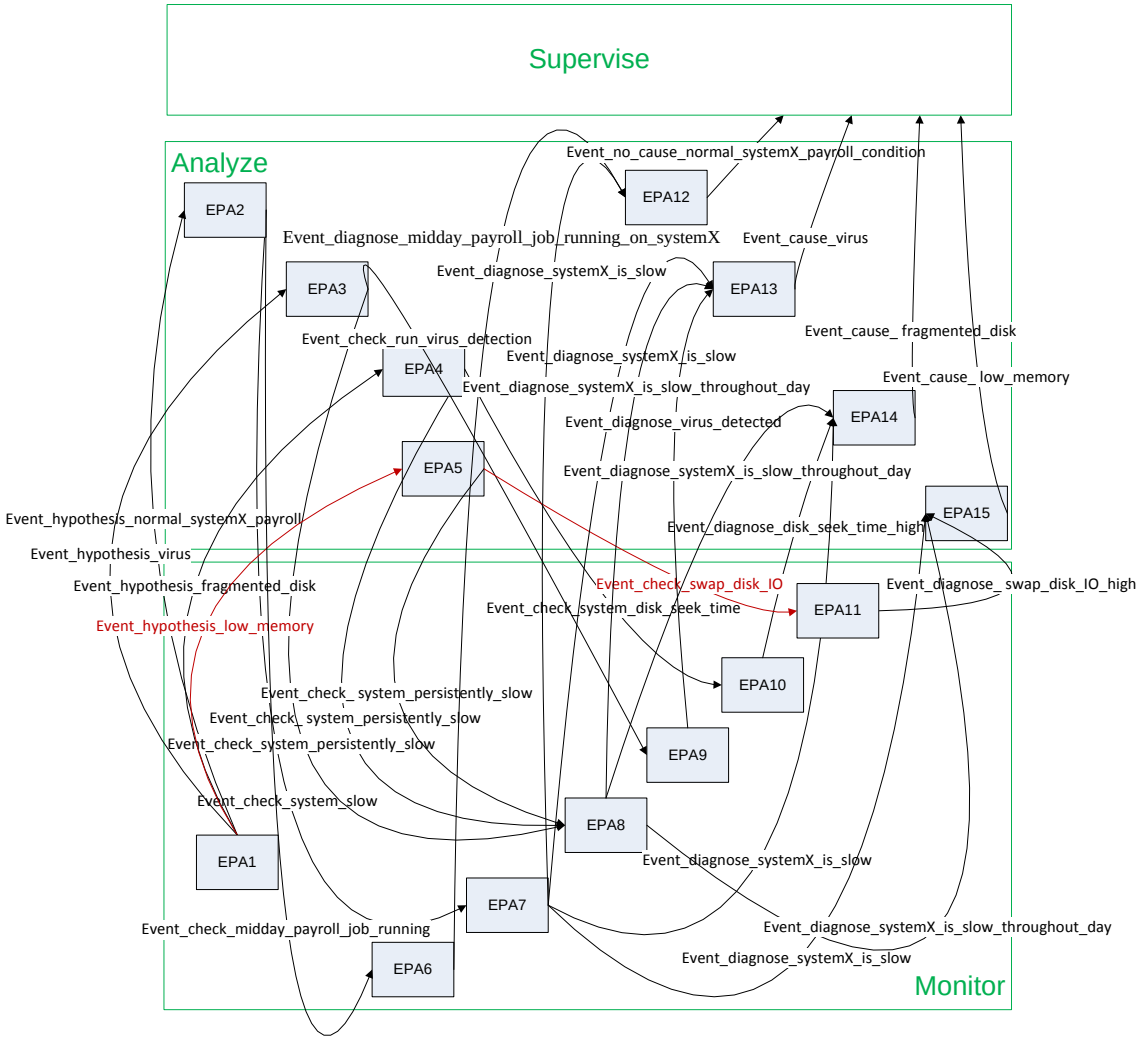


The following table shows the event subtypes we defined in the Use Case Three Unit:

Event type	**Event subtype**	**Event description**
Symptom	Event_symptom_systemX_is_slow	Sign: SystemX is perceived to be slow
Hypothesis	Event_hypothesis_normal_systemX_payroll	Hypothesis: normal SystemX midday payroll
	Event_hypothesis_virus	Hypothesis: virus
	Event_hypothesis_fragmented_disk	Hypothesis:fragmented disk

