Abstract

Monitoring and Diagnosis for Autonomic Systems: A Requirement Engineering Approach

Yiqiao Wang
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2010

Autonomic computing holds great promise for software systems of the future, but at the same time poses great challenges for Software Engineering. Autonomic computing research aims to design software systems that self-configure, self-repair, self-optimize and self-protect, so as to reduce software maintenance cost while improving performance. The aim of our research is to develop tool-supported methodologies for designing and operating autonomic systems. Like other researchers in this area, we assume that autonomic system architectures consist of monitoring, analysis/diagnosis, planning, and execution components that define a feedback loop and serve as the basis for system self-management.

This thesis proposes an autonomic framework founded on models of requirements and design. This framework defines the normal operation of a software system in terms of models of its requirements (goal models) and/or operation (statechart models). These models determine what to monitor and how to interpret log data in order to diagnose failures.

The monitoring component collects and manages log data. The diagnostic component analyzes log data, identifies failures, and pinpoints problematic components. We transform the diagnostic problem into a propositional satisfiability (SAT) problem solvable by off-the-shelf SAT solvers. Log data are preprocessed into a compact propositional...
encoding that scales well with growing problem size. For repair, our compensation com-
ponent executes compensation actions to restore the system to an earlier consistent state.
The framework repairs failures through reconfiguration when monitoring and diagnosis
use requirements. The reconfiguration component selects a best system reconfiguration
that contributes most positively to the system’s non-functional requirements. It selects a
reconfiguration that achieves this while reconfiguring the system minimally. The frame-
work does not currently offer a repair mechanism when monitoring and diagnosis use
statecharts.

We illustrate our framework with two medium-sized, publicly-available case studies.
We evaluate the framework’s performance through a series of experiments on randomly
generated and progressively larger specifications. The results demonstrate that our ap-
proach scales well with problem size, and can be applied to industrial sized software
applications.
Acknowledgements

First and foremost, I want to thank my supervisor Dr. John Mylopoulos. It has been an honour and a pleasure to be his Ph.D. student. This thesis would not have been possible without his on-going and kind encouragement, support, and guidance. I am grateful to my paper co-authors Dr. Sheila McIlraith and Dr. Yijun Yu. I want to thank Sheila for teaching me many things about Artificial Intelligence (AI). A significant portion of my thesis is based on AI theories of action and diagnosis. Sheila’s guidance made this possible. I am grateful to Yijun for spending countless hours discussing with me possible solutions to address many issues that I faced during this work. I thank my committee members Dr. Steve Easterbrook, Dr. Sheila McIlraith, and Dr. Dave Wortman for their guidance and advice throughout my Ph.D. research. I thank Dr. Eric Yu and Dr. William Robinson for participating in my external thesis defense and for their insightful comments and support.

On the personal front, I am grateful to my best friend and husband Borys Bradel. His friendship, love, understanding, encouragement, and support has been a big part of my Ph.D. experience and my life. I am also grateful to Borys for his expert advice on how to optimize my framework implementation for better scalability. I thank Patrick Lewtas, for both being a close friend and for proofreading many chapters of this thesis. I thank my many colleagues and friends in the Software Engineering lab at the University of Toronto. Finally, I thank my parents Yan Zhang and Yongzhang Wang, who have supported me throughout my life. To them I dedicate this thesis.
## Contents

1 Introduction .............................................. 1

2 Literature Survey ....................................... 8
  2.1 Goal Oriented Requirement Engineering ................. 8
  2.2 Autonomic Computing .................................. 19
  2.3 Monitoring Software Systems ......................... 22
  2.4 AI Theories of Diagnosis ............................. 42
  2.5 Reconfiguration ....................................... 55

3 Architecture Overview .................................... 59
  3.1 Preparing a Legacy Software System .................. 59
  3.2 Architecture Overview ................................ 61
  3.3 Implementation Details ............................... 64

4 Requirement Monitoring and Diagnosis ..................... 66
  4.1 Preliminaries .......................................... 66
  4.2 A Running Example ................................... 68
  4.3 Monitoring ............................................. 69
  4.4 Diagnosis ............................................... 75
  4.5 Evaluation ............................................. 95
  4.6 Multi-layer Monitoring and Diagnosis .................. 103
List of Tables

4.1 Squirrel Mail Annotated Goal Model ............................................. 72
4.2 Optimization of Algorithm 4 Over Algorithm 3 ............................... 91
4.3 Tradeoff Between Monitoring Overhead and Diagnostic Precision (First Set of Experiments) .......................................................... 99
4.4 Scalability to Goal Model Size with Log Preprocessing (Second Set of Experiments) ............................................................. 100
4.5 Number of Goals and Tasks at Each SOA Layer ............................... 107
5.1 ATM Annotated States ................................................................. 119
5.2 ATM Annotated Transitions ......................................................... 119
5.3 Karnaugh Map for Transition Denial .............................................. 127
5.4 Scalability to Statechart Size (First Set of Experiments) ................. 135
5.5 Scalability to Statechart Size (Second Set of Experiments) ............. 136
6.1 Finding a Best Global Reconfiguration ........................................... 150
6.2 Choosing a Best Global Reconfiguration for ATM ................................ 156
7.1 Annotated Goal Model for the Online Ordering System .................... 174
7.2 Sample Log Data for the Online Ordering System .......................... 175
7.3 Participating Diagnostic Components for the Online Ordering System ... 176
7.4 Preconditions and Effects of Monitored Goals in the Day Trading Application 176
## List of Figures

2.1 The Sub-Models of the F3 Approach [BK94] ........................................ 18
2.2 Structure of an Autonomic Element [KC03] ...................................... 21
3.1 Architecture Overview ............................................................. 62
4.1 Squirrel Mail Goal Model .......................................................... 69
4.2 Example Goal Model .................................................................. 74
4.3 Partial ATM Goal Model ............................................................. 97
4.4 Scalability to Goal Model Size (Encoding with Log Preprocessing) .... 101
4.5 Comparison of Encoding and Diagnostic Time Taken by the Two Encoding Algorithms .......................................................... 102
4.6 Comparison of Size of $\Phi$ Generated by the Two Encoding Algorithms .......................................................... 103
4.7 ATM Global Goal Model at the 3 Layers of SOA ............................ 105
4.8 Scalability to Goal Model Size - SOA ......................................... 108
5.1 Example Statechart ................................................................. 114
5.2 ATM Simulation Statechart ....................................................... 116
5.3 Enabledness of a transition [MLP97] ........................................... 124
5.4 Basic Step Algorithm[MLP97] ..................................................... 125
5.5 Scalability to Number of States and Transitions (First Set of Experiments) 136
5.6 Scalability to Number of Transitions (Second Set of Experiments) .... 137
6.1 Example Goal Model ............................................................... 141
6.2 Data Structures Created When Calculating a Best Configuration for G3
   in Figure 6.1 ................................................................. 152
6.3 Extended ATM Goal Model .............................................. 155
6.4 Performance Evaluation of Algorithms 8, 9, and 11 ............... 159

7.1 SAM UI for the Online Ordering System: High Response Time .... 166
7.2 SAM UI for the Online Ordering System: Server Topology ....... 167
7.3 SAM UI for the Online Ordering System: Server 1 not Bind to Port . 168
7.4 SAM UI for the Online Ordering System: Server 3 is Overloaded .... 169
7.5 SAM UI for the Online Ordering System: Load Balancer not Working on
       Server 3 .................................................................. 170
7.6 SAM UI for the Online Ordering System: Diagnostic Knowledge Base .. 171
7.7 Goal model for the Online Ordering System .......................... 172
7.8 High Level Goal Model of the Day Trading Application ............ 172
7.9 Decomposition of Goal “Perform Transaction” of Figure 7.8 ....... 173
Chapter 1

Introduction

The ever-increasing complexity of software systems makes them ever more difficult and costly to maintain. This demands a growing number of skilled IT professionals to understand and maintain system components and their interoperations. Such human involvement is expensive, non-scalable, and error-prone. Autonomic computing arose in response to these trends [GC03, Hor03, KC03]. According to [Aut], “an autonomic system is a system that operates and serves its purpose by managing itself without external intervention even in case of environmental changes.” Autonomic computing, an IBM initiative and a rapidly growing research area, aims at creating self-managing software systems. These shift management complexity from administrators to the software systems themselves. Once system administrators specify high level goals and policies, autonomic systems perform all self-management tasks. These include self-configuration, self-optimization, self-healing, and self-protection. An autonomic system therefore needs to monitor itself and its environment, analyze and interpret monitored data, plan all appropriate changes, then execute the plan. This Monitor, Analyze, Plan, and Execution loop has become known as the MAPE loop for autonomic systems [GC03, Hor03, KC03]. Some researchers refer to autonomic systems as adaptive systems. We treat the two terms as synonyms.
There are many challenges in designing autonomic systems, and in making existing legacy software systems more autonomic. Firstly, it is difficult to fully understand existing complex legacy software systems, and to continue to understand evolving systems. We need to understand not only the “what” (functional requirements) but also the “why” (nonfunctional requirements) aspects of a system. Secondly, it is difficult to monitor the behavior of a software system. We need to be able to answer the research questions of “what” should be monitored, and “where” the monitors should be inserted. The log data should be generated at a right granularity so that it doesn’t contain too much or too little information. Thirdly, it is difficult to diagnose failures. It is difficult to manage and correlate log data, especially data that are generated under different systems, and in different format. It is difficult to make sense of the data. It is difficult to determine the root causes to failures. Finally, it is difficult to repair the failures when they occur. Some failures are not repairable, and some can only be repaired to a certain degree. It is difficult to repair failures efficiently without causing unwanted side effects to the rest of the system.

Our research aims to address some of these challenges, and to developing tool-supported methodologies for designing and running “autonomic” components for software-intensive systems. In this thesis, we propose a model-based autonomic framework that contains monitoring, diagnostic, reconfiguration, and execution components [WMYM07, WMYM09, WM09]. The framework (1) monitors a system’s operations for failures; (2) diagnoses root causes when failures occur; (3) identifies the actual failure when multiple diagnoses explain monitored data; and (4) generates a system reconfiguration that avoids the failed function(s) and compensates for any effects thereof.

To address the challenge of understanding software systems, we proposed a reverse engineering tool that semi-automatically extracts requirement goal models from source code [YWM+05]. A goal model for a software system represents its functional and non-functional requirements. The tool automatically extracts a system’s functional re-
quirements from its source code. Extraction of non-functional requirements is semi-automated\footnote{This thesis does not present the details of our reverse engineering tool. Interested readers can refer to [YWM+05] for details}.

Our proposal fits within the context of on-going research on adaptive and autonomic software. Much of this research achieves autonomicity by inserting special purpose code that checks predefined low level system properties and reacts if certain conditions become true [CdLG+08]. Our research, by contrast, specifies correct system behavior in terms of requirement models and/or other system models. Our framework then generate diagnoses given such models and log data. We assume the diagnosability of these software systems.

The monitoring and diagnostic components verify that a system complies with its specification (of its correct behaviors). Our work deals with two kinds of system specifications: (1) requirement models [DvLF93, MLN92] describing the stakeholder goals that systems should fulfill; and (2) statecharts [Har87, Har96] specifying systems’ correct dynamic behaviors. Requirements and statecharts describe different aspects of a system at different levels of abstraction. Requirements describe system’s design time functional and nonfunctional requirements, whereas statecharts describe a system’s dynamic behaviors. Accordingly, when monitoring and diagnosis employ both models, they can identify a wide range of system failures. Thus our framework uses requirements to infer requirement denials (denials of goals and tasks in a requirement model) (Chapter 4), and statecharts to infer denials of states and transitions in a statechart (Chapter 5). These two techniques therefore complement each other.

We assume the existence of traceability links between a system’s source code and specification. We then instrument a software system using AspectJ technologies enabling the monitoring component to monitor the execution of its functions. The monitoring component, at run time, collects and manages log data generated by the instrumented program.
The framework monitors the satisfaction of goals and tasks in a requirement goal model when monitoring and diagnosis makes use of requirements (presented in Chapter 4). We associate these goals and tasks with monitoring switches, preconditions and effects. Monitoring switches are Boolean flags that can be “on” or “off” to indicate whether the framework monitors the satisfaction of their associated goals/tasks. Preconditions and effects are propositional formulas in CNF (Conjunctive Normal Form). The framework infers goal and task satisfaction/denial by analyzing the truth values of the associated preconditions and effects. Monitoring granularity can be tuned according to diagnostic feedback. When failures occur, monitoring granularity can be tuned up to monitor lower level goals and tasks. When this is done, more complete log data are generated and more precise diagnoses can be inferred. If the system runs correctly, monitoring granularity can be tuned down to monitor higher level goals. We generate less complete log data and reduce monitoring overhead. In other words, we generate as much as (or as little as) log data as needed for diagnosis, depending on how successful the application runs. Our work on requirement monitoring and diagnosis is presented in [WMYM07, WMYM09].

When monitoring and diagnosis makes use of statecharts, we verify that a system’s run time behavior complies with a subset of the semantics described in its statechart (presented in Chapter 5). We say subset because our work deals with a (rather small) subset of statechart semantics. Specifically, this subset includes transition firing, and changes in the system due to transition firing. Our work currently does not deal with more complex statechart notations such as concurrency, history, conflicts, priority, non-determinism, hierarchy, or temporal relationships. Much future work is needed to deal with full statechart semantics.

When statecharts are used, the framework monitors successful executions of transitions and satisfactions of domain constraints associated with states. Our framework associates states and transitions with monitoring switches to indicate whether they are
monitored at runtime. It associates monitored states with state condition formulas that describe domain constraints and assumptions. Transitions can be (but do not have to be) associated with preconditions and effects. Added preconditions and effects can further specify transitions’ correct behaviors.

The framework’s diagnostic component analyzes generated log data and identifies failures. Propositional satisfiability (SAT) is the problem of determining whether a propositional formula admits any truth assignments to its literals that make the formula true. Advancements in SAT solver technology in recent years have encouraged SAT-based solutions to software engineering problems. We transform the problem of diagnosing software systems into a SAT problem by encoding a system’s specification and its log data into a propositional formula solvable by an off-the-shelf SAT solver. We use SAT4J [LB05], an efficient SAT solver, to solve the formula and thereby discover possible diagnoses. Our diagnostic component is sound and complete with respect to the diagnoses it generates. Each SAT truth assignment corresponds to a correct diagnosis consisting of one or more system failures. To use a SAT solver effectively, we preprocess log data to generate precise formulae so that our framework scales with the size of its specification.

When failures occur, an autonomic system needs to either repair the failures or reconfiguration itself to bypass the failures. Our work offers repair through reconfiguration if requirement goal models are used as the basis for monitoring and diagnosis (presented in Chapter 6). Our reconfiguration component offers three reconfiguration algorithms (Algorithms 8, 9, and 11) that compute best global and local reconfigurations free of failures. Algorithm 11 is the most efficient out of the three. It returns a best global reconfiguration that contributes most positively to system’s non-functional requirements. It also scales to the size of the goal model. Our work on reconfiguration is presented in [WM09].

The execution component runs compensation actions to restore the system to its previous consistent state and reconfigures the system under the new configuration. We adopt
a model of long-term database transactions to deal with compensation actions [GM87].

Our framework does not offer a reconfiguration/repair mechanism when monitoring and
diagnosis make use of statecharts. We plan to address this limitation in future research.

We illustrate our framework with two medium-sized, publicly-available case studies:
a Web-based email client (Squirrel Mail) [Cas07], and an ATM simulation [Bjo]. Squirrel
Mail is an open source email application that consists of about 70000 LOC written in
PHP. ATM simulation is an illustration of OO design used in a software development
class at Gordon College. This application simulates an ATM performing customers’
withdraw, deposit, transfer and balance inquiry transactions. Its source code contains
36 Java Classes with 5000 LOC. We show that our framework turns the original ATM
software into an adaptive system with self-reconfiguration capabilities.

We evaluate the framework’s performance through a series of experiments on ran-
domly generated and progressively larger specifications. The largest goal model con-
tained 3000+ goals and tasks, and the largest statechart contained 1000 states and 10,000
transitions. The results demonstrate that our approach scales well with the size of the
specification. Therefore, it is feasible to scale our approach to industrial software systems
with medium to large specifications.

This work advances the field in several respects and makes the following major con-
tributions. (1) Our work verifies that a system complies to its requirement goal models.
This is the first SAT-based solution to the diagnosis of the software requirement satisfac-
tion problem that is sound and complete. (2) Our work verifies that a system complies to
a subset of the semantics specified in its statechart. (3) We presented three reconfigura-
tion algorithms on goal models. (4) The proposed autonomic framework is implemented
and evaluated. Experimental results show that our approach scales with the size of the
specification. It is therefore feasible to apply our approach to industrial software systems
with medium size specifications.

The rest of this thesis is structured as follows. Chapter 2 reviews the relevant lit-
Chapter 1. Introduction

We review research areas of goal oriented requirement engineering, autonomic computing, monitoring software systems, AI theories of diagnosis, and reconfiguration. Chapter 3 provides an overview of our autonomic architecture, and describes four software engineering steps needed to prepare the architecture’s input. It also discusses framework implementation details. Chapters 4 and 5 present monitoring and diagnosis using requirement goal models and statecharts respectively. These chapters present necessary preliminaries; describe running examples; present our monitoring and diagnosis approach; give evaluation results; and discuss assumptions made as well as contributions and limitations of our approach. Chapter 6 describes framework’s reconfiguration and execution components. It explains our three reconfiguration algorithms in detail and evaluate them. Chapter 7 presents two case studies conducted in collaboration with an industrial partner, Computer Associate (CA). In Chapter 8, we conclude assumptions made and the limitations and contributions of our framework. We also discuss our future plans. Our work on monitoring and diagnosis is presented in [WMYM07] and [WMYM09]. Our reconfiguration approach is presented in [WM09].
Chapter 2

Literature Survey

2.1 Goal Oriented Requirement Engineering

Since the mid 1970s, Requirement Engineering (RE) was established as a distinct research discipline and its importance has been repeatedly recognized during the past 30 years. Inconsistent, incomplete, and obscure software requirement specifications account for over half of all failures in industrial software development projects [Lam00]. It has been recognized that the cost of recovering from an erroneous design and implementation is far more expensive than the cost of initially conducting comprehensive requirement analysis prior to system development. Early RE research [Ins83] focuses on answering the “what” questions of a system such as “what the system’s requirements are” and “what components the system should have”. Later RE research [Ant96, BK94, DvLF93, KL98, LKP+98, MLN92, RSA98, Yu97] evolved to incorporate modeling systems’ environments and organizational contexts in the RE process and tries to answer the “why” questions such as “why the software is needed” and “why a design alternative is justified”. By exploring these “why” questions, requirement analysts are able to model and analyze the rationalities behind various design decisions and model purposeful systems such as enterprise applications. These research projects that explore the “why” questions of
systems can be further categorized into goal-driven [BK94, DvLF93, KL98, LKP+98, MLN92, Yu97] and scenario based [Ant96, RSA98] approaches. Goal driven approaches model organizational objectives and support conceptualization of purposeful systems. They aim at interpreting requirements prior to system’s development. Scenario based approaches focus on modeling user’s view points and help in modeling purposeful system usage from which system functions can be derived [RP00]. [RP00] nicely summarizes the complementary relationship between goals and scenarios: “scenarios provide dynamic meaning to goals whereas goals provide the intentional setting within which scenarios find meaning”. In the following sections, we review major goal-driven and scenario based requirement engineering methods and projects, with an apparent focus on the goal-driven research projects.

2.1.1 The NFR Framework

[MLN92] proposed a NFR (non-functional requirement) framework for representing and using non-functional requirements and it was further developed in [CNYM00]. Non-functional requirements play a critical role during software development process, yet up till the proposal of the NFR framework, non-functional requirements were largely ignored in software engineering practices - NFRs are generally stated informally, often contradictory and hard to enforce during development process [MLN92]. The aim of the NFR framework is to develop process-oriented techniques for capturing NFRs for the software system, to identify different ways of decomposing NFRs (identify different design alternatives) and to justify design decisions during software development process in terms of NFRs. Different design decisions may have different positive or negative effects on non-functional requirements, and the NFR framework provides techniques to analyze if a certain design decision meets a particular NFR and the NFR framework can help in choosing a best design that satisfies a set of NFRs among many design alternatives.

The proposed framework consists of five major components: collections of goals, links,
methods, correlation rules, and a labeling procedure. A Goal model is an AND/OR graph structure capturing functional (represented as hard goals) and non-functional requirements (represented as softgoals) and their interdependencies of a software system (represented using links). If a hard goal G is AND (or OR) decomposed into subgoals, all (or at least one) of its children must be satisfied in order for G to be satisfied. Unlike hard goals, it’s hard to define clear-cut criteria for a softgoal to be “accomplished” or “satisfied”. Different design decisions contribute, to different degrees, positively or negatively to a particular softgoal. Therefore, the term “satisficing” is used to express that the softgoal is meet to an acceptable degree, rather than absolutely [MLN92]. Therefore if a softgoal G is AND (or OR) decomposed into sub-softgoals, all (or at least one) of its children need to be satisficed (or denied) for to be G satisficable (deniable).

The framework supports three kinds of soft goals: NFR goals, satisficing goals and argumentation goals. NFR goals represent non-functional requirements to be considered; satisficing goals models lower-level operationalized goals that can satisfice NFR goals; and argumentation goals represent design rational for other goals or goal refinements. Besides the AND/OR refinement links betweens goals and their offspring, links can also relate other links to argumentation goals to indicate if an argument contributes positively or negatively to a goal refinement. Therefore, links can also be satisficed or denied. The framework provides goal refinement methods (or just called methods) for guidance of refining goals into their offspring. Correlation rules are provided for discovering implicit relationships between goals and selecting a design that best meets a set of NFRs. Each node in the goal graph has one of four satisfaction labels: satisficed, denied, conflicting (if both satisficed and denied), and undetermined if it is neither. A labeling procedure can be used to determine the status of each node in the goal graph through the assignment of one of the four labels.

Overall, the framework integrates non-functional requirements to the software development process and provides techniques and guidelines for software engineers to use in
capturing a space of valid design decisions, and choosing a best design that meets a set of chosen NFRs. The framework puts non-functional requirements foremost in developers’ mind and justifies design rational in terms of non-functional requirements.

### 2.1.2 i* Framework

i* [Yu97, YL00] is an agent-oriented framework for modeling and reasoning early-phase information systems’ requirements and their organizational environments. Prior to i* framework, much of the RE research assumes the availability of original requirements statements, which express “what” the system should do and give less attention to supporting requirement activities that precede the formulation of the initial requirements. Early-phase requirement engineering addresses the “why” aspects of the system such as finding answers to questions like “why the system is needed” and “what alternatives might exist” prior to system development. i* consists of two main modeling components: the strategic dependency (SD) model and the strategic rationale (SR) model. The SD model is used to describe dependency relationships among various actors in an organizational context and the SR model is used to describe the interests and concerns of actors in the model and how they can be addressed or impacted by different system configurations.

i* centers on the notion of intentional actor and intentional dependency. Actors have intentional properties such as goals, beliefs, abilities, and commitments. Actors depend on each other for goals to be achieved, tasks to be performed, and resources to be delivered. Because of this dependency, actors can achieve goals that are difficult to achieve on their own. On the other hand, actors become more vulnerable if the actors they depend on do not deliver. Actors are intentional and purposeful in the sense that they have to balance between their opportunities and vulnerabilities and seek re-arrangements of their environment to best serve their interests [Yu97].

A strategic dependency model is a network of dependency relationships among actors. Four kinds of dependencies exist that differentiate among the kinds of relationships
between a depender and a dependee: resource dependency, task dependency, goal dependency and softgoal dependency. SD model captures the intentionality of actors and their interdependencies, and provides one level of abstraction for describing organizational environments and information systems. The strategic rationale model are used to provide a more detailed level of modeling by “looking inside” actors to model their internal intentional relationships, and by exploring actors’ interests and rationales. The SR model contains two types of links: task decomposition links and means-end links. Task decomposition links decompose a goal or a task into subgoals and subtasks that form a routine. The means-end links provide understanding about why an actor pursues a goal, needs a resource or performs a task, etc.

The $i^*$ framework makes a distinction between early-phase and late-phase requirement engineering processes and is concerned about modeling the “why” aspects of software systems, business processes, and organizational contexts. $i^*$ associates intentions and objectives to actors which capture various stakeholders’ goals and the purposiveness of many software systems it models.

### 2.1.3 KAOS

The KAOS project provides a requirement acquisition approach with rich and formal semantics [DvLF93]. KAOS stands for Knowledge Acquisition in autOmated Specification [vLDDD91]. Requirement analysis consists of requirement acquisition and formal specification tasks. In the requirement acquisition stage, a preliminary model representing the requirements of the software application is elaborated and expressed in acquisition language. In formal specification stage, the preliminary model obtained is refined and formalism is added for formal proofs and reasoning.

The KAOS approach has three components: the conceptual model, an acquisition strategy, and the acquisition assistant. The conceptual model is a domain independent meta-model that is rich enough to allow any functional and non-functional requirement
to be specified precisely for any composite system. Acquisition strategy defines steps for acquiring requirement models from the composite system that are instances of the conceptual meta-model. The acquisition assistant provides automated support in following an acquisition strategy. The KAOS approach to requirement acquisition involves modeling at meta level, domain level and instance level. At the meta-level, abstractions such as meta-concepts, meta-relationships and meta-constraints etc are modeled. The meta-model determines the structure of requirement acquisition language. The domain level concepts are instances of meta-concepts in the meta-model and they are domain specific. Links between concepts at domain level are also instances of meta-relationships in the meta-model. The instance level refers to specific instances of domain level concepts [DvLF93]. KAOS uses temporal first-order logic to represent time. At domain level, formal assertions can be attached to attributes to express assertions such as “property P holds in the next state”.

KAOS meta-model contains Goal, Constraint, Object and Action meta-concepts. An Object is a thing of interest which can be an Entity, an Event, a Relationship, or an Agent. An entity is an autonomous object in the domain that may exist independently from other object instances, and entities are connected to other objects in the domain through relationships. An event is an instantaneous object that exists in one state only. The Action meta-concept is linked to Object meta-concept by input/output meta-relations meaning that actions take objects as their input and output, and is linked to Event meta-concept through cause/stop meta-link. Actions are associated to meta-attributes: PreCondition, TriggerCondition, and PostCondition. An action may be applied if its precondition is true, while it must be applied when its trigger condition is true. Therefore, an action’s trigger condition logically implies the action’s precondition. Actions’ Postconditions are true after the action is applied. Modify and Inspect are two specializations of Action. Modify actions change state of the world, while Inspect actions do not. An agent can be a human, organizational units, physical devices, or software, who
is capable of performing actions.

KAOS’ definition for a Goal is “a goal is a nonoperational objective to be achieved by the composite system” [DvLF93]. Nonoperational means the goal can be not be achieved by actions available to an agent. Meta-concepts Goal and Object are linked through concerns meta-relationship, and at the domain level goals are associated to the objects they refer to. Five patterns are defined for goals based on their formal definitions: Achieve, Cease, Maintain, Avoid, and Optimize. A goal can be a system goal or a private goal. System goals are “application specific goals that must be achieved by the composite system”, while private goals are “agent-specific goals that might be achieved by the composite system” [DvLF93]. Therefore, goals can be conflicting with each other. Priority meta-attribute was introduced to help with Conflict resolution, and a Wish meta-relationship was introduced between Agent and Goal. System goals can be categorized into: SatisfactionGoals, that are concerned with satisfying agent’s requests, InformationGoals, that are concerned with keeping agents informed, RobustnessGoals, that are concerned with keeping automated agents robust, ConsistencyGoals, that are concerned with maintaining consistency within the composite system, SafetyGoals, that are concerned with maintaining agents in safe states, and PrivacyGoals that are concerned with keeping visibility to agents restricted. High level, informal goals need to be refined to lower level and more formal subgoals, and goal refinement is represented by Reduction meta-relationship in the meta-model.

Goals are made operational through constraints. Constraints are formulated in the meta-model with objects and actions available to some agents in the system and can be accomplished and controlled by one of the agents. Thus, it can be viewed that constraints “implement” goals.

KAOS requirement acquisition strategy amounts to traversal of the meta-model and it involves the following steps: identify goals and concerned objects; identify agents and their capabilities; operationalize goals into constraints; refine objects and actions; derive
strengthened objects and actions to ensure constraints; identify alternative responsibilities; and assign actions to responsible agents [DvLF93]. The KAOS modeling philosophy is that modeling involves a variety of concepts such as goals, agents, constraints, objects, and actions. The KAOS meta-model can be used to support reuse of generic domain modeling patterns.

KAOS does not provide as rich taxonomy for non-functional requirements as that in [MLN92], yet it is a well developed requirement acquisition methodology with a solid formal framework to aid in the acquisition and modeling of correct and complete requirements for a software system.

2.1.4 GBRAM

Goal-Based Requirements Analysis Method (GBRAM) [Ant96, Ant97] emphasizes on goal identification from various sources of information. GBRAM is useful in identifying goals when they are not already given and evolving goals when the requirements change. Therefore, the GBRAM approach consists of two main activities: goal analysis that concerns exploration of documentation for goal identification followed by goal classification, and goal evolution that concerns evolving (elaborating and refining) goals when requirements change [Ant96].

Goal analysis is the process of identifying goals, classifying maintenance and achievement goals, and identifying agents and stakeholders of the system by determining what agents are responsible for achieving or maintaining identified goals. Goals evolve because stakeholders change their mind or at least goals’ priorities change over the years. Goal evolution consists of goal elaboration and goal refinement.

Goal elaboration can be assisted by identifying goal obstacles, analyzing scenarios and constraints, and operationalizing goals. Goal obstacle analysis is the process of identifying the possible ways in which a goal can fail and identifying possible exceptional cases. Scenarios have proved to be helpful in requirements elicitation in a number of
ways: to elicit requirements in various situations, to help discovering exceptional cases, to understand the needs of stakeholders through scenario prototyping, and to create context for design [RP00]. When goal priorities change, scenarios help the evolution of new goal priorities. Goal operationalization can further assist in goal elaboration process because it looks at the all possible ways of how a goal can be achieved or blocked.

The goal refinement process removes redundant goals, reconciles synonymous goals, merges goals into subgoal categories, and operationalizes goals. During goal refinement and merging of achievement goals, identification of their precedence is required for the purpose of identifying goal ordering and goals’ pre and post conditions. Goal precedence ordering specifies which goals must be fulfilled before other goals can be fulfilled. The main purpose of goal precedence ordering is to order and refine goals, rather than for analyzing goal and agent strategic dependencies as that in the i* framework [Yu97].

Anton summarized in [Ant96] lessons learned from designing GBRAM: multiple document sources yield better goals and give a rich picture of requirements, identification of stakeholders provide insight into their different viewpoints that may lead to potential conflicts, categorizing goals suggests goal operationalizations, scenario and constraint analysis may help to discover new requirements and constraints, and the identification of goal obstacles helps to discover exceptional cases.

2.1.5 Other Approaches

Bubenko et al. proposed the F3 (From Fuzzy to Formal) [BK94, BRLA94] enterprise modeling project where different modeling “worlds” or sub-models are used. The F3 project aims to address the challenge of proceeding from the initial informal fuzzy statements of requirement to a formal and precise definition of system requirements that are agreed upon among different stakeholders. Requirement specification is represented as a structured description of eight interrelated sub-models, each representing a different “world” or concern of requirement specification. Figure 2.1 presents F3’s sub-models.
Chapter 2. Literature Survey

The objectives sub-model describes the “why” and the long term objectives of the enterprise system. It is a graph with goals and problems as nodes connected with “motivates” relationships. The concept sub-model provides definitions for the “things” in the universe of discourse that are modeled. The actors sub-model defines the actors in the domain and their relationships with actives and objects. Actors can be people, groups, roles, or physical devices. Actors are stakeholders of the system that can achieve goals. The activities sub-model describes the processes and tasks of the enterprise. Activities can be performed by actors in the actor sub-model to achieve goals in the objective sub-model. Functional and non-functional requirements of the information system are represented in the information system requirements sub-model. The process of requirement specification is not a simple, linear process and many different alternative designs may exist. Different possible configurations of design and specification are modeled in the configuration sub-model [BK94]. Bubenko argued that the reason to separate requirement specifications into eight sub-models is that there is a need and benefit to separate between concerns in requirement engineering because each sub-model represents a particular concern that needs to be addressed with a different set of expertise.

The F3 project offers a set of interrelated models to address both the “why” and “what” questions of designing an enterprise system. The set of components and relationship types in the sub-models reflects the motivation and rationale for designing an information system. Enterprise modeling was further refined in the Enterprise Knowledge Development (EKD) method to support change management [KL98, LKP+98, RNG97, RLKN99]. EKD is an integrated collection of methods, and tools that support the process of analyzing, planning, designing and changing business requirements.

Pure goal based requirement engineering methods assume that the systems to be designed are purposive and are constructed with goals in mind [RP00]. In reality, goals are not given and the task of acquiring goals presents an on-going challenge. An alternative to pure goal modeling is scenario-based approach. Scenarios can be useful for goal elici-
Figure 2.1: The Sub-Models of the F3 Approach [BK94].

tation because it is generally easier to identify scenarios than to discover goals, and goals can be made more explicit once their associated scenarios are identified. Scenarios can be descriptive, explanatory, or exploratory [RP00]. Descriptive scenarios give descriptions of system’s processes, and objects. Explanatory scenarios provides rationales for the system of interest. Exploratory scenarios are useful when different solutions exist to fulfill the system’s requirements, and the scenarios can assist in the process of choosing a best solution to best meet system requirements. Scenarios can have different contents: scenarios can capture the behavior information of the system identifying actions and activities; they can describe objects and their attributes in the domain; or they can represent organizational and stakeholder information. Rolland et al. proposed a scenario based requirement engineering project called CREWS [RSA98]. The project aims to guide the requirement elicitation process and to discover goals from scenarios through
interleaved goal modeling and scenario authoring. A bi-directional goal-scenario coupling is created between goals and scenarios. The goal-scenario coupling can be exploited in the forward direction, from goals to scenarios: when a goal is discovered, a scenario can be authored for it. Once a scenario has been authored, it can be explored to yield goals (the reverse direction). In the forward direction, the goal-scenario coupling facilities goal operationalization whereas in the reverse direction it promotes goal discovery. The goal discovery process is centered around the notion of Requirement Chunks (RC) (which are pairs of goals and scenarios), and the discovery of RCs. [RAC+98] proposed a scenario classification framework for classifying scenario based approaches based on four dimensions: form (in which form is a scenario expressed), contents (what is the knowledge expressed in a scenario), purpose (why using a scenario), and life cycle (how to manipulate a scenario). Scenarios are developed for different purposes, with different contents, and are expressed with different notations. The framework helps us to understand and clarify existing scenario based approaches.

2.2 Autonomic Computing

Autonomic computing [Hor03, GC03, KC03] is an IBM initiative and a rapidly growing research area that aims at creating more self-managing computing systems and reduce software maintenance cost and shift management complexity from administrators back to software systems themselves. The word “autonomic” comes from the human body’s autonomic nervous system. The autonomic nervous system is able to automatically carry out a wide range of low level body functions for a human to adjust to his environment without the human’s conscious recognition [Hor03]. For example, when a person runs for a bus, the brain does not need to tell the body to raise heart rate to pump more blood or to sweat to reduce body temperature. The same idea applies to software systems - autonomic systems manage themselves at lower level in accordance with policies and
goals provided by humans without conscious recognition of system administrators.

Given high level goals and policies from system administrators, autonomic computing systems perform four self-management tasks: self-configuration, self-optimization, self-healing, and self-protection. The autonomic system is able to automatically reconfigure itself when new components come into or are removed from the system (self-configuration); able to continually tune its parameters for optimization (self-optimization); able to monitor, analyze, and recover from faults and failures when they occur (self-healing); and able to protect itself from malicious attacks (self-protection). All the four self management tasks require the system to consciously monitor itself and its environment, analyze monitored data to learn what it means, plan for changes to be made if any, and execute the plan. This Monitor, Analyze, Plan, and Execution loop is known as the MAPE loop for autonomic systems.

Autonomic systems consist of interactive collections of autonomic elements, which are individual autonomic components that manage their own internal behavior and resources as well as interacting with other autonomic components and humans in accordance with goals and policies provided by humans. Figure 2.2 shows the structure of an autonomic element [KC03]. Within each autonomic component, there is one autonomic manager and one or more managed resources, that can be physical devices or software.

Autonomic manager manages managed element and interacts with other autonomic elements. Autonomic manager monitors its managed element and its environment, analyzes monitor data to make sense of what it means and evaluates if any adjustments are needed. If adjustments are needed, the autonomic manager plans for actions and changes necessary and executes the plan. The autonomic system is likely to evolve when designers gradually increase the sophistication of autonomic managers.

Correctness and completeness of the knowledge an autonomic manager has is critical as it is the only knowledge an autonomic manager has during the monitor, analyze, plan and execution loop. While autonomic systems greatly reduce chances for human errors,
they will also magnify any errors humans make in specifying goals and policies [KC03]. Goal models [DvLF93, MLN92] are useful in correctly and completely specifying high level goals and knowledge for the autonomic systems. For any autonomic system, traceability links are needed to link between monitored system and its high level model (describing goals and policies). Extracting and maintaining traceability links is beyond the scope of this thesis. There is much research done in this area. For example in [Teo01], Teoh presented research on extracting structural information from object code.

Monitoring and analyzing monitored data are essential features of autonomic elements. Autonomic systems need to continuously monitor themselves and analyze monitored data to ensure that they are meeting their requirements and objectives. One of the main challenges of designing autonomic computing paradigm is therefore to design autonomic managers that do not cause too much overhead and compete for too much resources with managed elements. In the following sections, we give an in-depth survey
on current literature and research projects conducted in the monitoring and diagnostic reasoning research areas.

### 2.3 Monitoring Software Systems

Monitoring can be defined as the process of dynamic collection, interpretation, and display of information concerning the behavior of a software system [JLSU87]. Monitoring constitutes a necessity for any software operational system. Monitoring can be useful for many purposes such as program understanding, debugging, management, visualization and compliance to system requirements.

A two level conceptual model underlines monitoring systems [Rob05c]. The design time model is a requirement model that represents system requirements and other artifacts such as policies the system should adhere to prior to system development. The run time model represents system execution, examples include program trace, work flow diagrams, etc. Monitoring systems construct their run time models by capturing, storing, composing, analyzing, reacting and visualizing events. The objective of requirement monitoring is to verify monitored system’s (also called the target system) run time behavior (run time model) is in compliance with its original requirements (design time model). The difficult research questions such as “what should be monitored”, and “where should the program be instrumented” are answered by analyzing the design time requirement model. However, in many cases, system’s design time model was never fully developed or was set aside and system’s run time model represents the entire monitoring system. Designers of the monitoring systems have to redefine monitoring requirements that are loosely based on the original design requirements that they remember. We refer to these kinds of monitoring systems, whose monitoring requirements not derived from their design time requirement model, simple software monitoring systems. Commercial monitoring systems are examples of software monitoring - they focus on tracing low level system
events and processes. The satisfaction of their high level goals, requirements and business objectives is not monitored and verified at runtime [Rob05c].

There has been considerable effort in developing monitoring components with overlapping functionality for different kinds of software systems from both academia and industry, but none provide a comprehensive solution [Rob05c]. The scope of this section is to provide an overview of representative software and requirement monitoring systems and research projects, and discuss important issues relating to monitoring.

2.3.1 Software Monitoring

Software monitoring provides a simple form of requirement monitoring. The monitored software is instrumented with insertions of information statements to produce log files that are later analyzed by software developers to detect requirement failures.

Papadopoulos [Pap03] carried out work on monitoring and diagnosis using statecharts and fault trees. Monitoring expressions are associated to systems’ statecharts, and they specify what abnormalities and failures to watch for at runtime. When any of the monitoring expressions becomes true, the diagnostic component traverses through the fault tree to identify failure components. The main difference between our respective approaches is that in Papadopoulos’s work, failures are expressed manually and explicitly through expressions, rules, and/or fault trees. Programmers need to predefine all expected failure scenarios and conditions. In our work, we use a model (requirement model or statechart) that specifies the correct/intended behaviors of a system. We can automatically infer diagnoses given this model and log data.

[MSS96] presents a configurable event service for distributed systems. The presented framework allows primitive events, that correspond to lowest level system activities, and composite events that can be dynamically composed from primitive events using temporal constraints and composition operators. A Generalized Event Monitoring (GEM) [MSS97] notation can be used to specify event specification which can be interpreted by event
monitors. The event specification contains individual rules that define events of interest and the actions they trigger if detected. These rules can be dynamically enabled, disabled, inserted, and deleted. Event monitors send primitive and composite events to event disseminators that then distribute events to event consumers. The presented architecture focuses on monitoring distributed systems and new event monitors and disseminators can be dynamically created on distributed nodes using Rigis/Darwin [MDK] configuration environment. This dynamic reconfiguration capability allows monitors and disseminators to be located close to where the events are generated, therefore, reduces added network traffic caused by monitoring.

The Meta [MCWB91] system is a collection of tools for managing distributed software applications. In a Meta managed distributed application, the Meta system resides in between application’s management layer and its functional layer with well defined interfaces. Separating application’s management layer and its functional layer makes modifying application’s management layer easier and Meta provides an abstract view of the application to the management layer. The management layer is referred to as the control program and is programmed in a rule-based language called Lomita. Using Meta to manage a distributed system involves three steps: instrumenting the program, describing program structures, and expressing policy rules. The program and its environment are instrumented with sensors and actuators that provide an interface between the program and the control layer. Sensors are functions that return values of application’s state such as application’s CPU usage, and the total throughput of the application, etc. Actuators can be used to change application’s states such as changing a process’s priority and migrating a process to another machine. In the second step, the programmers describe the application’s structure using Lomita’s object-oriented modeling facilities. Components in the application and its environment are modeled by entities, following entity-relationship terminology. This data model is then referenced by the control program in the final management step. The control program, maybe written as a Lomita
script, defines the intended behavior of the system and can make direct calls to the sensors and actuators in the program. For example, rules like “when condition do action” are defined at the control layer. The monitoring provided by Meta is limited to low level system activities such as CPU usage and system processes. Moreover, the programmers need to define all (expected) conditions that may occur and upon which rectifying actions should be triggered. Therefore, unexpected and undefined exceptions and conditions are left un-monitored.

In [SM06] a collection of software and hardware performance monitors that are integrated with existing software and hardware mechanisms for COTS based systems are proposed. Depending on the monitored system and its objectives, performance monitoring can happen at three levels: event monitoring at operational level and performance monitoring at low and high architecture level. The low operational level relates to logging system activities which deliver data on system hardware, software or configuration etc. High level architecture monitoring monitors system resources availability and usage issues such as processors, I/O devices and network ports. Low level architecture monitoring relies on build-in hardware mechanisms to detect errors at finer level such as CPU exceptions, memory address violation, etc. All the logging and monitoring activities rely on available software and hardware. The presented work monitors issues related to performance and system errors, rather than monitoring application related events or satisfaction of system’s requirements or constraints.