	Event_hypothesis_low_memory	Hypothesis: low memory
Check or Inquire	Event_check_midday_payroll_job_running	Check: is midday payroll job running?
	Event_check_system_slow	Check: is system slow?
	Event_check_system_persistently_slow	Check: is system persistently slow?
	Event_check_run_virus_detection	Check: run virus detection
	Event_check_ disk_seek_time	Check: disk seek time
	Event_check_swap_disk_IO	Check: swap disk IO
Diagnosis or Result	Event_diagnose_midday_payroll_job_running_on_ systemX	Diagnose result: midday payroll job running on SystemX
	Event_diagnose_systemX_is_slow	Diagnose result: systemX is slow
	Event_diagnose_systemX_is_slow_throughout_day	Diagnose result: SystemX is slow throughout day
	Event_diagnose_virus_detected	Diagnose result: virus detected
	Event_diagnose_disk_seek_time_high	Diagnose result: disk seek time is high
	Event_diagnose_ swap_disk_IO_high	Diagnose result: swap disk IO is high
Cause	Event_cause_normal_systemX_payroll_condition	Root (no) cause: normal SystemX payroll condition
	Event_cause_virus	Root cause: virus
	Event_cause_ fragmented_disk	Root cause: fragmented disk
	Event_cause_ low_memory	Root cause: low memory

Based on the event subtypes defined in above table, we can build the EPN of Use Case Two Unit according to the Use Case Three Workflow.



The EPN of Use Case Three Unit consists of eleven EPAs (EPA1 toEPA11). They are defined in following tables:

EPA Type	EPA ID	Input event(s)	Output event(s)
Monitor (I)	1	Event_symptom_systemX_is_slow	Event_hypothesis_normal_systemX_payroll; Event_hypothesis_virus; Event_hypothesis_fragmented_disk; Event_hypothesis_low_memory
Analyze (I)	2	Event_hypothesis_normal_systemX_payroll	Event_check_midday_payroll_job_running; Event_check_system_slow
	3	Event_hypothesis_virus	Event_check_system_persistently_slow;

			Event_check_run_virus_detection
	4	Event_hypothesis_fragmented_disk	Event_check_system_persistently_slow; Event_check_ disk_seek_time
	5	Event_hypothesis_low_memory	Event_check_system_persistently_slow; Event_check_swap_disk_IO

EPA Type	EPA ID	Input event(s)	Output event(s)
Monitor (II)	6	Event_check_midday_payroll_job_running	Event_diagnose_midday_payroll_job_running_on_systemX
	7	Event_check_system_slow	Event_diagnose_systemX_is_slow
	8	Event_check_system_persistently_slow	Event_diagnose_systemX_is_slow_throughout_day
	9	Event_check_run_virus_detection	Event_diagnose_virus_detected
	10	Event_check_ disk_seek_time	Event_diagnose_disk_seek_time_high
	11	Event_check_swap_disk_IO	Event_diagnose_swap_disk_IO_high
Analyze (II)	12	Event_diagnose_midday_payroll_job_running_on_systemX; Event_diagnose_systemX_is_slow	Event_cause_normal_systemX_payroll_condition
	13	Event_diagnose_systemX_is_slow; Event_diagnose_systemX_is_slow_throughout_day; Event_diagnose_virus_detected	Event_cause_virus
	14	Event_diagnose_systemX_is_slow; Event_diagnose_systemX_is_slow_throughout_day; Event_diagnose_disk_seek_time_high	Event_cause_ fragmented_disk
	15	Event_diagnose_systemX_is_slow; Event_diagnose_systemX_is_slow_throughout_day; Event_diagnose_swap_disk_IO_high	Event_cause_ low_memory

Appendix D: ESPER Introduction

The Esper engine was designed to process complex events and analyze event streams in real-time or near real-time settings. Esper manual gives some typical examples of applications [Espe09]:

- Network and application monitoring (intrusion detection, SLA monitoring),
- Sensor network applications (scheduling and control of fabrication lines, air traffic),
- Business process management and automation (process monitoring, business activity monitoring), and
- Finance (algorithmic trading, fraud detection, risk management).

Unlike databases, which store data and run queries against stored data, the Esper engine allows applications to store queries and run the data through. The matching process is thus continuous rather than only occurring when a query (such as a SQL query) is submitted into the database.

Event Representations

In Esper [Espe09], an event can be represented by any of the following underlying Java objects:

Table 29: Event underlying Java objects in Esper

Java Class	Description
java.lang.Object	Any plain-old java object with getter methods following JavaBean conventions; Legacy Java classes not following JavaBean conventions can also serve as events .
java.util.Map	Map events are key-values pairs
org.w3c.dom.Node	XML document object model (DOM)

"Plain-old Java object events" are object instances that expose event properties through JavaBeans-style getter methods. Events classes or interfaces do not have to be

fully compliant to the JavaBean specification; however, for the Esper engine to obtain event properties, the required JavaBean getter methods must be present.

Esper Processing Model

The Esper processing model is continuous: update listeners and/or subscribers to statements received, and update data as soon as the engine processes events for that statement. The interface for listeners is com.espertech.esper.client.UpdateListener. Implementations must provide a single update method that the engine invokes when results become available:

```
update(EventBean[] newEvents, EventBean[] oldEvents) : void
```

The engine provides statement results to update listeners by placing results in com.espertech.esper.event.EventBean instances. A typical listener implementation queries the EventBean instances via getter methods to obtain the statement-generated results.

```
get(String property) : Object
getUnderlying() : Object
getEventType() : EventType
```

The Event Processing Language

The Event Processing Language (EPL) of Esper:

- Uses a SQL-like language with SELECT, FROM, WHERE, GROUP BY, HAVING and ORDER BY clauses.
- Selects properties from streams. EPL statements are used to derive and aggregate information from one or more streams of events, and to join or merge event streams.