There are many commercial monitoring tools that monitor business processes and evaluate monitored applications’ performance. Microsoft Operations Manager (MOM) for BizTalk Server [MOM] is a typical example. MOM can be used to monitor Microsoft BizTalk Server’s messaging statistics, message queuing queue size, and sizes of the tracking and BizTalk Orchestration, etc. MOM captures service-level events and defines rules to automate resolution responses that include raising exceptions and alerts. IBM’s WebSphere Business Monitor [IBM] is another example which is capable of monitoring in
flight business processes, business performance of active processes, business situations. It is also able to take actions, gather business intelligence from collected data, and create dashboards for visualization. The WebSphere monitor is based on Common Event Infrastructure (CEI) which is IBM’s implementation, transmission, and distribution of a wide range of events based on Common Base Events (CBE). Other software monitoring tools include NetLogger [TG02], log4j [Apa], Open Group’s Enterprise Management Forum’s Application Response Measurement (ARM) API [ARM], the CORBA event service [Gro], the JINI distributed event service [JIN], and the ECHO event service [ECH]. Most of these monitoring tools, commercial or not, lack the monitoring abstract layer that captures the system’s high level requirements. As a consequence, monitoring provides low-level alerts that have little correspondence to the original business requirements.

2.3.2 Requirement Monitoring

The key difference between software monitoring and requirement monitoring approaches is that in requirement monitoring, the objective is to track system’s runtime behavior for deviations from its requirement specification and therefore to monitor system’s satisfaction of its design time requirements. The common approach that supports requirement monitoring at run time assumes a system provider provides requirement specification that the system is implementing, identifies the set of requirements to be monitored at run time and generates monitoring specifications from identified requirements. A combination of events or conditions is also derived from these specifications, that when observed at run time implies denial (or satisfaction) of system’s requirements. Compares to software monitoring, requirement monitoring brings together system’s original design time requirements and run time execution monitoring so that the satisfaction of stakeholders’ requirements and objectives is monitored and verified. According to [RPV03], requirement monitoring differs from exception handling in three ways: (1) it considers composite events instead of a single thread of execution, (2) it links system’s runtime behavior to its
design time specification and provides traceability, and (3) it provides more information
than exception handling because log data generated by monitors may provide information
on how to reconcile system’s run time behavior to its requirement [RPV03]. In the
following sections, we review in detail the research projects lead by Fickas, Feather et.
al [FF95, FFLP98] and Robinson [Rob02, Rob03, Rob05a], as well as giving an overview
of other representative research projects in the requirement monitoring research area.

FLEA Based Requirement Monitoring

Fickas and Feather et. al. proposed a run-time technique and environment for monitoring
requirements satisfaction of a software system [FF95, FFLP98]. Requirement specification
language is KAOS [DvLF93]. In [FF95], Requirement - Assumption - Remedy tuples
are identified. For each requirement represented in KAOS that is of monitoring interest,
the authors identify assumptions made on the environment and users that have to be true
in order for that requirement to be met at run time. Remedies to be executed in case of
assumption violation are also identified at design time. At run time, satisfaction of these
assumptions made on the environment and the users is monitored, and remedies (which
generally involve either parameter tuning or switching to an alternative design) would
be executed if these assumptions are violated. The authors give for example, in a license
manager software, if the satisfaction of requirement “serve license fairly to individual
users” is to be monitored at run time, one of the assumptions made on the environment
could be “longest waiting users gets the license first”. If at run time this assumption is
violated, its associated remedy “have license manager maintain queue of waiting users”
would be executed [FF95].

In [FFLP98], this requirement monitoring framework is further extended with FLEA
(Formal Language for Expressing Assumptions) [CFNF97]. FLEA language provides
capabilities for expressing temporal combinations of events. KAOS requirements and
assumptions are identified, and negations to assumptions are identified as “breakable
assertions”, which are monitored at run time. Monitor specification and monitoring code is generated automatically by FLEA using these breakables assertions. The running system maintains a specialized database for receiving events and distributing notifications of occurrence of event combinations. A mapping between events and database triggers is maintained and at run time. If FLEA captures a temporal combination of event occurrence, its corresponding trigger action (the remedy) would be activated. Remedy could either be parameter tuning or switching to a different alternative. Therefore, the system becomes adaptive in the sense that it performs predefined self-reconfiguration tasks in case of requirement violations [FFLP98].

[FF95] and [FFLP98] focus on monitoring the relevant changes to the system’s environment and satisfaction of system’s non-functional requirements rather than monitoring satisfaction of system’s functional requirements. Therefore their approach is suited for software that has build-in monitoring and diagnostic component for checking its compliance to its functional requirements. One of the contributions of Fickas and Feather’s work is to answer the research question of “what to monitor” in a distributed software system in a dynamic environment by associating system’s requirement specification to environment assumptions that are monitored at run time. [CFNF97] discussed monitoring desiderata which includes “flexibility and convenience”, “automatic compilation” of monitors and monitor specification, “applicable to black-box systems” and monitors should be “incremental” are valid guidelines for designing requirement monitors today. In fact, the paper [FF95] that was published in Requirement Engineering (RE) conference in 1995 was awarded the “most influential paper” award in RE 2005 ten years later.

Fickas et. al. reported a case study in year 2002 [FBM02] on requirement monitoring using a different line of approach. In [FBM02], a case study was conducted on monitoring the success of “asking a friend for a ride” task taken up by people with cognitive impairments. This approach uses a timeline editor [SHE01] that gives a graphical view of events that happen in time line sequence as well as an automata representing legal
state transitions with each legal event and a text representation that is later translated into an event tree to be monitored. Proposed tool Emu (Event Monitoring Utility) serves as the back-end of timeline editor and translates text representation of events provided by timeline into event tree in XML format. The initial “ask for a ride” request can be viewed as the trigger event that triggers the event tree to be monitored. If the event tree is executed successfully, the requirement is satisfied, otherwise the requirement is denied.

ReqMon

Robinson presented a requirement monitoring framework, ReqMon, using three program architectures: assertion and model checking [Rob02], SQL query [Rob03], and Event Condition Action (ECA) rules using Jess rule-based system [Rob05a]. In [Rob02], a combination of assertion and model checking were used for requirement monitoring. Suspect conditions are conditions that are entailed by negations of requirement. In other words, if requirement of interest is defined, suspect conditions would be observed at run time. Therefore, these suspect conditions are to be asserted if observed. Assertion requires these weakened, suspect conditions to be specified correctly and completely in order for all requirements to be monitored, the task of which can become intractable. Model checking is used as a complementary approach to determine if a requirement could fail in the future without the need to explicitly specify suspect conditions. Model checking is exploration of all possible states in a state-based model. Analysis can prove if certain conditions will, or will not occur, if the program and its current state of execution can be represented as state-based model [Rob02]. ReqMon translates requirement specification into an automata based model and a model checker was used to exhaustively check all the states of the model.

ReqMon can also use SQL query to monitor composite web services to support Business-to-Business transactions [Rob03]. Obstacles are derived from KAOS requirement specification language using techniques presented in [LL00]. Typically, obstacles
are negations of the requirement. For example, if the requirement is: “the system shall not maintain location information” for privacy reasons, obstacle to this requirement would be: “system storage of location information”. For each obstacle, an analyst can either remove it by modifying requirement or monitor it at run time using presented monitoring framework. Monitor specifications are derived from monitored obstacles, and an agent is assigned by the analyst to monitor each obstacle. Aggregate obstacles can be monitored using relation aggregation function that can be found in SQL counterparts at the run-time level. The monitoring framework contains two levels: “the monitor implementation language that defines the types of events and relationships that are understood” and the “distributed monitor architecture that defines relationships among the monitors, agents, and hosts of a distributed system.” The monitoring framework alerts the analysts for occurrence of any primitive or aggregate obstacles.

In [Rob05a], ReqMon was implemented as a rule-based system and was shown to be more efficient in terms of monitoring overhead. As in [Rob02] and [Rob03], obstacles are derived from KAOS requirement specifications and these obstacles are monitored at run time. Because KAOS uses temporal logic which is difficult to define and understand, ReqMon adopts the nine temporal patterns (universal, absence, existence, response, precedence etc) and five variable scopes (global, before event R, after event Q, between Q and R, and After Q until R) defined by Dwyer [DAC99] as its requirement language to specify temporal relations. The following steps are involved in the monitoring framework: (1). Instrumented target program (event source) outputs program executions trace information; (2). An event transport framework moves collected events to event sinks (where all the events are); (3). Relevant Events are moved from Event sinks to ReqMon server; (4). As Event arrives, monitors in ReqMon uses Event-Condition-Action (ECA) rules (in forms of “if conditions then actions” statements) to update satisfaction status of a requirement based on if the event is a satisfaction or violation event. ECA rules specify what actions the monitor should take after interpreting an event. The action taken could
be either execution of a script or updating status of requirement satisfaction. Much of the ReqMon’s managing of events and their transportation are handled by commercial tools. For example, event management is implemented using Microsoft Windows Management Instrumentation (WMI) and Enterprise Instrumentation Framework (EIF). Monitor implementation is done in Jess rule-based systems, where specification (temporal patterns) of requirement and monitors are translated into Jess rules. ReqMon can also be configured for statistical-based monitoring where ReqMon accumulates information about reoccurring patterns using data mining techniques.

In summary, ReqMon integrates goal-directed obstacle analysis, monitor deviation, and requirement monitoring [Rob05c]. ReqMon provides real-time feedback on requirement satisfaction and compliance. Robinson’s requirement monitoring work is most similar to the approach of Feather et. al [FFLP98, FF95] because both use KAOS as requirement specification language and both use a special database for storing and processing events. The ReqMon approach builds on that of Feather et. al. by providing temporal logic primitives and their combinations for monitors and ReqMon relies on standard commercial tools whereas Fickas and Feather’s approach doesn’t.

**ZM4/SIMPLE**

ZM4/SIMPLE [HKM+94, Moh90, Moh91] is a monitoring environment that allows monitoring and analysis of system’s functional behavior and performance in distributed systems. System requirements are captured as program behaviors in an event-based formal functional model using graphs, Petri-nets or queuing model. Monitoring model (representing monitoring requirements) is derived from (and is a subset of) the functional model. The functional model explicitly models the functional interdependence of system activities. Important artifacts to be monitored are represented as events in both the functional and the monitoring model. Therefore, the two models share the same sets of events and the monitoring model forms the basis of instrumentation. Model driven instrumen-
tation guarantees a one-to-one mapping between the model events and the monitoring events. The tool AICOS was developed for automatic instrumentation of C software. Model driven instrumentation is the foundation for model-driven monitoring. Instrumented software generates and records event trace during execution. The set of all legal event sequences is defined by the functional model and at execution time recorded event sequence is checked against this predefined pool of legal event sequences. The monitoring architecture’s two main components are ZM4, a distributed hybrid monitoring system, and SIMPLE, a performance evaluation environment.

ZM4 (abbreviation for German “Zahlmonitor 4”) monitoring system is structured as a master/slave system with a control and evaluation computer (CEC) as the master and an arbitrary number of monitor agents (MA) as slaves. The CEC is the host of the whole monitoring system and it controls the measurement activities of MAs. Each MA is equipped with up to four dedicated probe units (DPU) and each DPU monitors any number of system objects. ZM4 has a distributed architecture and allows an arbitrary number of monitor agents. ZM4 has a global clock that provides basic mechanism for obtaining a consistent global view of events generated on different distributed nodes.

SIMPLE (Source-related and Integrated Multiprocessor and computer Performance Evaluation, modeling and visualization Environment) is a tool environment designed for performance evaluation for arbitrarily formatted event traces with a modular structure and standard interface. SIMPLE’s first evaluation step is to generate a global event trace from several independent local event trace files (tool MERGE) to obtain a global view of the whole object system. The next step is trace validation where SIMPLE checks if the measurement data (traces) is generated correctly. Tool CHECKTRACE was developed to perform simple validation tests on an event trace. Tool VARUS enables the user to specify rules for validating event trace and performing more detailed and application specific validation checks as assertions in a formal language. Reports are generated containing detected errors. Trace visualization is realized through tools LIST, SMART,
and VISMON that offer textual and graphical visualizations of event traces.

**Other Requirement Monitoring Projects**

In [SM04] and [MS04], an event calculus based monitoring framework was proposed to monitor the requirements of service based systems that are dynamically composed from autonomous web services. The proposed approach builds upon integrity checking in temporal [Cho95] and temporal deductive database [Ple93]. Requirements to be monitored are specified in forms of behavior properties of the system, assumptions on the behaviors of the system, and its constituent services. System’s behavior properties are automatically extracted from BPEL4WS specifications and they are expressed in event calculus. The monitoring framework captures three kinds of deviations from the requirements: inconsistencies between system’s recorded behavior and its assumption; inconsistencies between system’s behavior property or assumption and system’s expected behavior; and when a behavior property is satisfied by the recorded system behavior but are violated by its expected behavior [SM04, MS04]. The proposed framework addresses complications that arise from monitoring service based systems such as interactions between services, that the other requirement monitoring approaches lack.

In [GRS94], Girgensohn et. al. proposed to use expectation agents to monitor end users’ working with prototype systems, and the framework reports mismatches between developers’ expectations of system usage and the actual recorded user behavior and system usage. Requirement specification is therefore “encoded” in expectation agents that know about set of tasks the system supports and the sequence of legal user actions expected by developers. In case of a mismatch, the agent may perform the following actions: (1) notify the developer, (2) provide the users with an explanation of developers intended rational of the system, and (3) solicit a response/comment from the user that will be passed back to the developer. Expectation agents initiate communication between developers and end users, and offer a complementary approach to the more expensive
(but richer) traditional developer-end user face-to-face interactions. Expectation agents consist of a trigger, a query, and an action. The trigger conditions define when the agent should become active and monitors users’ behaviors. The query defines any information, typically an object, that must be located before an action can take place. The trigger and query together decide if an action should be taken. To reduce the burden of requirement elicitation such as specifying expectation agents, programming by demonstration method may be useful. The developer can demonstrate how a task should be performed, and the recorded sequence of actions taken are analyzed to form legal action patterns that are used by the agents. Contribution of the proposed approach is that it captures requirement deviations between different agents.

Peters et al proposed requirement-based monitors for real-time systems [Pet00, PP00]. The authors made the distinction between internal monitors, where monitors observe directly values of software system’s input and output variables, and external monitors, where the monitors observe vector functions of time representing system’s controlled quantities [Pet00]. As was demonstrated in [Hen80], system behavior requirement can be effectively described by values of controlled qualities for a period of time [Pet00]. System requirement specification is modeled as a finite state automaton whose states represent equivalence classes of system executions and edges represent acceptable system events. Events are detected by identifying value changes of variables that make the system transit from state to state in the automaton.

MOP (Monitoring Oriented Programming) [CDR04] is a formal framework for software development that integrates formal specification and implementation. System specifications are inserted into programs as annotations. Monitors can be automatically synthesized from formal annotations and are integrated into proper places in the program. Monitor configuration attributes are specified to describe different monitoring needs for the application under consideration. Violations of specifications can trigger user defined notification/recovery code such as outputting messages or raising exceptions.
The Monitor and Checking framework for Java programs (Java-MaC) [KKLS01] is another example of requirement monitoring. Java-MaC framework checks a target system’s correct runtime execution against its formal requirement specification. System’s requirement model is a formal requirement specification that contains two layers: low-level specification that corresponds to extracted trace information from instrumented code during its execution, and high-level specification that corresponds to composite and high-level events the monitors receive at run time. The program is instrumented at executable (bytecode) level. Run time components such as filter and run time checker are generated from low-level and high-level specifications respectively. A filter is generated from low level specification and is inserted to the executable code for generating changes in state information that can be used by an event recognizer. An event recognizer detects events and sends them to the runtime checker. The runtime checker determines whether or not received event execution sequence satisfies the high-level requirement specification. Java-MaC bridges together formal verification and testing, and provides a light-weight formal method solution in runtime monitoring. The drawback to Java-MaC is that the event recognizer needs to receive and filter events in the same order as they are generated because of its requirement language limitations.

Similar but less comprehensive research projects to Java-MaC include a Lightweight Architecture for Monitoring (ALAMO) [JZTB98], JASS [BFMW01], Java Run-time Timing constraint Monitor (JRTM) [ML97a, ML97b], Java Event Monitor (JEM) [LM99], ANNA [SM03] and Time Rover [Cor03]. The ALAMO [JZTB98] architecture consists of (1) an automatic instrumentation mechanism for C sourcecode, (2) abstractions for event selection, composition and manipulation, (3) an execution model, and (4) an access library that allows the monitors to access target program states. Events are generated by instrumentation of C sourcecode. Monitors filter and receive broader or narrower range of events they are interested in by specifying and modifying event code masking. ALAMO does not provide a high-level requirement specification that describes at the
high level goals and requirements of the system.

Design by Contract (DBC) [Mey00] is a software design methodology that allows specifications to be associated with programs as assertions and invariants which are then compiled into time checks. JASS [BFMW01] is a DBC based approach for Java. JASS is a pre-compiler that supports assertion to Java programs and allows runtime checks of specification violations. Requirement specifications are annotated into Java programs in forms of preconditions, postconditions, and invariants. JASS translates these annotations to pure Java programs so that violations to requirements are indicated by Java exceptions. Pre and post-conditions specify the valid states before and after method invocations respectively, and invariants specify allowed global states of a class. Requirements that can be monitored by JASS are therefore low level requirements that can be mapped to implementations.

JRTM [ML97a, ML97b] aims to detect timing constraint violations in Java programs. The requirement specification language is Real Time Logic (RTL) [JG90] and it defines the timing constraints to be monitored. “A timing constraint is an assertion which relates the time of occurrences of different event pairs” [ML97a]. Manual instrumentation to the appropriate places in the software where events will happen is required.

JEM [LM99] is a CORBA like event-mediator system that accepts and monitors composite events specified in Java Event Specification Language (JESL). Besides supporting event publishing/subscription capabilities supported by CORBA event model, JEM also detects composite events by detecting the satisfaction or violation of timing constraints. Event consumers provide composite event specifications (can be thought of as the requirement model) that should be monitored at run time. Once such specified composite events are detected, they are sent to event subscribers. JEM architecture includes a composite event compiler that compiles composite events, a composite event monitor, and an event subscription server that interacts with event consumers and providers.

ANNA [Luc90] is an Ada language extension that includes the capabilities for formally
specifying intended Ada program behavior, so that well-defined formal methods could be applied to Ada programs. [SM03] presents a methodology that continuously monitor and verify the consistency between Ada programs’ execution with system’s requirements specified in ANNA. Programs are annotated with constructs from ANNA requirement specification, which are then translated into runtime checking code that are inserted into the programs. Calls to the checking code are inserted into the program at places where potential inconsistencies may arise [SM03]. Diagnostic information is provided if inconsistencies occur.

Time-rover [Cor03] Inc. provides many tools that enable run-time verification of requirements compliance. TemporalRover [Dru00] is a specification based verification tool for C, C++, Java, Verilog and VHDL applications. System requirements are specified using Linear Temporal Logic (LTL) and Metric Temporal Logic (MTL) assertions and are inserted to the application source code as comments. Temporal Rover generates executable code from the assertions which is then linked to the application source code. At execution time, software application’s behavior is validated against its formal temporal specification requirement.

### 2.3.3 Intrusiveness of Monitoring Systems

In the previous sections, we categorized monitoring systems into software and requirement monitoring categories based on what they monitor. Monitors can also be categorized by their degrees of intrusiveness on monitored systems. Intrusiveness is the effect that monitoring may have on the behavior of the monitored system as a result of monitors sharing resources, such as memory and network channels, with the monitored system [MSS93]. Impact of intrusive monitors on monitored systems may include degradation of system performance, increased execution time, and incorrect global ordering of events. One of important parameters by which intrusiveness can be measured is the way in which the monitoring system detects occurrences of events. Using this measurement, monitors
can be grouped into three categories: hardware monitors, software monitors, and hybrid monitors [MSS93]. A distinction needs to be made between “software monitors” in this context, where we categorize monitors based on their intrusiveness, and “software monitoring” in the previous sections where we categorize monitors based on whether they monitor system’s requirements. There is not a necessary overlap between the two “software monitors” categories.

**Hardware Monitors**

In this category of monitoring systems, events are detected by a separate object (a hardware monitor) observing system’s physical environment such as system bus, memory ports, or I/O channels [MLCS90]. Hardware monitors are the least intrusive because they do not compete with the monitored system for runtime resources and therefore has little to no effect on performance or behavior of the monitored system. According to [WH88, MSS93], disadvantages of hardware monitors include hardware monitors are less portable and more expensive; they provide very low-level system data that is hard to understand and use, and usually considerable processing is needed to compose application level data from the low machine level data; monitored low machine level data is not sufficient for application level debugging; and finally installation of hardware monitors requires great expertise and thorough knowledge of the system. ZM4 [HKM+94] monitoring system supports both hardware and hybrid monitoring.

**Software Monitors**

By contract to hardware monitors, software monitors present application level information that is easy to understand and use; they are portable, cheaper, and more flexible to design and install. However, software monitors generally share runtime resources such as time and space with the monitored system and cause degradation of the system’s performance or inaccuracy/alternation to system’s behavior. For this reason, instrumentation
and monitoring should be limited to essential and important application events. Pure monitoring is not very suitable for on-line, real time monitoring [MSS93]. Various instrumentation approaches exist for probing the software and detecting events; they include instrumentation of the software in the sourcecode, in library routines, in compiled objects (bytecode), and in the kernel [MSS93].

**Instrumenting Sourcecode**

Software probes are inserted into appropriate places in monitored system’s sourcecode to generate and detect events. Instrumentation at sourcecode level provides powerful and flexible monitoring. The disadvantage of instrumentation at sourcecode level is that instrumenting large and complex software applications is laborious and error prone. Example of monitoring systems that perform instrumentation in the sourcecode include ALAMO [JZTB98], JRTM [ML97a, ML97b], TemporalRover [Dru00], and Meta [MCWB91].

**Instrumenting Library Routines**

Instrumentation of library routines of monitored system provides an alternative to sourcecode probing. Software probes can be inserted to the sourcecode of the library routines of monitored software to detect and report events. The advantages of instrumentation at the library routines level are that any software that uses instrumented library routines can be monitored, and libraries can be linked or disconnected easily, therefore, a complete recompilation of monitored system can be avoided. The disadvantage of this method is that only events with their corresponding probes in the library routines can be monitored. NetLogger [TG02], log4j [Apa], Open Group’s Enterprise Management Forum’s Application Response Measurement (ARM) [ARM], and Jade [JLSU87] are examples of monitoring by instrumentation of library routines.
Instrumenting Compiled Objects

Instrumentation can also be done at compiled bytecode level using special instrumentation compiler. This instrumentation approach is less intrusive than instrumentation in the sourcecode and in the library routines methods because it does not involve any application’s sourcecode editing and can be transparent to the programmers. The disadvantage of this method is that it requires an instrumentation compiler [MSS93]. JavaMac [KKLS01], MOP [CDR04] and BIT [LZ97] are examples of monitoring systems that instrument at the bytecode level.

Aspect Oriented Programming (AOP) [KLM+97] is a software development technique aiming at separation of concerns [TOHS99]. AspectJ [Asp] is an aspect-oriented extension to the Java programming language that enables modularization of cross cutting concerns of logging and error handling. Programmers can use the AspectJ Development Tools (AJDT) to instrument monitored systems at the bytecode level.

Instrumenting the Kernel

When instrumentation is done at the kernel level, software probes are inserted to the code of the kernel to detect events. Advantage to this instrumentation method is that it is not intrusive, while its disadvantage includes only events related to kernel calls can be detected. None of the monitoring systems we reviewed in previous sections fall into this category of monitoring.

Hybrid Monitors

Hybrid monitors try to benefit from the advantages of both hardware and software monitors and utilize both hardware devices and software probes to detect events. Typically, hybrid monitoring systems insert software probes into monitored systems to gather application level data, and specialized hardware devices are used to receive, analyze and display this monitoring information. Compare to hardware monitors, hybrid monitors are
cheaper and more portable; they are more intrusive because they share some resources with the monitored system, and hybrid monitors generate application level monitoring information (like software monitors) instead of low level machine data hardware monitors generate. Compared to pure software monitors, hybrid monitors are less portable, and more expensive because they require some hardware devices; they are less intrusive because they share less runtime resources with the monitored system because of the utilization of hardware devices; and they both generate application oriented monitoring information that is easy to understand and helpful for debugging. For the above reasons, many monitoring systems favor hybrid monitors. Examples of hybrid monitoring systems include monitoring approach presented in [SM06] and ZM4 [HKM⁺94].

This concludes our summary of existing monitoring methodologies, systems, and tools. Overall, it is desirable for monitoring systems to generate high level application oriented monitoring information that is useful for application debugging and diagnosis. It is also critical for monitoring systems to be as non-intrusive to monitored applications as possible while taking into consideration monitoring system’s cost, flexibility, power, and type of monitoring information it generates. On line monitors, especially intrusive software monitors, should limit themselves to monitor only essential and important application/system events.

Generated monitoring information and log files may contain errors that correspond to abnormalities of the monitored system’s behaviors and system failures. Further analysis and diagnosis are normally required to explain such failures and answer the questions of “what went wrong with the system” and “what are the root causes of the problem”. In the following sections, we review fundamental theories and representative research projects on diagnosis from Artificial Intelligence (AI) area of research.
2.4 AI Theories of Diagnosis

There are two different kinds of theories and approaches to the design of diagnostic reasoning systems in the AI research: the diagnosis from the first principles approach and the experiential approach. In the diagnosis from the first principles approach, a description of the system to be diagnosed and observations of its behavior are available. System description captures the structure and design of the system and its intended/correct behavior. If system’s observations conflict with system’s specified intended behavior, diagnosis is needed to explain this discrepancy. The only available information to diagnostic reasoning is the system description and observations of the system behavior - no heuristic information about system failure is available. This approach is also known as Model Based Diagnosis (MBD) approach where a system’s structural and behavior properties are declaratively modeled. On the contrary, under the experiential approach, heuristic information plays a dominant role and system’s description is only weakly represented, if at all. Therefore little to none “deep” knowledge of the system is presented, and diagnosis relies on codified human experience of past system failures [Rei87]. The two diagnostic reasoning approaches are complementary to each other. Since our proposed research on diagnosis adopts and extends on existing diagnosis from the first principles research, we limit ourselves to exclusively overview fundamental theories and representative research in the category of diagnosis from the first principles approach only.

Early AI research of model based diagnosis (diagnosis from first principles) [dKMR92, Rei87] focuses on diagnosing static systems such as electric circuits and it aims to answer the question of “what is wrong” with the system. Malfunctioning of system components is accountable for any aberrant system behaviors. Recently, more research focuses on diagnosing dynamic systems where actions that affect states of the system are reasoned about to explain system failures. The aim of dynamic diagnosis is to answer the question of “what went wrong” with the system [McI98, Iwa02, CT94]. In this research, actions or failure of actions are typically root causes of system failure or malfunctioning.
2.4.1 Diagnosing Static Systems

Within formal accounts of diagnosis of static systems, two widely accepted definition of diagnosis are consistency based diagnosis [Rei87, dKMR92], and abductive explanation [dKMR92]. Consistency based diagnosis looks for a minimal set of abnormal system components that is consistent with the union of system specification, system input settings, and system observations. While abductive diagnosis looks for a minimal conjunction of abnormal and normal components such that its union with system’s specification knowledge base (KB) and system input settings entails system observations.

In [Rei87], Reiter presented a formal theory of consistency based diagnosis from first principles which built upon and generalized work of de Kleer [dK76] and Genesereth [Gen84], and it provided formalization to the diagnostic theory that [dK76] and [Gen84] lack.

Conventional Model Based Diagnostic research views the system as a set of interacting components and each of the component has the property of being either abnormal or normal. Reiter’s theory of diagnosis [Rei87] appeals to first order logic with equality as a language and system description and observation are presented as a set of first-order sentences. However the theory is general and requires only the system be described in a suitable logic. A diagnosis is a conjecture of some minimal set of components that are faulty or abnormal. Component abnormality is expressed with a unary predicate $AB$, interpreted to mean “abnormal”, e.g. $AB(C)$ holds when component $C$ is abnormal. The assumption that each component of this set is abnormal ($AB$) and all other system components are normal ($\neg AB$) should be consistent with the system description and observation, and when consistency is decidable, diagnoses are computable.

Diagnoses are computed using the concept of conflict sets and hitting sets. Specifically, a set of faulty components is a diagnosis to a system if and only if this set is a minimal hitting set for system’s minimal conflict sets. [Rei87] presented how minimal hitting sets for a collection of sets can be calculated using a HS-tree, and three tree pruning
techniques for efficient calculation were discussed. Diagnoses can then be calculated by first computing the collection of all conflict sets, and the pruned HS-tree is used to compute the minimal hitting set. These minimal hitting sets are diagnoses to the system. A single fault diagnosis is a diagnosis with only one faulty component, and a multiple fault diagnosis is one with two or more faulty components. The faulty component of a single fault diagnosis must be an element of every minimal conflict set for the system. When multiple diagnoses are found, additional observations (measurements) are needed in order to discriminate among competing diagnoses. Measurements are first-order sentences that reject the diagnoses they disconfirm. When a diagnosis predicts logical formula $P$ or $\neg P$, measuring $P$ retains all diagnoses predicting $P$, and rejects all diagnoses predicting $\neg P$.

Reiter also argued in [Rei87] that diagnostic reasoning is nonmonotonic, and a close connection between diagnostic reasoning and nonmonotonic reasoning can be observed when the underlying languages are first-order logic and default logic. A nonmonotonic logic is a formal logic whose consequence relation is not monotonic meaning that adding a formula to a theory (learning a new piece of knowledge) may reduce the set of what is already known (set of consequences).

Reiter’s research is significant in diagnostic reasoning because it provides a formal definition and theory of consistency based diagnosis; it explicitly used the $AB$ predicate for expressing fault and the relationship between faults. The theory is also general enough to allow the use of a wide variety of logics; it presented an algorithm for calculating diagnoses and made the distinction between single fault and multi-fault diagnoses. Finally Reiter’s work discussed measurements in discriminating competing diagnoses and made a connection between diagnostic reasoning and nonmonotonic reasoning.

Like Reiter’s work, most diagnostic approaches characterize all diagnoses for a system as a set of failing components that explain the discrepancy between system expectations and behaviors. These approaches that model only the correct behavior of components does not generalize easily and cannot be applied to models that model faulty behavior
Chapter 2. Literature Survey

of the system. More formally put, the generalization problem arises because not every superset of components of a diagnosis necessarily provides a diagnosis. In [dKMR92], de Kleer proposed two approaches to address this problem: to consider the notion of kernel diagnosis that is free of this problem and to consider restricting the axioms used to describe the system so that the problem does not arise.

De Kleer et al.’s work [dKMR92] followed definitions and conventions presented in Reiter’s framework [Rei87]. Most model-based diagnoses define a diagnosis to be a set of faulty components, and all other components are assumed to function correctly. In [dKMR92], components that are functioning correctly are made explicit and a diagnosis is a conjunction which explicitly indicates whether each component is normal or abnormal. This representation of diagnosis is consistent in semantics with previous definitions of diagnosis in the literature, but it generalizes more naturally. Under this definition, a diagnosis exists (can be computed) when the union of first order sentences of system description and observations is satisfiable. A conflict represents any discrepancy between system expectations and behaviors. A diagnosis is a minimal diagnosis if and only if each faulty component in the diagnosis is a prime implicant of the set of positive minimal conflicts of the system. Under this definition, every superset of the faulty components of a minimal diagnosis is also a diagnosis. Therefore, finding positive minimal conflicts and finding prime implicants are main steps of many diagnostic algorithms. De Kleer et al. [dKMR92] also presented the concept of a partial diagnosis: a partial diagnosis is a diagnosis where we do not provide normal or abnormal status to the components whose statuses do not matter. The minimal partial diagnosis is called a kernel diagnosis. Therefore, $D$ is a diagnosis to the system if and only if there is a kernel diagnosis that covers it. Moreover, partial diagnoses of a system are (prime) implicants of the minimal conflicts of the system. An algorithm of computing kernel diagnoses from the minimal conflicts is presented. In prime diagnoses, a diagnosis of a system is characterized in terms of diagnoses of its individual components. Prime diagnoses characterize
An alternative to the above consistency based approach is to define diagnoses in terms of abduction. Abductive diagnoses are diagnoses in which observations about input and output are differentiated. In abductive diagnostic reasoning, the union of first order sentences of system description, input and abductive diagnosis should be satisfiable and entail/predict system output. Abductive diagnoses are characterized: $D$ is an abductive diagnosis to the system if and only if there is a kernel abductive diagnosis which covers it.

In a $P$-abductive diagnosis, union of a conjunction $P$ of $AB$-literals, system description, and input is satisfiable and entails system output. A $P$-abductive diagnosis is not covered by other $P$-abductive diagnoses.

The second approach of addressing the problem faced by minimal diagnosis approach is to make sure that every superset of the faulty components of a minimal diagnosis provides a diagnosis (referred to as “Minimal Diagnosis Hypothesis”) by enforcing each $AB$-literal to be positive. [dKMR92] presented some practical restrictions on observations and system descriptions to ensure Minimal Diagnosis Hypothesis holds.

De Kleer et. al. proposed formal definitions of all spaces of diagnoses and addressed some problems faced by minimal diagnostic reasoning. The presented diagnostic definitions are general because they can model both system inputs and outputs; both correct and faulty behaviors of the system. This formal framework of characterizing all diagnoses bridges diagnoses with notions of implicates and implicant. The limitation of this framework is that it is not practical in diagnosing real world applications because the problem of computing implicates and implicants is NP-hard, and the number of kernel/prime diagnoses is large in real world applications.

### 2.4.2 Diagnosing Dynamic Systems

The consistency based and abductive diagnostic reasoning presented in the previous section typically focuses on static systems where there are no agents in the system capable
of performing actions. Because many applications are dynamic in the sense that they are agent oriented and are purposive, being able to reason about effects of actions constitutes a necessity for diagnosing dynamic systems.

McIlraith’s work on diagnosing dynamic systems [McI98] took as the starting point existing model based diagnostic research on characterizing diagnoses for static systems without representations of actions [Rei87, dKMR92], and adopted the situation calculus theory of action that enables the integration of a representation of actions with the representation of the system [McI97, McI98]. Axiomatization used in representing the system includes both the behavior of the static system, the actions that can affect the states of the system, as well as actions that are needed for testing and repair in case of a system failure [McI98].

The situation calculus language is a sorted first-order language with equality that includes representation of primitive actions, situations and domain objects. Actions are treated as first class citizens and are associated with precondition axioms and successor state axioms for specifying their preconditions and effects. Situations are simply sequence of actions. The evolution of the world starts with an initial situation $S_0$, and with each action execution, the world evolves to a new situation which is the conjunction of the history of executed actions and the newly executed action. Action’s precondition axiom describes the necessary and sufficient conditions under which an action is executable. Predicates and variables whose values may vary from situation to situation are called *fluents*. If a fluent $f$ is not mentioned in the effect of an action that is executed in a situation, we would not know the value of $f$ after action execution in the next situation, and $f$ can take on an arbitrary value. To fully capture the dynamics of a changing knowledge base (KB), it is also necessary to have formulas specifying which fluents are unaffected by performing an action. These formulas are often called frame axioms and they present a serious problem (sometimes called the frame problem) because it will be necessary to reason with a large number of frame axioms for all the fluents, actions, and
situations in the KB. McIlraith’s research on diagnosis using situation calculus [McI98] addressed the frame problem using successor state axioms [Rei91]. Intuitively, successor state axioms states that if a fluent \( f \) is true in a situation, then it must be either some action was executed that made it true, or a state constraint make it true, or it was already true in the previous situation and neither an action or a state constraint made it false [McI98]. This axiomatization made a completeness assumption on successor state axioms which means that all the conditions under which an action \( a \) can change the value of a fluent \( f \) are captured in the axiomatization of the system.

McIlraith’s formulations of system description follow the conventional notation in the Model Based Diagnosis literature [Rei87, dKMR92], and axiomatization includes a background theory, a set of interacting components and system observations. McIlraith added to this conventional axiomatization of system description also the history (sequence) of executed actions starting from the initial situation. The aim of explanatory diagnosis is to find out “what happened” or “which sequence of actions that must have occurred” in addition to the history of executed actions so that the observations are true. More formally put, an explanatory diagnosis for a system with history of executed actions, \( HIST \), is a sequence of actions \([a_1, \ldots, a_n]\) such that the observations are true after performing the action sequence \([HIST \cdot a_1, \ldots, a_n]\) from the initial situation. Therefore, identifying the sequence of actions composing explanatory diagnosis is analogous to the AI plan synthesis problem. Computing an explanatory diagnosis is analogous to generating a plan that achieves the goal of system observation starting from the situation after the execution of action sequence recorded in history. Extending Reiter’s results on soundness and completeness of regression [Rei91, Rei92], McIlraith showed that the task of generating explanatory diagnosis can be reduced to regression followed by entailment. Regression searches backwards from the observation rather than search forward from the initial situation to constrain search space [McI98].

Difficulties faced by computing explanatory diagnosis include the search space maybe
large and the initial knowledge base may be incomplete. McIlraith further proposed to
(1) make assumptions regarding the domain, (2) relax criteria for explanatory diagnosis,
and (3) verify candidate diagnosis rather than generate to ease the problem. In the first
approach, in the absence of information to the contrary, assumptions are made that all
the components are initially functioning correctly. Assumption-based explanatory diag-
nosis was proposed under this assumption. The computation of explanatory diagnosis
can also be made easier by relaxing the criteria for defining an explanatory diagnosis.
The initial definition of explanatory diagnosis requires that the knowledge base entails
actions’ precondition in the action sequence which may not be reasonable in an incom-
pletely specified database. Potential explanatory diagnosis was then proposed where the
definition of a diagnosis was relaxed to require the union of knowledge base and actions’
preconditions of actions in the [HIST explanatory • diagnosis] action sequence to be
satisfiable. In the last approach to address the problem of generating explanatory di-
gnosis, McIlraith proposed to compile a knowledge base of explanatory diagnoses and
their associated context (in terms of observations and action sequence history) so that
the problem of generating explanatory diagnosis is reduced to the verification problem.
Given a system and a candidate diagnosis $E$, the problem is to verify that $E$ is indeed a
diagnosis for the system.

McIlraith’s research [McI97, McI98] was the first to explicitly address the problem
of integrating a rich representation of actions to the problem of diagnostic reasoning.
It put actions to the foreground of diagnosis generation and provided a comprehensive
representation of actions’ preconditions and effects as well as addressing the frame, rami-
fication and qualification problems. McIlraith’s research made apparent the link between
explanatory diagnosis generation and planning.

Baral, McIraith, and Son [BMS00] extended McIlraith’s diagnostic reasoning by ex-
ploting narratives to express observations and they defined the notion of diagnostic and
repair planning which aims at discriminating and refining diagnoses and possibly find-
ing a repair. The distinction between sensing actions and world-alternating actions was made which enabled the authors to distinguish between changes in the state of the world and changes in an agent’s state of knowledge. The language $L$ for specifying and reason with narratives was used and extended to support sensing actions and observable fluents. The language $L$ support three language constructs: domain description language, a language to specify observations, and a query language. A narrative is a pair of domain description and a set of observations. [BMS00] followed conventional definition of diagnosis [Rei87, dKMR92, McI97, McI98] and assumed the system contains a set of interrelated components that can be abnormal (indicated using $AB(C)$ predicate). In addition, a new predicate $break(C)$ was proposed over components to describe unexpected observations about $AB(C)$. Static causal laws and dynamic causal laws are used to describe causal relation between fluents and action’s effects. A system needs a diagnosis if its narrative (composed of system description and observation) is inconsistent, or does not have a model. A diagnostic model is then defined as the model of system’s narrative and a diagnosis is a set of faulty system components, represented using conjunctions of $AB(C)$ predicates, such that there exists a model in which the diagnosis holds.

In [BMS00], the language $L$ was augmented to incorporate sensing actions and observable and unobservable fluents. A sensing action is an action that when executed, generates more knowledge for the database and may consequently disconfirm some candidate diagnoses with added constraints. This augmentation enabled diagnostic and repairing planning which was missing from authors’ previous work on model based diagnosis and dynamic diagnosis with reasoning about actions [McI97, McI98]. Both explanatory diagnosis [McI98] and diagnosis with narratives and sensing [BMS00] integrates model based diagnosis and reasoning with actions and their effects. Their main difference, in terms of the definition of what constitutes a diagnosis, is that in explanatory diagnosis work, a diagnosis consists of a sequence of actions that when executed leads to observed aberrant system observations, while in diagnosis with narratives and sensing research, a diagnosis
consists of a set of abnormal/faulty components, and actions do not constitute part of a diagnosis.

Iwan extended McIlraith’s work on explanatory diagnosis and proposed history based explanatory diagnosis [Iwa02] where actions composing a diagnosis are allowed to occur at any point of the action history instead of only at the end as that in McIlraith’s research. Iwan argued that the history containing the sequence of executed actions is “assumed” history, and the basic action theory was extended to take into account the possibility that some actions in the history may not occur as assumed, or have occurred but did not achieve their predefined effects, or exogenous events may happen that are out of agent’s control that cause aberrant system behaviors.

To accommodate this assumption, Iwan proposed the concept of extended variation to the history where the basic action theory is extended with two predicates $\text{Varia}$ and $\text{Inser}$ meaning valid “variations” of actions and “insertions” of exogenous events/actions in the history respectively. Intuitively, a variations of an action can be one of two things: the action does not occur because its precondition is not meet, or the action was initiated but its effects do not hold true after action’s execution. Variations to actions are another source of failure that could explain an aberrant system observation. For this reason, action variations are actions, and if conjectured, constitute part of a diagnosis. For example, if we have in our domain action $\text{take}(x)$ meaning take an object $x$, its variation can be $\text{notake}(x)$, meaning either the action take did not occur or it occurred but its effects are false. In this example, $\text{notake}(x)$ would be part of the diagnosis. Insertions are suitable additional events/actions that may have occurred and can assist in explaining an observation. Variations and insertions of actions in the history constitute a diagnosis, and history based diagnoses are possible histories that are extended variations of the assumed history. More formally put, an explanatory history based diagnosis for an observation $\text{OBS}$ and a history $\text{HIST}$ is an extended variation $\text{OBS}$ of $\text{HIST}$ which is executable and after which $\text{OBS}$ still holds. [Iwa02] discussed diagnosis preference criteria used for
choosing a preferred diagnosis among many diagnoses, and showed how this preference value can be calculated. History-based diagnosis templates are proposed to compactly represent action sequence which constitutes a diagnosis, and a formula constraining the possible instances of the action sequence was also presented.