- Uses aggregation functions, including: sum, avg, count, min/max, median and stddev.
- Uses predefined views (time, length, batch mode). Some views represent windows over a stream of events. Other views derive statistics from event properties, group events or handle unique event property values. Views can be staggered onto each other to build a chain of views.

Selecting all event properties in a stream	select * from stream_def
Selecting events based on values	select * from RawEvent(EventID='#%#$##$$@!!')
Time Window	select EventID, count(*) from RawEvent.win:time(30 sec)
Length Window	select * from CursingEvent.win:length(10)
Select As	select count(*) as EventCount from RawEvent.win:time(10 sec) long EventCount = ((Long) newEvents[0].get("eventcount")). longValue();
Group By/Having	select Symptom, count(*) from RawEvent.win:time(30 sec) group by Symptom having count(*) > 10

Data views can be defined as:

win:length(n)	Keep only n events
win:length_batch(n)	Batch up events until n received
win:time(p)	Keep only events from last period p
win:time_batch(p)	Batch up events from period p
win:time_accum(p)	Accumulate events for period p until no more
win:keepall()	Keep all inserted events
std:unique(prop)	Keep most recent event that has unique value for property
std:groupby(prop)	Group sub-views based on property value
std:lastevent()	Keep only the most recent value

Esper Interface

The user can use Esper's administrative interface to create and manage EPL and pattern statements, and set runtime configurations. An instance of the Esper engine is obtained via static methods on the EPServiceProviderManager class. The getDefaultProvider method and the getProvider(String providerURI) methods return an instance of the Esper engine. The EPServiceProviderManager determines if the provider URI matches all prior provider URI values and returns the same engine instance for the same provider URI value. If the provider URI has not been seen before, it creates a new engine instance.

The next code snippet outlines a typical sequence of use [Espe09]:

```
// Configure the engine, this is optional
Configuration config = new Configuration();
config.configure("configuration.xml"); // load a configuration from file
config.set....(...); // make additional configuration settings
// Obtain an engine instance
EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider(config);
// Optionally, use initialize if the same engine instance has been used before to start clean
epService.initialize();
// Optionally, make runtime configuration changes
epService.getEPAdministrator().getConfiguration().add...(...);
// Destroy the engine instance when no longer needed, frees up resources
epService.destroy();
```

Appendix E: Implementation Details

In the following code snippet, we configure the Esper engine first, and then obtain an Esper engine instance. We exam incoming request messages to detect if they include strings like “script” (which indicates Cross-site Scripting (XSS) might happen) or strings like “1=1” (a condition constantly true indicates SQL Injection might happen). In a sophisticated implementation, these detected conditions should be fetched from a knowledgebase and such an example is introduced in Lin’s Master thesis [Lin10].

```
Public class EsperAdaptor
{
    private static EsperAdaptor instance = null;
    private EsperAdaptor() { }
    private EPServiceProvider epService;
    public void 215nitialize()
    {
        // configure the engine
        Configuration config = new Configuration();
        config.addEventType(“RequestMonitor”, RequestMonitor.class);
        config.addEventType(“ResponseMonitor”, ResponseMonitor.class);
        config.addEventType(“AbstractMonitor”, AbstractMonitor.class);

        // obtain an engine instance
        epService = EPServiceProviderManager.getDefaultProvider(config);
        System.out.println(“Esper engine=” + epService.toString());

        EPStatement statement = null;
        String stmt = null;

        //detect upon incoming request
        String whereString = “where queryString like ‘%script%’ or queryString like ‘%or 1=1
%’”;
```

```
        stmt = "select * from RequestMonitor " +  whereString;
        statement = epService.getEPAdministrator().createEPL(stmt);
        statement.addListener(new RequestMonitorListener());

        // detect upon outgoing response
        String whereString1 = "where queryString like '%script%' or queryString like
'%Welcome%'";
        stmt = "select * from ResponseMonitor " + whereString1;
        statement = epService.getEPAdministrator().createEPL(stmt);
        statement.addListener(new ResponseMonitorListener());
    }

    public void sendEvent(Object event)
    { epService.getEPRuntime().sendEvent(event); }
}
```

Esper engine will emit a new event once it detects a matching pattern. Such a new event will be processed by UpdateListener. The following code snippet is an implementation of UpdateListener.

```
    Public abstract class BaseListener implements UpdateListener
    {
        public void update(EventBean[] newEvents, EventBean[] oldEvents)
        {
                System.out.println("Using Esper Event Stream Processing : \n\t" +
newEvents[0].toString());
                AbstractMonitor monitor = (AbstractMonitor) newEvents[0].getUnderlying();
                monitor.setIsAttack(true);
                monitor.getOriginalIntrusions().add(ABIDSUtils.getIntrusionType(monitor));
        }
    }
```