The main contribution of Iwan’s work is that it considers action failures and exogenous events as possible explanations to a contradicting observation. It is an extension to McIlraith’s explanatory diagnosis [McI98] because actions constituting a diagnosis can occur in between actions in the history rather that at the end. One of the drawbacks to Iwan’s notation of action variation is we can not differentiate between the case where an action was never executed because of a previous failure and the case where the action was attempted but failed. Domain analysts need to look into the details of action history and its diagnosis to tell the difference.

Prior to McIlraith’s [McI98] and Iwan’s [Iwa02] proposed research on explanatory diagnosis, Cordier et. al proposed event-based diagnosis [CT94] that is similar in motivation. The diagnostic task is to determine a temporal sequence of events that occurs in between observations so as to yield any observed aberrant system behaviors. The events that constitute a diagnosis correspond to system’s components changing their internal states, to external events that have affected the system’s behavior, or to the system’s own change of dynamics. The major drawback of this proposed research is that it lacks a comprehensive representation of actions and their associated preconditions or effects. Without an sufficient representation of actions, the frame, ramification and qualification problems were left unaddressed [McI98].

Sampath et. al [SSST95, SSST96, LSSS01] proposed a DES (Discrete Event Systems) approach to the failure diagnosis problem and the notion of diagnosability of event languages. One of the advantages of proposed research is that it does not require detailed, in-depth modeling of the system to be diagnosed. The proposed approach consists of two main steps: (1) model the system behavior as regular language and represent it by a finite
state machine (FSM); and (2) construct the diagnoser from the FSM model of the system and also represent it in a FSM. The FSM model of the system captures both the normal and abnormal behavior of the system, and failure is modeled as unobservable events. The diagnoser’s task is then to infer past occurrences of these failure events by estimating the state of the system after the occurrence of each observable event. A language is diagnosable if it is possible to detect occurrences of unobservable failure events [SSST95].

In [PC05], Pencole and Cordier proposed a framework for decentralized diagnosis of large scale discrete event systems. Discrete event systems (DES) are characterized by a discrete-state space of logical values and event driven dynamics. The authors argued that traditional model based diagnostic reasoning is not sufficient in diagnosing large scale distributed systems because they require the computation of a global model of supervised system which is impossible. The aim of proposed research is to provide on-line, decentralized diagnostic reasoning by following “divide-and-concur” computing strategy. A distributed system is naturally composed of several interconnecting subsystems. The task of diagnosing the entire system can be reduced to the problem of diagnosing individual subsystems. Diagnosis for the entire system (a global diagnosis) can then be obtained by merging diagnoses of all subsystems. The diagnosis problem is defined as identifying failure events and their propagations along the network which explain observed aberrant system behaviors. Propagations of event failures are modeled as paths of event transitions in the network. The drawback of this research remains to be the lack of a rich representation of actions.

Decentralized approaches to diagnosis complex systems addresses the state explosion problem for large sized models. In [CG07], Cordier and Grastien raised the issue that the size of the diagnosis itself should also be considered. The authors defined two independent properties: state-independence and transition-independence. These two properties enable us to get decentralized diagnosis representation without losing any information. In [LCD], Console et al. proposed a framework for decentralized model-based diagnosis.
Subsystems are associated with local diagnosers, which are supervised by a supervisor. The supervisor knows the global communication architecture between peers, and integrates a global diagnosis by talking with local diagnosers. The computation of global diagnoses from local ones is centralized.

There has been research work on reducing the problem of diagnosis to that of a satisfiability (SAT) problem that can be solved by SAT solvers. In [GARK07], Grastien et al. presented using a SAT-based approach to find out whether the behavior of a system is normal or faulty. Grastien et al.’s work encodes a propositional formula $\Phi$ where $\Phi = \Phi_{SD} \cup \Phi_{OBS} \cup \Phi_{Que}$. The first component $\Phi_{SD}$ represents system description; the second component $\Phi_{OBS}$ represents observed system behavior; and the third component $\Phi_{Que}$ is the conjunction of negations of all possible failure events. The authors predefined a set of possible failure events. $\Phi_{Que}$ represents none of these predefined failure events have occurred. Hence, if $\Phi$ is satisfiable, the system behavior is normal. Otherwise, the system behavior is faulty.

The above described approach only tells you if a system behavior was normal or faulty. It doesn’t tell you how many failures have occurred, nor does it identify any failures. To address this issue, the authors rewrote $\Phi$ as $\Phi = \Phi_{SD} \cup \Phi_{OBS} \cup \Phi_{Que_i}$. Components $\Phi_{SD}$ and $\Phi_{OBS}$ still represent system description and behavior respectively. The original $\Phi_{Que}$ component can be thought of as $\Phi_{Que_0}$ as it indicates that the behavior of the system was correct. The authors built formula $\Phi_{Que_i}$ to represents exactly $i$ faults occurred in the system. Therefore, $\Phi \ (= \Phi_{SD} \cup \Phi_{OBS} \cup \Phi_{Que_i})$ is satisfiable if and only if $i$ faults have occurred. If $\Phi$ is not satisfiable, $i$ is incremented to $i+1$ to test if system contains $i+1$ faults. This process continues till $\Phi$ becomes satisfiable, and the exact $i$ is discovered. To identify the faults, the authors extract them from SAT solver’s satisfying truth assignment [GARK07]. Rintanen and Grastien also proposed diagnosability testing using SAT algorithms [RG07]. Grastien et al.’s work depends on the availability of a set of predefined failure events. There is no action representation and reasoning.
Our diagnostic reasoning work is largely inspired by the traditional Model based diagnosis (MBD) research [Rei87, dKMR92] and explanatory diagnosis that integrates fundamental theories of MBD and representation and reasoning of actions [McI98, Iwa02]. We adopted and extended these AI theories of diagnosis and action to address the problem of diagnosing of software applications.

2.5 Reconfiguration

Reconfiguration research can be grouped under two categories. Research in the first category are given an initial configuration, and their aim is to identify a goal configuration that the system should configure to. Our reconfiguration work falls under this first category. Research in the second category are given both an initial and a goal configuration, and their aim is to identify actions that lead the system from the initial to the goal configuration. We review these two categories of research in the following sections.

2.5.1 Identifying Configurations

In [SW06], conflict-directed A* search is used to find good points in a search space. The search first defines a score for each point in the search space. It then identifies plateaus, which are group of points in the search space with the same scores. These plateaus are ordered in best-first order. The search looks for solutions that satisfy all constraints at each plateau starting with the plateau with the highest score. A SAT solver can be used. The first solution found is guaranteed to be the best since the search starts from the highest plateau.

In [LS03, dGHLZ], algorithms that solve the weighted constraint satisfaction problem (weighted CSP or WCSP) are evaluated. The WCSP problem deals with finding a point in a search space with constraints. A CSP is specified with variables, their domains, and constraints on the variables. The constraints specify a list of assignments of variables
that must occur in a valid point in the search space. WCSP adds a weight to each constraint, which allows some constraints to be soft (i.e. do not have to be satisfied). However, there is a cost to not satisfying the constraints. The WCSP solver will look for a solution that satisfies the constraints in a way that minimizes the costs.

Both of the approached in [SW06, LS03, dGHLZ] are similar to ours in that they identify best configurations from a search space. They differ from ours in that we use different algorithms for doing so. Furthermore, our work uses goal models, whereas theirs do not.

In [WNN96], Williams et al. apply conflict directed best first (A*) search to reconfiguration. It is similar to our work in that it looks at reconfiguration of autonomic systems. It differs from our reconfiguration work in that it looks at a transition system instead of a goal model and we use different algorithms to identify good reconfigurations.

Salifu et. al presented a formal approach for analyzing the impact of changes in a software system’s context (environment) on the satisfaction of the system’s requirements [SYN, SYN08]. The impact of varying contextual properties on monitoring and switching was studied. Monitoring aims to detect changes in the system’s context. If changes are detected, the software system’s requirements are violated. Behavior switching specifications specify new behaviors for the system to switch to. The main difference between our work and Salifu et. al’s is that their approach focuses on monitoring requirements violations caused by changes in the system’s context, while our work diagnoses failures within the software system itself.

Li et. al. presented a self-reconfiguration system for service oriented systems [LSQC05]. A software system’s non-functional requirements are represented as service level agreements (SLAs). Reconfiguration is triggered if either an SLA is violated, or if a resource over- or under- consumption is detected. The difference between our work and Li et. al’s is that their approach monitors the satisfaction of system’s non-functional requirements. Consequently, the system reconfigures for the purpose of self-optimization. Our work
Chapter 2. Literature Survey

monitors the satisfaction of system’s functional requirements. The system reconfigures for the purpose of failure recovery.

In [NSK03, KNK05], Neema et al. present an algorithm that explores the design space of a software system. A system is specified using a dataflow diagram in which some elements are groups of elements that can be chosen between. The dataflow diagram is turned into a configuration space representation that is similar to a goal model. If there is a choice to be made, something similar to an OR-decomposed goal is created. Otherwise, an AND-decomposed goal is created. Choosing a design from the design space would be then similar to choosing a valid configuration from the goal model. Their approach adds constrains on what designs can be chosen. This is similar to adding extra constraints to a SAT solver when computing participating diagnostic components. The dataflow diagram is encoded into boolean formulas in the form of Ordered Binary Decision Diagram, which can be used to find valid designs in the design space. Their work only looks for valid configurations and does not look at choosing between them. In contrast, our reconfiguration work looks at using some kind of scoring system to decide between multiple configurations.

2.5.2 Identifying Actions to Reach Configurations

In [CFM+06], AI planning is used to help scientists use three applications to process images and specify antenna movements. The basic approach involves the following steps: specify all possible pre- and post- conditions of actions or tasks; ask the users for the starting point and the end goal; then find all possible combinations of actions that can lead the system to go from the starting point to the goal given all the conditions. This is analogous to generating all valid plans that satisfy all constraints, and asking the users to choose their favorite plan. In contrast, our work does not require underlying rules or user interaction. Furthermore, their focus is on creating all valid plans, whereas our focus is on examining already existing plans (reconfigurations) and figure out what to do
if they are not fully satisfied.

[GH04] looks at the steps required for a system to reconfigure itself dynamically. Given a reconfiguration, the paper describes what steps need to be taken for the system to change successfully. For example, consider a system with one master component and many slave components. If the master component is to be replaced, the first step is to notify all slaves. This work focuses on the mechanism of performing a specific reconfiguration, and does not look at how to decide what the reconfiguration should be.

In [BHR+06] Balani et al. shows how to update sensor networks by either updating an entire module, a sub-module, or specific parameters. Again, this work focuses on how to efficiently perform a reconfiguration, instead of choosing what the reconfiguration is.

In [WN97], the authors present an approach to online planning of individual actions given an initial and a final state. Computationally expensive parts of planning are performed at compile time and data is stored in a way that allows efficient access. This work differs from ours in that it looks at what actions should be taken to achieve a reconfiguration, instead of looking at what the reconfiguration should be.
High variability software systems deliver their functionalities in multiple ways by reconfiguring their components. If one of these ways fails, the system can switch to an alternative. We propose an autonomic architecture that takes as input a high variability legacy software system, and extends it with self-repair capabilities through reconfiguration. In this Chapter, we describe four software engineering steps needed to prepare our architecture’s input (Section 3.1); give an overview of the architecture (Section 3.2); and discuss framework implementation details (Section 3.3).

3.1 Preparing a Legacy Software System

Four software engineering steps are necessary to prepare for the architecture’s input. (1) A reverse engineering step where a specification (either requirements model or statechart) is reverse engineered from source code. (2) An annotation step where domain experts annotate the specification. (3) An instrumentation step where the managed software system is instrumented so that it produces log data at run time. And (4), a compensation preparation step where compensation actions are prepared by domain experts. In this section, we describe the four software engineering steps in detail.

Our architecture is built on the premise that the system’s specification is available
at run time. A system specification is either a requirement goal model or a statechart. In either case, goal models and/or statecharts can be reverse engineered from source code using techniques we presented in [YWM+05], or they can be provided by domain experts. Both goal models and statecharts are hierarchical. In a goal model, a goal can be decomposed to to lower level subgoals and tasks. Similarly in a statechart, a hierarchical (or super) state can be refined to lower level hierarchical states and basic states. This hierarchical nature of our specifications allows us to model and analyze a software system at different levels of abstraction.

In the annotation step, domain experts annotate the goal models or statecharts. If requirements are used, the goal model is annotated with monitoring switches, preconditions and effects. When these switches are enabled, the satisfaction of the corresponding goals and tasks is monitored at run time. Goal and task satisfaction/denial is analyzed using the truth values of their associated preconditions and effects. If statecharts are used, they are annotated with monitoring switches, state condition formulas, preconditions and effects. The framework monitors the satisfaction of monitored states and transitions. State and transition satisfactions/denials are inferred using the truth values of their annotations.

In the instrumentation step, the instrumentation component obtains monitoring information, such as “what” to monitor (from specification), and “where” the monitors should be inserted (from the traceability links). Using this information, the instrumentation component semi-automatically generates monitoring specifications in AspectJ terminologies. The AspectJ compiler then uses these specifications to automatically instrument the software at its byte code level [Zho08].

In the compensation preparation step, domain experts provide compensation actions for all the tasks in the system’s goal model or all the actions in the system’s statechart. These compensation actions are added to the system’s source code, and are linked to the specification through traceability links. We adopt a model of database long lived trans-
actions (LLTs) for executions of compensation actions [GM87]. An LLT (also referred to as a saga) consists of a sequence of sub-transactions $T_1, T_2, \ldots, T_n$. Compensation actions $C_1, C_2, \ldots, C_n$ are defined for all the tasks (or transitions) $T_1, T_2, \ldots, T_n$ in the goal model (or in the statechart). The compensation executer component ensures that either the sequence $T_1, T_2, \ldots, T_n$ or the sequence $T_1, T_2, \ldots, T_j, C_j, \ldots, C_2, C_1$ is executed for some $0 \leq j < n$ [GM87]. Executions of these compensation actions restore a system to its previous consistent state when failures occur. It’s noteworthy that in many applications, system actions are fairly independent and they do not need to access the same resources. In these cases, it may make sense to compensate for only failed actions, rather than for all actions that are successfully completed. Because compensations are application and domain dependent, the domain experts need to decide on a case-by-case basis on what are appropriate compensation behaviors for a given software system.

3.2 Architecture Overview

The proposed framework verifies that a system is in compliance with its specification (either its requirements or its statechart). Figure 3.1 provides an overview of our architecture when requirements are used as a basis for monitoring and diagnosis. The architecture contains Monitoring, Diagnosis, Reconfiguration Planner, and Reconfiguration and Compensation Executor components that respectively correspond to the monitor, analyze, plan, and execute (MAPE) components in an autonomic system. The current framework does not offer a reconfiguration mechanism when a monitored system’s specification is its statecharts. It is one of our future work to extend the framework so that repair is possible when monitoring and diagnosis are based on statecharts.

At run time, the instrumented system generates complete log data that are collected and managed by the monitoring component. Complete log data contain log traces for all the tasks and some selected goals in the goal model. The monitoring component collects
the complete log data and saves them to a *Log Database*. To reduce diagnostic reasoning overhead, the monitoring component selects a subset of the complete log data and passes it to the diagnostic component for analysis. The selected log data contain log traces for a subset of selected goals and tasks in the goal model, and they contain enough information to allow the Diagnosis component to infer whether any requirements are denied. Using these selected log data, the system can reconfigure without finding a precise diagnosis, if finding a precise diagnosis is not important.

We transform the problem of diagnosing software systems into a SAT problem. In our work a diagnosis specifies for each goal and task whether or not it is fully denied. The diagnostic component contains three sub-components: a SAT encoder, a SAT solver, and a SAT decoder. The SAT encoder encodes goal model relations and log data into a propositional formula that is satisfied if and only if there is a diagnosis. A symbol table records the mapping between propositional literals and diagnosis instances. The SAT solver finds one possible satisfying assignment, translated by the SAT decoder into a possible diagnosis. The SAT solver can be repeatedly invoked to find all truth assignments that correspond to all possible diagnoses. The diagnostic component analyzes the returned diagnoses, and checks whether or not a denial of system requirement is found. If denials of system requirements are found, either a single diagnosis is returned identifying one or more failed tasks, or multiple competing diagnoses are returned, each pinpointing
different goal/task denials.

The reconfiguration planner component computes a best reconfiguration from all reconfigurations without failed goals/tasks. The reconfiguration component therefore has to “know” which goals/tasks have failed. If a single diagnosis is generated, it indicates either an absence of denials or pinpoints specific denials. In either case, the reconfiguration planner needs no further information. If multiple diagnoses are returned, the diagnostic component can follow one of three approaches. The first two infer a single precise diagnosis. (1) The diagnostic component can retrieve a subset of relevant log data corresponding to failed goals/tasks from the Log Database, and infer therefrom a precise diagnosis. (2) The diagnostic component can generate a most probable diagnosis using the available partial log data. These approaches are useful when repair makes it important to find the exact root causes of failure. At other times, it may be more desirable to find reconfigurations quickly. In this case, the diagnostic component can (3) pass all the generated diagnoses to the reconfiguration planner component. The reconfiguration planner component analyzes the goal model structure and the generated diagnoses to compute all valid reconfigurations. The partial log data used contains enough information to make this possible.

Of course, if no requirements are denied, no reconfiguration is needed. Otherwise, the reconfiguration planner component computes a best reconfiguration that the system should adopt in its next execution. In our work, a configuration contains a list of tasks from the goal model, whose successful executions lead to the satisfaction of the root goal. The planner component first computes all reconfigurations that do not include failed goals/tasks. Then it chooses a best reconfiguration among them based on how positively it contributes to the soft goals in the goal model, and how minimally it reconfigures the system from its current configuration. The same or the new configuration is passed to the Reconfiguration and Compensation Executer component for execution.

If the configuration passed to the Reconfiguration Executor component is the same as
the system’s current configurations, no configuration changes are made. Otherwise the executor component executes the necessary compensation actions to bring the system to its previous consistent state. We adopt a model of long-term database transactions for executions of compensation actions [GM87]. The executor component reconfigures the system using the new configuration. The steps described above constitute one execution session and may be repeated.

3.3 Implementation Details

The presented framework containing monitoring, diagnosis, reconfiguration planner, and executor components has been implemented using the Java programming language. The source code contains 22 Java classes with about 9000 LOC. We translated the problem of requirement diagnosis to a SAT problem. The diagnostic component uses SAT4J [LB05], an efficient SAT solver, by including its jar file in the framework’s compile path.

Monitoring specifications and instrumentation code are generated semi-automatically using AspectJ [Asp]. Aspect Oriented Programming (AOP) is a technology that implements crosscutting concerns in modular units. AspectJ is an aspect oriented extension to Java. The instrumentation component obtains monitoring information, such as “what” to monitor (from the goal model), and “where” to insert the monitors (from the traceability links). Using this information, the instrumentation component can semi-automatically generate monitoring specifications in AspectJ terminologies. The AspectJ compiler then uses these specifications to automatically instrument the software. Interested readers can refer to [Zho08] for a complete account of the instrumentation component.

High variability software systems have their own configuration mechanisms. In many cases, the configuration is done through command line arguments, configuration files, or simply by changing the states of program variables. Traceability links connect goals and tasks in a goal model to the methods/modules that implement them. The system’s own
configuration mechanism contains information on how to configure the system so that a
certain method/module will (or will not) be included. The executor component automatical-
ically generates an aspect file that contains all the above information - information on
how to reconfigure a system so that a certain goal or task will be executed. The executor
component then taps into the system’s configuration mechanism to effect any necessary
configuration changes.
Chapter 4

Requirement Monitoring and Diagnosis

4.1 Preliminaries

4.1.1 Goal Models

Requirements Engineering (RE) is a branch of Software Engineering (SE) that deals with the elicitation and analysis of system requirements. In recent years, RE has used goal models to model and analyze stakeholder objectives [DvLF93]. Software systems’ functional requirements are represented as hard goals, while their non-functional requirements are represented as soft goals [MLN92]. A goal model is a graph structure including AND-and OR-decompositions of goals into subgoals, as well as means-ends links that relate leaf level goals to tasks (“actions”) that can be performed to fulfill them. We assume that traceability links are maintained between system source code and goals/tasks. At the source code level, tasks are implemented by simple procedures or composite components that are treated as black boxes for the purposes of monitoring, diagnosis, and reconfiguration. This allows us to model a software system at different levels of abstraction. If goal $G$ is AND/OR-decomposed into subgoals $G_1, \ldots, G_n$, then all/at-least-one of the
subgoals must be satisfied for \( G \) to be satisfied.

Following [GMNS02], and apart from decomposition links, hard goals and tasks can be related to each other through various contribution links: \( ++S, --S, ++D, --D, ++, -- \). Given two goals \( G_1 \) and \( G_2 \), the link \( G_1 \xrightarrow{++S} G_2 \) (respectively \( G_1 \xrightarrow{--S} G_2 \)) means that if \( G_1 \) is satisfied, then \( G_2 \) is satisfied (respectively denied). But if \( G_1 \) is denied, we cannot infer denial (or respectively satisfaction) of \( G_2 \). The meanings of links \( ++D \) and \( --D \) are dual w.r.t. to \( ++S \) and \( --S \) respectively, by inverting satisfiability and deniability. Links \( ++S \) and \( --S \) (respectively \( ++D \), and \( --D \)) propagate satisfaction (respectively denial) of the source goal/task to the target goal/task. Links \( ++ \) and \( -- \) are shorthand for the \( ++S \), \( ++D \), and \( --S \), \( --D \) relationships respectively, and they propagate both satisfaction and denial of the source goal/task to the target goal/task. These \( ++ \) and \( -- \) links represent the strong MAKE\((++\)\) and BREAK\((--\)\) contributions between hard goals/tasks. Hard goals and tasks can have one of two satisfaction labels: \( \neg FD \) and \( FD \), representing the full evidence that a hard goal/task is satisfied (not denied) and denied respectively. We chose to follow this label propagation process presented in [GMNS02] because it is already proven to be sound and complete.

The partial (weaker) contribution links HELP \((+\)\) and HURT \((-\)\) are not included between hard goals/tasks because we do not reason with partial evidence for hard goal and task satisfaction and denial. These weaker links may proceed from hard goals/tasks to soft goals. Soft goals can have one to four satisfaction labels: \( FS \), \( FD \), \( PS \), and \( PD \) representing the full and partial evidence that a soft goal is satisfied and denied respectively. The class of goal models used in our work has been formalized in [GMNS02], where sound and complete algorithms are provided for inferring whether a set of root-level goals are satisfied.

We associate goals and tasks with preconditions and postconditions (hereafter effects to be consistent with AI terminology), and monitoring switches. Preconditions and effects are propositional formulae in Conjunctive Normal Form (CNF). Monitoring switches are
Chapter 4. Requirement Monitoring and Diagnosis

boolean flags that can be switched on/off to indicate whether the corresponding goal/task is to be monitored. Satisfactions/denials for goals/tasks are inferred using truth values of their respective preconditions and effects. In addition, we associate soft goals with one of three priority levels: high, medium, or low. Soft goal priority values are used to guide the reconfiguration process to select a reconfiguration that contributes most positively to soft goals of higher priorities.

4.1.2 SAT Solvers

The propositional satisfiability (SAT) problem is concerned with determining whether there exists a truth assignment $\mu$ to variables of a propositional formula $\Phi$ that makes the formula true. If such a truth assignment exists, the formula is said to be satisfiable. A SAT solver is any procedure that determines the satisfiability of a propositional formula, identifying the satisfying assignments of variables.

The earliest and most prominent SAT algorithm is DPLL (Davis-Putnam-Logemann-Loveland) [DLL62], which uses backtracking search. Even though the SAT problem is inherently intractable, there have been many improvements to SAT algorithms in recent years. Chaff [MMZ+01], BerkMin [GN02] and Siege [Rya04] are among the fastest SAT solvers available today. For our work, we use SAT4J [LB05], an efficient SAT solver that inherits a number of features from Chaff.

4.2 A Running Example

We use the SquirrelMail [Cas07] case study as a running example throughout this report to illustrate how our framework works. SquirrelMail is an open source email application that consists of 69711 LOC written in PHP. Figure 4.1 presents a simple, high-level goal graph for SquirrelMail with 4 goals and 7 tasks, shown in ovals and hexagons, respectively.

The root goal $g_1$ (send email) is AND-decomposed into task $a_1$ (load login form), goal
Chapter 4. Requirement Monitoring and Diagnosis

4.3 Monitoring

The monitoring component monitors requirements and generates log data at different levels of granularity. There is a tradeoff between monitoring granularity and diagnostic precision. The finest level of monitoring granularity is at the functional level where all leaf level tasks are monitored. In this case, task-level log data is generated, and a
single precise diagnosis can be inferred. The disadvantage of task level monitoring is high monitoring and diagnostic overhead. Coarser levels of granularity only monitor higher-level goals in a goal model. In these cases, less complete and goal-level log data is generated, leading to multiple competing diagnoses. Both the monitoring and diagnostic overheads are lower. The disadvantage is that if requirements denials are found, multiple diagnoses are returned, each pinpointing possible failures.

As shown in Figure 3.1, the instrumented system generates complete log data at run time. This data contains log traces for all tasks, allowing for a single precise diagnosis to be inferred. It also contains log traces for some selected goals. Section 4.3.2 describes which goals are selected for monitoring. The monitoring component selects a subset of the complete log data and passes it to the diagnostic component for analysis. Analyzing a smaller amount of data reduces diagnostic reasoning overhead, and the selected log data are still enough to allow the diagnostic component to infer whether any requirements are denied. In addition, the selected log data permits fast reconfiguration without a precise diagnosis.

Monitored goals and tasks need to be associated with preconditions and effects whose truth values are monitored and are analyzed during diagnostic reasoning. Preconditions and effects may also be specified for goals and tasks that are not monitored. This allows for more precise diagnoses by constraining the search space for analysis. Precondition and effects can be specified for each goal. Alternatively, they can be specified at the task level and then propagated to higher level goals using techniques presented in [MF02]. Errors may be introduced if (1) the goal model is not correct, i.e. it does not correctly or completely capture the monitored system’s requirements, and (2) the specified preconditions and effects for goals and tasks are not correct, i.e. they do not correctly capture the desired behaviors of the software system. Detecting or dealing with these two types of errors is beyond the scope of our work. We assume that both the goal model and its associated preconditions and effects are correctly specified for the application.
Chapter 4. Requirement Monitoring and Diagnosis

In the log data, each task occurrence is associated with a specific logical timestep \( t \). We introduce predicate \( \text{occ}_a(a_i, t) \) to specify occurrences of tasks \( a_i \) at timestep \( t \). For example, if task \( a \) is executed at timestep 1, its log instance is: \( \text{occ}_a(a, 1) \). Successful execution of tasks in an appropriate order leads to satisfaction of the root goal. A goal is satisfied in some execution session \( s \) if and only if all the tasks under its decomposition are successfully executed in \( s \). Goal satisfaction or denial may vary from session to session. The logical timestep \( t \) is incremented by 1 each time a new batch of monitored data arrives and is reset to 1 when a new session starts.

We say a goal has occurred in \( s \) if and only if all the tasks in its decomposition have occurred in \( s \). Goal occurrences are not directly observable from log data. Instead, they are inferred by our framework from tasks occurrences in the log data. Two timesteps, \( t_1 \) and \( t_2 \), are associated with goal occurrences, representing the timesteps of the first and the last executed task in the goal’s decomposition in \( s \). We introduced predicate \( \text{occ}_g(g_i, t_1, t_2) \) to specify occurrences of goals \( g_i \) that start and end at timesteps \( t_1 \) and \( t_2 \) respectively. For example, suppose goal \( g \) is decomposed into tasks \( a_1 \) and \( a_2 \), and we have in the log data \( \text{occ}_a(a_1, 4), \text{occ}_a(a_2, 7) \) indicating that tasks \( a_1 \) and \( a_2 \) have occurred at timesteps 4 and 7 respectively. Then \( \text{occ}_g(g, 4, 7) \) is inferred to indicate that \( g \)'s occurrence started and ended at timesteps 4 and 7. We say that a session is complete when the root goal of the goal model has occurred. Each log file contains log traces for one execution session only on the entire goal model.

The monitored system’s runtime behavior is traced and recorded as log data consisting of truth values of observed domain literals (specified in goal/task preconditions and effects) and the occurrences of tasks, each associated with a specific timestep \( t \). A log is made of a sequence of log instances, defined as follows:

**Definition 1 (Log instance)** A log instance is either the truth value of an observed literal or the occurrence of a task, at a specific timestep \( t \).
For example, if literal $l_1$ was true at timestep 1, task $a$ occurred at timestep 2, and literal $l_2$ was false at timestep 3, their respective log instances are: $l_1(1)$, $occ_a(a, 2)$, and $\neg l_2(3)$.

### 4.3.1 SquirrelMail Example Log Data

Table 4.1 lists the details of each goal/task in the SquirrelMail goal model (Figure 4.1) with its monitoring switch status (column 2), and associated precondition and effect (columns 3 and 4). In this example, the satisfaction of goal $g_4$, and tasks $a_1, a_2, a_6,$ and $a_7$ are monitored.

<table>
<thead>
<tr>
<th>Goal/Task</th>
<th>Monitor switch</th>
<th>Precondition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>on</td>
<td>URL entered</td>
<td>correct form loaded</td>
</tr>
<tr>
<td>a2</td>
<td>on</td>
<td>$\neg$wrongIMAP$\land$ correct form loaded</td>
<td>correct key entered</td>
</tr>
<tr>
<td>a3</td>
<td>off</td>
<td>correct key entered</td>
<td>form shown</td>
</tr>
<tr>
<td>a4</td>
<td>off</td>
<td>form shown</td>
<td>form entered</td>
</tr>
<tr>
<td>a5</td>
<td>off</td>
<td>form entered</td>
<td>webmail started</td>
</tr>
<tr>
<td>a6</td>
<td>on</td>
<td>wrongIMAP</td>
<td>error reported</td>
</tr>
<tr>
<td>a7</td>
<td>on</td>
<td>webmail started</td>
<td>email sent</td>
</tr>
<tr>
<td>g1</td>
<td>off</td>
<td>URL entered</td>
<td>email sent $\lor$ error reported</td>
</tr>
<tr>
<td>g2</td>
<td>off</td>
<td>correct form loaded $\lor$ wrongIMAP</td>
<td>webmail started $\lor$ error reported</td>
</tr>
<tr>
<td>g3</td>
<td>off</td>
<td>correct form loaded $\land$ $\neg$wrongIMAP</td>
<td>webmail started</td>
</tr>
<tr>
<td>g4</td>
<td>on</td>
<td>correct key entered</td>
<td>webmail started</td>
</tr>
</tbody>
</table>
The monitored system’s runtime behavior is traced and recorded as log data consisting of truth values of observed domain literals (specified in goal/task preconditions and effects) and the occurrences of tasks, each associated with a specific timestep $t$. The following is an example of log data from the SquirrelMail case study:

URL entered(1), occ$_a$(a1, 2), correct form loaded(3), ¬wrongIMAP(4),
occ$_a$(a2, 5), correct key entered(6), occ$_a$(a3, 7), occ$_a$(a4, 8), occ$_a$(a5, 9),
¬webmail started(10), occ$_a$(a7, 11), ¬email sent(12).

The log data contains two errors (¬webmail started(10), and occ$_a$(a7, 11)): (1) the effect of $g_4$ (webmail started) was false, at timestep 10, after all the tasks under $g_4$’s decomposition (a3, a4, and a5) were executed at timesteps 7, 8, and 9 respectively; and (2) task a7, send message occurred at timestep 11 when its precondition webmail started was false before its occurrence, at timestep 10. The diagnostic component (presented in Chapter 5.4) analyzes the log data and infers that the goal $g_4$ and the task $a_7$ are denied.

### 4.3.2 Selecting Monitoring Granularity

Our framework allows for fast reconfiguration without a precise diagnosis, if finding a precise diagnosis isn’t important. The framework achieves this by selecting and following an optimal monitoring granularity. It monitors as few goals/tasks as necessary. Yet, when failures occur, satisfactions/denials of these monitored goals/tasks give the reconfiguration component enough information to find all reconfigurations free of failures.

Intuitively, goals and tasks that share the same reconfiguration are monitored collectively, rather than individually. Consider the example goal model given in Figure 4.2. Tasks $a_3$, and $a_4$ share the same reconfiguration, because if either fails, their respective reconfigurations are the same. It is therefore sufficient to know if any of these tasks fails, rather than which task fails. We introduce the concept of AND-subtree given in Definition 2. An AND-subtree does not contain any OR decompositions between its root
and any of its tasks. The root of an AND-subtree is iteratively AND-decomposed to all of its leaf level tasks through intermediate goals.

**Definition 2 (AND-subtree)** An AND-subtree is part of a goal model where all the decompositions between the root of the subtree and its leaf level tasks are AND-decompositions.

**Definition 3 (Maximal AND-subtree)** A Maximal AND-subtree is an AND-subtree where no other AND-subtrees contain it.

In the goal model given in Figure 4.2, goals $g_5$, $g_7$, $g_8$, and $g_9$ root AND-subtrees. Goals $g_5$ and $g_7$ root two Maximal AND-subtrees.

We propose Algorithm 1 for selecting an optimal monitoring granularity. Algorithm 1 selects for monitoring: (1) goals and tasks with multiple parents; (2) root goals of maximal AND-subtrees; and (3) tasks that are OR-decomposed from their immediate parents. Goals/tasks with multiple parents are monitored because they participate in multiple reconfigurations. It is necessary to know whether or not they are denied before all reconfigurations can be computed. All tasks that belong to a maximal AND-subtree, and that do not have multiple parents share the same reconfiguration. We monitor their
Algorithm 1 Select an Optimal Monitoring Granularity

```c
select_monitoring_granularity (goal_model) {
    for each goal g {
        if (g has multiple parents)
            select g for monitoring
        else if (g roots a maximal AND-subtree)
            select g for monitoring
    }
    for each task t {
        if (t has multiple parents)
            select t for monitoring
        else if (t is OR-decomposed from its parent)
            select t for monitoring
    }
}
```

satisfactions collectively by monitoring their root goal in the subtree. Tasks that are OR-decomposed from their immediate parents are monitored, because these tasks’ satisfactions/denials determine if they participate in a reconfigurations. When this optimal monitoring granularity is followed, all reconfigurations free of failures can be computed. This allows for fast reconfiguration without a precise diagnosis. In the goal model given in Figure 4.2, the following goals and tasks are recommended for monitoring: $a_1, a_2, g_5, a_5, a_6$, and $g_7$ (their respective ovals/hexagons are highlighted in Figure 4.2).

### 4.4 Diagnosis

#### 4.4.1 Formal Foundations

This section presents the formal foundations of our proposed diagnostic component. The axiomatizations generated for diagnostic reasoning (presented in sections 4.4.1 and 5.4.5)
are adaptations of the theoretical diagnostic framework proposed in [Rei87, dKMR92, McI98].

**Basic Formulation for SAT**

We reduce the problem of searching for diagnoses to that of the satisfiability of a propositional formula \( \Phi \). \( \Phi \) is written in the form:

\[
\Phi := \Phi_{LOG} \land \Phi_{deniability} \land \Phi_{goal} \land \Phi_{domain\ constraints}
\]  

(4.1)

The first component \( \Phi_{LOG} \) represents log data generated by monitors as specified in Definition 7 (Chapter 4.3). The second component \( \Phi_{deniability} \) encodes denials of tasks and goals (sections 4.4.1 and 5.4.5). The third component \( \Phi_{goal} \) encodes goal relations and forward and backward propagation (section 4.4.1). The last component, which is optional, \( \Phi_{domain\ constraints} \), encodes any domain constraints and relations that are not represented in the goal graph.

**Axiomatization of Deniability**

We formulate the denial of goals and tasks in terms of the truth values of the predicates representing their occurrences, preconditions and effects. Intuitively, if a task’s precondition is true and the task occurred at timestep \( t \), and if its effect holds at the subsequent timestep \( t + 1 \), then the task is not denied at timestep \( t + 1 \). Two scenarios describe task denial: (1)\(^1\) if the task’s precondition is false at timestep \( t \), but the task still occurred at \( t \); or (2) if the task occurred at timestep \( t \), but its effect is false at the subsequent timestep \( t + 1 \). Axiom (1) captures both of these cases. The preconditions and effects are specified in CNF. All propositional literals are grounded to domain instances. For example, a propositional literal \( a \), representing a task, may be grounded to task instance *send email* in an email application domain.

\(^1\)In many axiomatizations it is assumed that \( \text{occ}_a(a, t) \rightarrow p(t) \), where \( p \) is the precondition of \( a \).
Axiom 1 (*Task Denial Axiom.*) A task \(a\) with precondition \(p\) and effect \(q\) is denied at timestep \(t + 1\) if and only if the task occurred at the previous timestep \(t\), and either \(p\) was false at \(t\), or \(q\) was false at \(t + 1\).

\[
FD(a, t + 1) \iff \text{occ}_a(a, t) \land (\neg p(t) \lor \neg q(t + 1))
\] (4.2)

A goal occurrence is indexed by two timestep arguments denoting the timesteps of the first and the last executed tasks under the goal’s decomposition. As with a task, the goal’s precondition and effect need to be true before and after, respectively, the goal’s occurrence for a goal to be satisfied.

Axiom 2 (*Goal Denial Axiom*) A goal \(g\) with precondition \(p\) and effect \(q\) is denied at timestep \(t_2 + 1\) if and only if the goal occurrence finished at a previous timestep \(t_2\), and either \(p\) was false when goal occurrence started at \(t_1\) \((t_1 \leq t_2)\) or \(q\) is false after goal occurrence finished at \(t_2 + 1\).

\[
FD(g, t_2 + 1) \iff \text{occ}_g(g, t_1, t_2) \land (\neg p(t_1) \lor \neg q(t_2 + 1)) \land (t_1 \leq t_2)
\] (4.3)

If there is only one task under \(g\)’s decomposition, the goal occurrence starts and ends at the same timestep as the task occurrence timestep. In this case, \(t_1 = t_2\). Denial of goals and tasks in the goal model are traced back to the monitored system’s sourcecode to identify buggy implementations and problematic components.

Axiom 3 (*Task and Goal Session Denial Axioms*) A task, \(a\), or a goal, \(g\), is denied during an execution session, \(s\), if \(a\) or \(g\) is denied at some timestep, \(t\), within \(s\).

\[
FD(a, t) \rightarrow FD(a, s)
\] (4.4)

\[
FD(g, t) \rightarrow FD(g, s)
\] (4.5)

As will become clear in the following sections, inferring the truth values of \(FD(a, s)\) and \(FD(g, s)\) on all tasks and goals is useful when we propagate their denial labels to the rest of the goal graph.
Returning to the SquirrelMail case study, the following denial axioms are generated for task $a7$, send message, goal $g4$, show compose message, and for timesteps 1 and 2:

\[
FD(a7, 2) \leftrightarrow occ_a(a7, 1) \land (\neg webmail\, started(1) \lor \neg email\, sent(2))
\]

\[
FD(g4, 2) \leftrightarrow occ_g(g4, 1, 1) \land (\neg correct\, key(1) \lor \neg webmail\, started(2))
\]

\[
FD(a7, 2) \rightarrow FD(a7, s)
\]

\[
FD(g4, 2) \rightarrow FD(g4, s)
\]

**Explanation Closure Axioms**

Propositional literals whose values may vary from time step to time step are called *fluents*. If a fluent $f$ is not mentioned in the effect of a task that is executed at timestep $t$, we would not know the value of $f$ after task execution at timestep $t + 1$. In this case, $f$ can take on an arbitrary truth value. To fully capture the dynamics of a changing knowledge base (KB), it is also necessary to know what fluents are unaffected by performing a task. Formulas that specify unaffected fluents retaining the same values are often called frame axioms. These present a serious problem because it will be necessary to reason with a large number of frame axioms for all the fluents, tasks, and timesteps in the KB.

We adopt Explanation Closure Axioms [Rei91] to capture the effects on fluents as well as to address the frame problem. We make a completeness assumption on tasks’ and goals’ effects: we assume that the effects specified for goals and tasks characterize all conditions under which a goal or a task can change the value of a fluent. Therefore, if the value of a fluent $f$ changes at timestep $t$, then one of the tasks/goals that has $f$ in its effect must have occurred at a previous timestep $t - 1$ and not have been denied at $t$.

Explanation Closure Axioms are described by axioms (5.7) and (5.8) which state that, for any fluent $f$ that is in a positive (or negative) effect of tasks $a_1, \ldots, a_n$ and goals $g_1, \ldots, g_m$, if $f$ does not hold (or does hold) at timestep $t$, but holds (or does not hold
respectively) at step \( t + 1 \), then one of the tasks \( a_i \) must have occurred at timestep \( t \) and not have been denied at the subsequent timestep \( t+1 \), or one of the goals \( g_j \) must have occurred between timesteps \( t_1 \) and \( t_2 \) and not have been denied at the subsequent timestep \( t_2 + 1 \), where \( t_1 \leq t \leq t_2 \).

If \( f \) is in a positive effect of tasks \( a_i \) and goals \( g_j \), \( (i \in [1 \ldots n] \text{ and } j \in [1 \ldots m]) \),

\[
- f(t) \land f(t + 1) \leftrightarrow \bigvee_i (occ_a(a_i, t) \land \neg FD(a_i, t + 1)) \lor \\
\bigvee_j (occ_g(g_j, t_1, t_2) \land \neg FD(g_j, t_2 + 1) \land (t_1 \leq t \leq t_2))
\]

If \( f \) is in a negative effect of tasks \( a_i \) and goals \( g_j \), \( (i \in [1 \ldots n] \text{ and } j \in [1 \ldots m]) \),

\[
f(t) \land \neg f(t + 1) \leftrightarrow \\
\bigvee_i (occ_a(a_i, t) \land \neg FD(a_i, t + 1)) \lor \\
\bigvee_j (occ_g(g_j, t_1, t_2) \land \neg FD(g_j, t_2 + 1) \land (t_1 \leq t \leq t_2))
\]

For example, in the SquirrelMail case study, according to Table 4.1, only the task \( a_7 \) has the fluent \( email \ sent \) in its positive effect. The following explanation closure axiom is generated for the fluent \( email \ sent \), for timesteps 1 and 2:

\[
- email sent(1) \land email sent(2) \leftrightarrow occ_a(a_7, 1) \land \neg FD(a_7, 2)
\]

The conjunction of axioms (4.2) to (5.8) encodes the \( \Phi_{\text{deniability}} \) component of the propositional formula \( \Phi \) (equation 5.1), and they represent goal and task denial relations.

**Axiomatization of the Goal Model**

Axioms (4.8) and (4.9) describe the forward and backward propagations of the goals’/tasks’ satisfaction/denial labels in the goal model. If a goal \( g \) is AND (or OR) decomposed into subgoals \( g_1 \ldots g_n \), and tasks \( a_1 \ldots a_m \) then there is full evidence that \( g \) is denied in a certain session, \( s_i \), if and only if at least one (or all) of the subgoals or tasks in its decomposition is (or are) denied in that session.
Axioms (6.1) to (4.13) describe the contribution links between goals. With the introduction of these links, the goal graph may become cyclic and conflicts may arise. We say a conflict holds if we have both \( FD(g, s) \) and \( \neg FD(g, s) \) in one execution session \( s \).

Since it does not make sense, for diagnostic purposes, to have a goal being both denied and satisfied at the same time, conflict tolerance in [SGM04] is not supported within our diagnostic framework.

\[
\begin{align*}
(g_1 \ldots g_n, a_1 \ldots a_m) \xrightarrow{\text{AND}} g : FD(g, s) & \iff (\bigvee_i FD(g_i, s)) \lor (\bigvee_j FD(a_j, s)) \quad (4.8) \\
(g_1 \ldots g_n, a_1 \ldots a_m) \xrightarrow{\text{OR}} g : FD(g, s) & \iff (\bigwedge_i FD(g_i, s)) \land (\bigwedge_j FD(a_j, s)) \quad (4.9)
\end{align*}
\]

The following propagation axiom is generated for the goal \( g_4 \) in the SquirrelMail example, stating that \( g_4 \) is denied if and only if at least one of its subtasks \( a_3, a_4, \) or \( a_5 \) is denied:

\[
FD(g_4, s) \iff FD(a_3, s) \lor FD(a_4, s) \lor FD(a_5, s)
\]

The conjunction of axioms (4.8) to (4.13) encodes the \( \Phi_{\text{goal}} \) component of the propositional formula \( \Phi \) (equation 5.1). These axioms represent the AND/OR decompositions and contribution links in the goal model.

**Characterizing Diagnoses**

**Definition 4 (Diagnosis)** A Diagnosis \( D \) for a software system is a set of \( FD \) and \( \neg FD \) predicates over all the goals and tasks in the goal graph, indexed with respect to
timesteps and a session, such that \( D \cup \Phi \) is satisfiable.

For example, consider a goal \( g \) that is AND-decomposed to tasks \( a_1 \) and \( a_2 \). If there are a total of 2 timesteps in the execution session \( s \), and if both \( a_2 \) and \( g \) are denied at timestep 2 during \( s \), the diagnosis to the system would contain:

\[
\neg FD(a_1, 1), \neg FD(a_1, 2), \neg FD(a_1, s), \neg FD(a_2, 1), FD(a_2, 2), FD(a_2, s), \neg FD(g, 1), FD(g, 2), FD(g, s).
\]

**Theorem 1** Let \( D \) be a set of \( FD \) and \( \neg FD \) predicates over all the goals and tasks in the goal graph, indexed with respect to timesteps and a session. \( D \) is a diagnosis for a software system if and only if \( D \cup \Phi \) is satisfiable.

**Proof** (If:) Assume that \( D \cup \Phi \) is satisfiable. If \( D \cup \Phi \) is satisfiable, \( \Phi \) is satisfiable. The SAT solver solves \( \Phi \) and returns one possible satisfying truth assignment \( \mu_i \), which the diagnostic component decodes to a valid diagnosis \( D_i \). Let \( \mu_1, \ldots, \mu_i, \ldots, \mu_n \) be all the possible truth assignments to \( \Phi \), and \( D_1, \ldots, D_i, \ldots, D_n \) be all the possible diagnoses decoded from them. Since any diagnosis \( D_i \) is decoded from \( \mu_i \), which is a possible truth assignment to \( \Phi \), \( D_i \cup \Phi \) is satisfiable. If \( D \) is a set of \( FD \) and \( \neg FD \) predicates over all the goals and tasks in the goal graph and \( D \cup \Phi \) is satisfiable, \( D \) equals \( D_i \) for some \( i \). Therefore \( D \) is one of the possible diagnoses of the system.

(Only if:) Assume that \( D \) is a diagnosis of the system. It follows straightforwardly from Definition 8 that if \( D \) is a diagnosis, \( D \cup \Phi \) is satisfiable.

Theorem 1 establishes the soundness and completeness of our diagnostic approach (presented in section 4.4.2). The theorem states that the diagnostic component finds a complete set of correct diagnoses representing all the possible denied and satisfied goals and tasks that could account for aberrant system behaviors recorded in the log file. The root cause of a goal denial is the denial of one or more tasks associated with the goal or its subgoals. Therefore, task level denial is the core or root cause of a diagnosis given in Definition 8. If a goal or a task is denied at any timestep \( t \) during an execution session \( s \),
it is denied during $s$ (Axiom 3). It is more useful for the diagnostic component to infer task level denials (*root causes*) during specific sessions.

**Definition 5** (*Core Diagnosis*) A *Core Diagnosis $CD$ for a software system* is a set of $FD$ and $\neg FD$ predicates over all the tasks in the goal graph, indexed with respect to a session, such that $CD \cup \Phi$ is satisfiable.

Consider the same example where goal $g$ and task $a_2$ are denied at timestep 2 during the execution session $s$, the core diagnosis to the system would only contain $\neg FD(a_1, s)$, and $FD(a_2, s)$.

**Corollary 1** Our diagnostic approach finds all the core diagnoses to the software system.

The proof to corollary 1 follows from Theorem 1. When the software system is monitored at the functional level, leaf level tasks are monitored and the most complete log data is generated. A single core diagnosis may be inferred containing denials of leaf level tasks. When the software system is monitored at the requirement level, higher level goals in the goal model are monitored and less complete log data is generated. If the diagnostic component infers that a goal is denied, it returns a complete set of core diagnoses representing all the possible combinations of task denials for leaf level tasks associated with the denied goal during the session. Therefore, in the worst-case, the number of core diagnoses is exponential to the size of the goal graph. To address this problem, we introduce the concept of *participating diagnostic components (PDC)* that correspond to individual task denial predicates that participate in core diagnoses. A core diagnosis can be thought of as a set of participating diagnostic components. Therefore, instead of returning all core diagnoses that represent all the possible combinations of task denials, the diagnostic component returns all *participating diagnostic components* for scalability.
Definition 6 (Participating Diagnostic Component) A participating diagnostic component \( PDC \) for a software system is an FD predicate over some task in the goal model, indexed with respect to a session, such that \( PDC \cup \Phi \) is satisfiable.

Corollary 2 Our diagnostic approach finds all the participating diagnostic components to the software system.

The proof to corollary 2 follows from Theorem 1.

4.4.2 Algorithms

This section discusses the four main algorithms of our framework, namely two encoding algorithms (Algorithms 2 and 3) for encoding an annotated goal model into the propositional formula, \( \Phi \), and two diagnostic algorithms (Algorithms 4 and 5) for finding all core diagnoses and all participating diagnostic components, respectively.

The difference between the two encoding algorithms, Algorithms 2 and 3, lies in whether the algorithm preprocesses the log data when encoding the goal model into \( \Phi \). Algorithm 2 does not preprocess log data and generates a complete set of axioms for all the timesteps during one execution session. The problem with this encoding algorithm is the exponential increase in the size of \( \Phi \) with the size of a goal model. Algorithm 3 addresses this problem by generating all necessary axioms while keeping the growth of the size of \( \Phi \) polynomial with respect to the size of the goal model. We present and compare experimental results using these two algorithms in Section 4.5.

For each task \( a \) in the goal model that is associated with a precondition \( p \) and an effect \( q \), Algorithm 2 generates a task denial axiom, and a task session denial axiom (axioms (2) and (4)) for all the timesteps during the execution session. These axioms cover all the possible task occurrence and denial timesteps. Similarly, for each goal \( g \) with precondition \( p \) and an effect \( q \), the goal denial axiom and the goal session denial axiom (axioms (3) and (5)) are generated for all possible combinations of timesteps \( t_i \).
Algorithm 2 Encode $\Phi$ Without Log Preprocessing

\begin{verbatim}
encode_\Phi_{\text{without\_log\_preprocessing}}(\text{goal\_model}, \text{total\_timesteps}) \{ 
    for each task $a$ { 
        //encode denial axioms 
        if ($a$ is associated with $p$ and $q$) { 
            for each $t_i \in [1, \text{total\_timesteps}]$ { 
                $\Phi = \Phi \land encodeTaskDenialAxiom(a, t_i);$
                $\Phi = \Phi \land encodeTaskSessionDenialAxiom(a, t_i);$
            }
        }
    }
    for each goal $g$ { 
        //encode denial axioms 
        if ($g$ is associated with $p$ and $q$) { 
            for each $t_i \in [1, \text{total\_timesteps}]$ 
                for each $t_j \in [t_i, \text{total\_timesteps}]$ { 
                    $\Phi = \Phi \land encodeGoalDenialAxiom(g, t_i, t_j);$
                    $\Phi = \Phi \land encodeGoalSessionDenialAxiom(g, t_i, t_j);$
                }
        }
    }
    //encode goal model structure 
    for each $t_i \in [1, \text{total\_timesteps}]$
        $\Phi = \Phi \land encodeLabelPropagation(\text{goal\_model}, t_i)$ }
    for each fluent $f$ { 
        for each $t_i \in [1, \text{total\_timesteps}]$
            $\Phi = \Phi \land encodeExplanationClosureAxiom(f, t_i);$
    }
    for each contribution link $l$ { 
        for each $t_i \in [1, \text{total\_timesteps}]$
            $\Phi = \Phi \land encodeContributionLink(l, t_i)$ }
    return $\Phi;$
\}
\end{verbatim}
and \( t_j (t_i \leq t_j) \). These axioms cover all possible goal occurrence and denial timesteps. In addition, explanation closure axioms (axioms (6) and (7)) are generated for all fluents and all timesteps, to specify that after each goal/task execution, truth values of unaffected fluents remain the same from timestep to timestep. Finally, axioms encoding the goal structure and contribution links (Axioms (8) to (13)) are generated for all timesteps. The SAT solver input formula \( \Phi \) is a conjunction of all the generated axioms. The size of \( \Phi \) grows exponentially with the size of the goal model under Algorithm 2.

To address the scalability issue, for each task \( a \), if its occurrence and truth values of its precondition and effect are observed in the log file, Algorithm 3 finds in the log three timesteps: \( t_{occ} \): \( a \)'s occurrence timestep during the execution session \( s \); \( t_p \): the latest observation timestep of \( a \)'s precondition before \( a \)'s execution; and \( t_q \): the earliest observation timestep of \( a \)'s effect after \( a \)'s execution. Then the algorithm generates and adds to \( \Phi \) an axiom stating that if \( a \) occurred at timestep \( t_{occ} \) and if either \( p \) or \( q \) was false at timesteps \( t_p \) and \( t_q \) respectively, \( a \) is denied for the execution session \( s \). It is possible for a task to occur more than once during an execution session. In this case, the algorithm repeats for each of \( a \)'s occurrences during the session. Similarly, for each goal \( g \) whose truth values of associated precondition and effect appear in the log, the algorithm calculates the start and the end timesteps of \( g \)'s occurrence, \( t_1 \) and \( t_2 \), from the occurrence timesteps of the tasks under \( g \)'s decomposition. The algorithm generates and adds to \( \Phi \) an axiom stating that if \( g \) occurred between timestep \( t_1 \) and \( t_2 \) and if either \( p \) or \( q \) was false at timesteps \( t_p \) and \( t_q \) respectively, \( g \) is denied for the execution session \( s \). Therefore, Algorithm 3 generates goal/task denial axioms only for the timesteps at which the goals/tasks actually occur as recorded in the log. Axioms encoding goal model structural and contribution links are generated for the execution session \( s \). As will be illustrated in Section 4.5, Algorithm 3 allows polynomial growth in the size of \( \Phi \) with respect to the corresponding goal model, and allows the diagnostic component to scale to larger goal models.
Algorithm 3 Encode Φ With Log Preprocessing

```c++
encode_Φ_with_log_preprocessing(goal_model, log) {
    for each task a {
        //encode denial axioms
        if (a’s occurrence, and associated p and q are recorded in log ) {
            t_{occ_a} = task occurrence time during s
            t_p = \max_t\{t \leq t_{occ_a} \text{ and } p(t) \in \log\}
            t_q = \min_t\{t > t_{occ_a} \text{ and } q(t) \in \log\}
            if (t_p \leq t_{occ_a} < t_q) {
                \Phi = \Phi \land FD(a, s) \leftrightarrow occ_a(a, t_{occ_a}) \land (\neg p_{t_p} \lor \neg q_{t_q})
            }
        }
        for each goal g {
            //encode denial axioms
            if (g’s associated p and q are recorded in log ) {
                //g’s occurrence is between timesteps t1 and t2
                t_1 = \min_t\{occ_a(a, t) \in \log \text{ and } a \in \text{descendents}(g)\}
                t_2 = \max_t\{occ_a(a, t) \in \log \text{ and } a \in \text{descendents}(g)\}
                t_p = \max_t\{t \leq t_1 \text{ and } p(t) \in \log\}
                t_q = \min_t\{t > t_2 \text{ and } q(t) \in \log\}
                if (t_p \leq t_1 \leq t_2 < t_q) {
                    \Phi = \Phi \land FD(g, s) \leftrightarrow occ_g(g, t_1, t_2) \land (\neg p_{t_p} \lor \neg q_{t_q})
                }
                //encode goal model structure
                \Phi = \Phi \land encodeLabelPropagation (goal model, s) 
            }
            for each contribution link l {
                \Phi = \Phi \land encodeContributionLink(l, s) 
            }
        }
        return \Phi;
    }
}
```
Theorem 2 Let $\Phi'$ be the $\Phi$ computed by Algorithm 3. Let $D$ be any set of $FD$ and $\neg FD$ predicates over all the tasks in the goal graph, indexed with respect to a specific session. $D$ is a diagnosis to the system if and only if $D \cup \Phi'$ is satisfiable.

Theorem 2 establishes the soundness and completeness of Algorithm 3.

**Algorithm 4 Find All Core Diagnoses**

```plaintext
find_all_core_diagnoses(\Phi) {
    while (\Phi is satisfiable) {
        \mu = satisfying assignments for all variables in \Phi
        //map \mu to diagnostic instance
        oneDiagnosis = decodeToDiagnosis(\mu)
        //obtain a new oneCoreDiagnosis containing session
        //level task satisfaction and denial predicates
        oneCoreDiagnosis = session level task satisfactions and denials in oneDiagnosis
        //add to \Phi the negation of both session level and timestep
        //level task denials and satisfactions in oneDiagnosis
        \Phi = \Phi \land \neg \mu \text{task denials and satisfactions in oneDiagnosis}
    }
}
```

Algorithm 4 finds all possible core diagnoses accounting for aberrant system behaviors recorded in the log. If the input formula $\Phi$ is satisfiable, the algorithm decodes the solver result $\mu^2$ into diagnostic instances that constitute a diagnosis. The diagnosis is then filtered into a core diagnosis that contains only $FD$ and $\neg FD$ predicates over tasks, indexed with respect to a session. To have the SAT solver search only on predicate symbols that encode the combinations of denials and satisfactions of tasks, the part of $\mu$ that encodes task denials and satisfactions (both at the session level and at the timestep level) in oneDiagnosis is negated and added back to $\Phi$. The solver is invoked again.

---

2Without loss of generality we treat the set as a conjunction of its elements.
to solve the new $\Phi$. When satisfied, a new $\mu$ is returned and a new core diagnosis is inferred. The procedure repeats until the formula becomes unsatisfiable, by which time it has found all possible core diagnoses that explain errors in the log file. Algorithm 4 finds a complete set of core diagnoses, which, in the worst-case, is exponential in number to the goal graph size, and may not scale to large goal models.

To address the scalability problem, Algorithm 5 returns all participating diagnostic components defined in Definition 6, instead of all core diagnoses. As with Algorithm 4, Algorithm 5 decodes the solver result $\mu$ into a core diagnosis if $\Phi$ is satisfiable. And as with Algorithm 4, Algorithm 5 negates part of $\mu$ and adds it back to $\Phi$, to have the SAT solver return another satisfying truth assignment, until $\Phi$ becomes unsatisfiable. However, the key difference between the two algorithms lies in which part of $\mu$ the algorithm negates and adds back to the solver. Algorithm 4 always negates and adds back all task denials and satisfactions in the returned diagnosis. Consequently, the algorithm has the solver search for all combinations of task denials and satisfactions. Algorithm 5 negates and adds back to the solver either task denials only, or both task denials and satisfactions, depending whether any denied task is associated with contribution links. Consequently, under Algorithm 5 the solver focuses on searching for individual task denials, instead of their exhaustive combinations.

The algorithm works as follows. It finds one core diagnosis and checks it to see if any denied task in it is associated with a contribution link (such as a MAKE (++) or a BREAK (--) contribution link). If any of the denied tasks is associated with a contribution link, and if the task is not already part of a previously returned diagnosis, the boolean flag $\text{negateCompleteConfig}$ is set to true. When this happens, the algorithm negates and adds to $\Phi$ the complete session level task failure configuration in $\mu$ (corresponding to both task denials and satisfactions, i.e., the $\text{oneCoreDiagnosis}$). Negating and adding to the solver the complete failure configuration in $\mu$, when contribution links are present, avoids contradictions between the negation and the constraints imposed by the contribution links.
themselves. Consequently, the algorithm avoids situations where $\Phi$ becomes unsatisfiable because of these contradictions before all the participating diagnostic components can be found. On the other hand, if the boolean flag $\text{negateCompleteConfig}$ is false, all the denied tasks in $\mu$ are either not associated with contribution links, or else are associated with contribution links, but have already been found by the process. In this case, the algorithm negates only the part of $\mu$ that encodes task denials, guiding the solver to move on quickly to other denied tasks.

The SAT solver solves the new $\Phi$ and returns another $\mu$, which is decoded to another core diagnosis, if $\Phi$ is satisfiable. This process repeats till $\Phi$ becomes unsatisfiable, by which time a set of core diagnoses are returned. The algorithm then filters all the returned core diagnoses to obtain all possible individual participating diagnostic components. The aim of Algorithm 5 is to return as few core diagnoses as possible. Nonetheless, the set of core diagnoses returned is still complete enough to cover the set of all possible participating diagnostic components.

The total diagnostic time taken by the framework is proportional to the number of times the SAT solver is invoked. This is equal to the number of core diagnoses returned. Algorithm 5 outperforms Algorithm 4 because Algorithm 4 finds all core diagnoses, whereas Algorithm 5 only finds the core diagnoses necessary to cover all possible participating diagnostic components (corresponding to individual task denials). However, it is noteworthy that core diagnoses contain more useful diagnostic information than participating diagnostic components, such as which tasks may or may not fail together. In situations where diagnostic performance is not a concern, one may decide to use Algorithm 4 instead of Algorithm 5.

Both Algorithms 4 and 5 terminate. Both algorithms terminate when $\Phi$ becomes unsatisfiable (they both contain a while loop). When $\Phi$ is satisfiable, the algorithms enter their respective while loop. During each loop, the algorithms filter out either core diagnosis or participating diagnostic components from the satisfying truth assignment
Chapter 4. Requirement Monitoring and Diagnosis

µ. The algorithms then add the negation of part of µ (that correspond to task denials/satisfactions) back to Φ. SAT solver then solves this new Φ. Φ is going to become unsatisfiable at some point because there are only a finite number of possible combinations of task satisfactions/denials in a goal model. Specifically, if a goal model contains n tasks, there are a maximal of $2^n$ possible combinations of task satisfactions/denials. Each time the algorithms add the negation of part of µ that corresponds to one possible task denial combination to Φ, they are effectively telling the SAT solver that “I have found one possibility, give me another one”. When the SAT solver returns all possible task denial combinations, Φ becomes unsatisfiable. When this happens, both algorithms terminate.

4.4.3 Implementation and Optimizations

The scalability of our framework is largely due to optimizations of our encoding and diagnostic algorithms and optimizations in their implementation. In particular, uses of Algorithms 3 and 5, instead of Algorithms 2 and 4, enhance scalability. In this section, we discuss how Algorithm 5 works differently from Algorithm 4.

The performance comparison of Algorithms 4 and 5 is straightforward when no MAKE/BREAK contribution links are present. Algorithm 4 finds all core diagnoses corresponding to possible combinations of task denials. Algorithm 5 only finds all Participating Diagnostic Components - that is, all individual task denials for tasks under the decomposition of denied goals. If a denied goal G has n tasks in its decomposition, Algorithm 4 returns up to $2^n$ core diagnoses. Algorithm 5 only returns up to n Participating Diagnostic Components.

The situation is more complicated when the goal model has contribution links. We cannot estimate the number of diagnostic returns for either algorithm, because these depend on the number and kind of contribution links in the goal graph. We report a set of 8 experiments in Table 4.2 that compare the efficiency of the two algorithms when
Table 4.2: Optimization of Algorithm 4 Over Algorithm 3

<table>
<thead>
<tr>
<th>#Contr Links</th>
<th>#Core</th>
<th>$Time_C$</th>
<th>#Core to Obtain All PDC</th>
<th>#PDC</th>
<th>$Time_{PDC}$</th>
<th>% Improv</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>n/f</td>
<td>n/f</td>
<td>27</td>
<td>27</td>
<td>1.391</td>
<td>≈ 100%</td>
</tr>
<tr>
<td>1</td>
<td>n/f</td>
<td>n/f</td>
<td>42</td>
<td>27</td>
<td>2.047</td>
<td>≈ 100%</td>
</tr>
<tr>
<td>10</td>
<td>n/f</td>
<td>n/f</td>
<td>53</td>
<td>27</td>
<td>2.782</td>
<td>≈ 100%</td>
</tr>
<tr>
<td>15</td>
<td>4096</td>
<td>7318.00</td>
<td>(&gt;2hrs)</td>
<td>91</td>
<td>5.219</td>
<td>97.78%</td>
</tr>
<tr>
<td>20</td>
<td>299</td>
<td>62.40</td>
<td>18</td>
<td>27</td>
<td>1.276</td>
<td>94.08%</td>
</tr>
<tr>
<td>22</td>
<td>128</td>
<td>17.02</td>
<td>17</td>
<td>27</td>
<td>1.286</td>
<td>86.46%</td>
</tr>
<tr>
<td>25</td>
<td>107</td>
<td>15.28</td>
<td>17</td>
<td>27</td>
<td>1.307</td>
<td>84.06%</td>
</tr>
<tr>
<td>27</td>
<td>16</td>
<td>1.38</td>
<td>10</td>
<td>27</td>
<td>0.953</td>
<td>37.50%</td>
</tr>
</tbody>
</table>

contribution links are present. All 8 experiments use the same goal model with 27 tasks and 23 goals. The 8 experiments feature different numbers and types of contribution links between the tasks in the goal model. For each experiment, the program randomly generates and inserts a number of MAKE/BREAK links between tasks. We injected an error in the log file that corresponds to the denial of the root goal. The table compares the performance of the two algorithms in terms of the numbers of core diagnoses returned. Note that, the number of returned core diagnoses is equal to the number of times the SAT solver is invoked. This, in turn, is proportional to the time the algorithms take to complete their runs.

Column 1 (#Contr Links) lists the number of contribution links in the goal model. Columns 2 and 3 give the performance of Algorithm 4. Column 2 (#Core) lists the number of core diagnoses returned. Column 3 ($Time_C$) lists the total diagnostic time (in seconds) taken to find all these core diagnoses. Columns 4 to 7 give the performance of Algorithm 5. Column 4 (#Core to Obtain All PDC) lists the total number of core
Column 5 (\(\#PDC\)) lists the number of \textit{Participating Diagnostic Components} obtained from parsing the core diagnoses listed in Column 4. Column 6 (\(Time_{PDC}\)) lists the total diagnostic time (in seconds) taken to find these core diagnoses. Column 7 (%Improv) gives the percentage improvement of Algorithm 5 over Algorithm 4. Observe that Algorithm 5 generally returns a small fraction of the core diagnoses returned by Algorithm 4. The percentage improvement is calculated by subtracting this fraction from 1. The fraction is calculated by dividing the number of core diagnoses returned by Algorithm 5 (Column 4) by the number of core diagnoses returned by Algorithm 4 (Column 2) (Equation 4.14).

\[
%\text{Improv} = 1 - \frac{\#\text{Core Diagnoses to Obtain All PDC (Algorithm 4)}}{\#\text{Core Diagnoses (Algorithm 3)}} \tag{4.14}
\]

The number of MAKE/BREAK contribution links increased from 0 to 27 over the course of the 8 experiments. As the number of contribution links increases, the number of core diagnoses decreases. This results from the fact that contribution links add constraints to the SAT solver search space, reducing the number of valid satisfying truth assignments. Note that Algorithm 4 did not finish the first 3 experiments (the first 3 rows in the Table) in real time (represented in the table as “n/f”s). Without “enough” constraining contribution links, there were too many core diagnoses for Algorithm 4 to find within a reasonable period of time. In contrast, Algorithm 5 returned few core diagnoses for these experiments, while still obtaining the complete set of 27 PDCs. The percentage improvement is therefore essentially 100%.

15 contribution links were inserted in the goal mode for the 4th experiment (the 4th row of the table). Algorithm 4 took 7318 seconds (more than 2 hours) to return 4096 core diagnoses. Algorithm 5 returned only 91 of these core diagnoses, from which it obtained the complete set of 27 PDCs. Its entire run took only 5.219 seconds. The percentage improvement is calculated as 1 - 91/4096, or 97.78%. Over the course of experiments 5 to 8 (rows 5 to 8), the number of contribution links increased from 20 to 27. The number of
core diagnoses returned by Algorithm 4 decreased from 299 to 16. In contrast, Algorithm 5’s returns decreases from 18 to 10. In each case, Algorithm 5 obtained all 27 PDCs.

Observe that the percentage improvement decreases as the number of contribution links increases. This is because the number of core diagnoses decreases as the number of constraining contribution links increases. As the number of core diagnoses decreases, Algorithm 4 is able to finish more quickly, narrowing its performance gap. Where the goal graph contains many contribution links, one might decide to use Algorithm 4 instead of Algorithm 5. This will yield a complete set of core diagnoses with more informative diagnostic information (i.e. which tasks/goals failed together).

The implementation of our framework involves other optimizations, as well. For example, we also optimized the encoding algorithm (Algorithm 3) in addition to preprocessing log data. We replaced “String” objects with “StringBuilder” objects with large buffers during the encoding of the propositional formula $\Phi$ used by the SAT solver. Using larger buffers allows for more efficient string concatenations, avoiding excessive memory allocation and copying. This seemingly trivial optimization resulted in a noticeable performance improvement for the encoding component.

4.4.4 Finding a Most Likely Diagnosis

When failures occur, multiple diagnoses may be generated to explain the error. When this happens, repair may make it important to find a single precise diagnosis. We offer two approaches for achieving this. The first approach retrieves a subset of the complete log data (from the Log Database) relevant to the failed goals/tasks, and infers therefrom a precise diagnosis. The second approach computes a most probable diagnosis using information from the existing ones. In this section, we discuss how we carry out the second approach.

We adopt the probability model presented in [Bra09] to calculate probabilities for goal satisfaction given the probabilities that its tasks are satisfied. The satisfaction of a root
goal in a goal model depends on the satisfaction of its tasks, as well as any contribution links that may be present in the goal model. Domain experts assign satisfaction probabilities to all the tasks. These probability values are then propagated to all the goals through AND/OR decomposition links and contribution links. In [Bra09], the probability that a root goal is satisfied is formulated as the conditional probability that its leaf level tasks are satisfied, given that all the contribution links are satisfied. Tools are provided to transform goal models into Bayesian Networks, where an effective Bayesian solver, JavaBayes [Coz01], is used to calculate probabilities of goal satisfactions ([Bra09]).

Probability values propagate straightforwardly from tasks to goals in a goal model without contribution links. The probability that an AND (or OR-) decomposed goal is satisfied is the probability that all (or at least one) of its children are satisfied. Consider the running example in Figure 4.2. If we assign a satisfaction probability of 0.5 to all the tasks, the probabilities that goals $g_4$ and $g_5$ are satisfied are 0.75 and 0.25 respectively. The calculations are as follows. Since $g_4$ is OR decomposed to $a_1$ and $a_2$, $g_4$ is satisfied if and only if not both $a_1$ and $a_2$ are denied. In other words, $p(g_4 \text{ is satisfied}) = 1 - p(a_1 \text{ and } a_2 \text{ are both denied}) = 1 - (1-0.5) \times (1-0.5) = 0.75$. $G_5$ is AND-decomposed to tasks $a_3$ and $a_4$, so $g_5$ is satisfied if and only if both $a_3$ and $a_4$ are satisfied. Therefore, $p(g_5 \text{ is satisfied}) = p(a_1 \text{ and } a_2 \text{ are both satisfied}) = 0.5 \times 0.5 = 0.25$. The calculation of goal satisfaction probabilities from tasks is much more complicated when contribution links are present. This is because contribution links may cause cycles in the goal model and therefore cause conflicts. We say a conflict exists when a goal is both satisfied and denied at the same time. The probability model given in [Bra09] addresses these issues by formulating goal satisfactions/denials as being conditional on all the contribution links being satisfied.

If multiple diagnoses are returned, one or more monitored root goals of maximal AND-subtrees are denied. We find a most probable diagnosis using only the parts of the goal model that contains failed goals/tasks. Satisfaction probabilities are first propagated
from tasks to goals in the failed AND-subtrees. We then use a minimal weight SAT solver [Lib00] to find a most likely diagnoses. The minimal weight SAT solver takes as input, not only a propositional formula Φ, but also a list of integers representing the “weights” of all the goals and tasks in the maximal AND-subtree. We map goal/task satisfaction probabilities to integers, in such a way that higher satisfaction probabilities are mapped to larger integers. The minimal weight SAT solver returns either the minimal weight satisfying truth assignments (if Φ is satisfiable), or UNSAT otherwise. We decode the minimal weight assignment to a most likely diagnosis, which consists of goals/tasks that are least likely to have been satisfied.

4.5 Evaluation

We applied our monitoring and diagnostic components to two medium-size public domain software systems to evaluate its correctness and performance: SquirrelMail [Cas07], a Web-based email client, and an ATM (Automated Teller Machine) simulation [Bjo]. We used the SquirrelMail case study as a running example to illustrate how our framework works. We then used the ATM simulation case study to show that it is feasible to scale our solution industrial applications with medium-sized requirements. All experiments reported were performed on a machine with a Pentium 4 CPU with 1 GB of RAM.

The SquirrelMail Running Example

The encoding component preprocesses the SquirrelMail log data (Section 3.2) as described in Algorithm 3. The diagnostic component infers that the goal $g_4$ and the task $a_7$ are denied during execution session $s$. Then it infers that at least one of $g_4$’s subtasks, $a_3$, $a_4$, $a_5$, must have been denied to account for the denial of $g_4$. Algorithm 4 returns the following 7 core diagnoses:
Core Diagnosis 1: \( FD(a_3, s); FD(a_7, s) \)
Core Diagnosis 2: \( FD(a_4, s); FD(a_7, s) \)
Core Diagnosis 3: \( FD(a_5, s); FD(a_7, s) \)
Core Diagnosis 4: \( FD(a_3, s); FD(a_4, s); FD(a_7, s) \)
Core Diagnosis 5: \( FD(a_3, s); FD(a_5, s); FD(a_7, s) \)
Core Diagnosis 6: \( FD(a_4, s); FD(a_5, s); FD(a_7, s) \)
Core Diagnosis 7: \( FD(a_3, s); FD(a_4, s); FD(a_5, s); FD(a_7, s) \)

Algorithm 4 returns all core diagnoses which are possible combinations of tasks denials for tasks \( a_3, a_4, \) and \( a_5 \) - the tasks which account for the denial of goal \( g_4 \). In contrast, Algorithm 5 returns individual task denials under one denied parent goal, leaving out their possible combinations. The following 4 participating diagnostic components were returned by Algorithm 5:

- Diagnostic Component 1: \( FD(a_3, s) \)
- Diagnostic Component 2: \( FD(a_4, s) \)
- Diagnostic Component 3: \( FD(a_5, s) \)
- Diagnostic Component 4: \( FD(a_7, s) \)

**Performance Evaluation with ATM**

The ATM simulation case study is an illustration of OO design used in a software development class at Gordon College [Bjo]. The application simulates an ATM performing customers' withdraw, deposit, transfer and balance inquiry transactions. The source code contains 36 Java Classes with 5000 LOC, which we reverse engineered to its requirements to obtain a goal model with 37 goals and 51 tasks. We show a partial goal graph with 18 goals and 22 tasks in Figure 4.3.

We report on two sets of experiments in this section. The first contains five experiments with increasing monitoring granularity, all applied to the goal model shown in
Figure 4.3. The goal graph is encoded in the SAT input formula \( \Phi \) using the log preprocessing algorithm (Algorithm 3). We demonstrate and discuss the tradeoff between monitoring granularity and diagnostic precision. The second set reports 20 experiments on 20 progressively larger goal models containing 50 to 1000 goals and tasks. We obtain these larger goal models by cloning the ATM goal graph to itself. We performed this second set of experiments using both encoding algorithms 2 and 3 to compare their efficiency on larger goal graphs. In both sets of experiments, the diagnostic component uses Algorithm 5 to return all participating diagnostic components.

The second set of experiments shows that our diagnostic framework scales to the size of the relevant goal model, provided the encoding is done with log file preprocessing (Algorithm 3) and the diagnostic component returns all participating diagnostic components (Algorithm 5). Our approach can therefore be applied to industrial software applications with medium-sized requirement models.
Table 4.3 reports the results of the first set of experiments. We injected an error into the implementation of task $a_{15}$, *update balance*, with the goal of pinning down a single precise participating diagnostic component, namely $FD(a_{15})$. Column 1 in Table 4.3 lists the number of monitored goals/tasks in the goal graph. Column 2 lists the number of participating diagnostic components returned by the diagnostic component. Columns 3 and 4 give the total numbers of literals and clauses in the propositional formula, $\Phi$, encoded for the SAT solver, using log preprocessing (Algorithm 3). Column 6 gives the total time (in seconds) taken by the diagnostic component to find all participating diagnostic components. $T_{\text{sum}}$ is the sum of the time taken to encode the goal graph into $\Phi$ ($T_{\text{encode}}$), and the time taken to find all diagnostic components. This latter time is calculated by multiplying the time taken to find one diagnostic component ($T_{\text{diagnose}}$) by the total number of returned diagnostic components ($\#\text{Diag Set}$) (Equation 4.15). $T_{\text{diagnose}}$ includes the time taken by the SAT solver to solve the propositional formula $\Phi$ ($T_{\text{solve}}$), and the time taken to decode the SAT result into one diagnostic component ($T_{\text{decode}}$) (Equation 4.16). Column 5 lists the average time ($T_{\text{avg}}$) the solver took to find one participating diagnostic set (in seconds), calculated by dividing $T_{\text{sum}}$ by $\#\text{Diag Set}$ (Equation 4.17)

\[
T_{\text{sum}} = T_{\text{encode}} + T_{\text{diagnose}} \times (\#\text{Diag Set}) \tag{4.15}
\]

\[
T_{\text{diagnose}} = T_{\text{solve}} + T_{\text{decode}} \tag{4.16}
\]

\[
T_{\text{avg}} = \frac{T_{\text{sum}}}{(\#\text{Diag Set})} \tag{4.17}
\]

In the first experiment (row 1 in Table 4.3), we monitored only the root goal $g_{1}$ (highest level of monitoring granularity). The diagnostic component inferred that $g_{1}$ was denied and at least one of the executed tasks under $g_{1}$’s decomposition must have been denied to account for this. A total of 19 participating diagnostic components were returned (column 2). The diagnostic framework took 1 second to find all diagnostic components, which averages to 0.053 second per diagnosis.
Table 4.3: Tradeoff Between Monitoring Overhead and Diagnostic Precision (First Set of Experiments)

<table>
<thead>
<tr>
<th></th>
<th>#Mon</th>
<th>#Diag</th>
<th>#Lit</th>
<th>#Clauses</th>
<th>$T_{\text{avg}}$(s)</th>
<th>$T_{\text{sum}}$(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>19</td>
<td>62</td>
<td>66</td>
<td>0.053</td>
<td>1.000</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>68</td>
<td>76</td>
<td>0.065</td>
<td>0.906</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>73</td>
<td>86</td>
<td>0.073</td>
<td>0.798</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>82</td>
<td>101</td>
<td>0.133</td>
<td>0.531</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>87</td>
<td>116</td>
<td>0.390</td>
<td>0.390</td>
<td></td>
</tr>
</tbody>
</table>

In experiments 2 to 5 (rows 2 to 5 in Table 4.3), the number of goals and tasks that were monitored increased from 3 to 11. With increased monitoring overhead and more complete log data, diagnostic precision improved (fewer diagnostic components were returned). Numbers of generated literals and clauses increased with increasing monitoring granularity, with the average time taken to find a single participating diagnostic component increasing from 0.065 to 0.390 seconds. It’s interesting to note that, even with this increase, the total amount of time the solver took to find all participating diagnostic components decreased from 1 to 0.390 second. This happened because the total number of core diagnoses decreased from 19 to 1.

This first set of experiments showed that the number of participating diagnostic components returned is inversely proportional to monitoring granularity. When monitoring granularity increases, monitoring overhead, SAT search space, and average time needed to find a single participating diagnostic component all increase. The benefit of monitoring at a high monitoring granularity is that we are able to infer fewer diagnostic components identifying a smaller set of possible faulty components. It is also noteworthy that the total amount of time taken to find all diagnostic components may not increase despite the fact that it takes longer to find one diagnostic component. The reverse is true when monitoring granularity decreases: we have less monitoring and diagnostic overhead, but
Table 4.4: Scalability to Goal Model Size with Log Preprocessing (Second Set of Experiments)

<table>
<thead>
<tr>
<th>Goal Model Size</th>
<th>$T_{sum}$ (s)</th>
<th>$T_{encode}$ (s)</th>
<th>$T_{diagnose}$ (s)</th>
<th>#Lit</th>
<th>#Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.469</td>
<td>0.044</td>
<td>0.425</td>
<td>81</td>
<td>207</td>
</tr>
<tr>
<td>100</td>
<td>0.647</td>
<td>0.066</td>
<td>0.581</td>
<td>157</td>
<td>411</td>
</tr>
<tr>
<td>150</td>
<td>0.819</td>
<td>0.100</td>
<td>0.719</td>
<td>233</td>
<td>615</td>
</tr>
<tr>
<td>200</td>
<td>1.006</td>
<td>0.119</td>
<td>0.887</td>
<td>309</td>
<td>819</td>
</tr>
<tr>
<td>250</td>
<td>1.134</td>
<td>0.128</td>
<td>1.006</td>
<td>385</td>
<td>1023</td>
</tr>
<tr>
<td>300</td>
<td>1.260</td>
<td>0.156</td>
<td>1.103</td>
<td>461</td>
<td>1227</td>
</tr>
<tr>
<td>350</td>
<td>1.384</td>
<td>0.200</td>
<td>1.184</td>
<td>537</td>
<td>1431</td>
</tr>
<tr>
<td>400</td>
<td>1.529</td>
<td>0.225</td>
<td>1.304</td>
<td>613</td>
<td>1635</td>
</tr>
<tr>
<td>450</td>
<td>1.650</td>
<td>0.241</td>
<td>1.410</td>
<td>689</td>
<td>1839</td>
</tr>
<tr>
<td>500</td>
<td>1.787</td>
<td>0.278</td>
<td>1.509</td>
<td>765</td>
<td>2043</td>
</tr>
<tr>
<td>550</td>
<td>1.969</td>
<td>0.312</td>
<td>1.656</td>
<td>841</td>
<td>2247</td>
</tr>
<tr>
<td>600</td>
<td>2.159</td>
<td>0.341</td>
<td>1.819</td>
<td>917</td>
<td>2451</td>
</tr>
<tr>
<td>650</td>
<td>2.316</td>
<td>0.375</td>
<td>1.941</td>
<td>993</td>
<td>2655</td>
</tr>
<tr>
<td>700</td>
<td>2.397</td>
<td>0.406</td>
<td>1.991</td>
<td>1069</td>
<td>2859</td>
</tr>
<tr>
<td>750</td>
<td>2.516</td>
<td>0.434</td>
<td>2.082</td>
<td>1145</td>
<td>3063</td>
</tr>
<tr>
<td>800</td>
<td>2.725</td>
<td>0.487</td>
<td>2.238</td>
<td>1221</td>
<td>3267</td>
</tr>
<tr>
<td>850</td>
<td>2.900</td>
<td>0.528</td>
<td>2.372</td>
<td>1297</td>
<td>3471</td>
</tr>
<tr>
<td>900</td>
<td>2.975</td>
<td>0.526</td>
<td>2.450</td>
<td>1373</td>
<td>3675</td>
</tr>
<tr>
<td>950</td>
<td>3.259</td>
<td>0.584</td>
<td>2.675</td>
<td>1449</td>
<td>3879</td>
</tr>
<tr>
<td>1000</td>
<td>3.444</td>
<td>0.628</td>
<td>2.816</td>
<td>1525</td>
<td>4083</td>
</tr>
</tbody>
</table>
the number of participating diagnostic components increases if the system is behaving abnormally. However, if the system is running correctly, and no requirements are denied, no faulty component will be returned, so minimal monitoring is advisable.

Table 4.4 reports the results of the second set of experiments, performed with the log file preprocessing algorithm (Algorithm 3). We experimented on 20 progressively larger goal models containing from 50 to 1000 goals and tasks in order to evaluate the scalability of the diagnostic component. We obtain these larger goal graphs by cloning the ATM goal graph structure (Figure 4.3) to itself. All the experiments are performed with complete (task level) monitoring. Only one diagnostic component is therefore returned for each experiment. Column 1 in Table 4.4 lists the number of goals/tasks in the goal model. Column 3, $T_{encode}$, lists the time taken (in seconds) to encode the goal model into the SAT propositional formula $\Phi$ with log file preprocessing. Column 4, $T_{diagnose}$, lists the time taken by the SAT solver to solve $\Phi$ plus the time taken to decode the SAT result into a diagnostic component. Column 2, $T_{sum}$, calculated by adding $T_{encode}$ and $T_{diagnose}$, represents the total time taken (in seconds) to find the diagnostic component. The total numbers of literals and clauses in $\Phi$ are listed in Columns 5 and 6.
Figure 4.4 depicts the relationship between the total time taken for diagnostic reasoning (the y-axis - the values in columns 2, 3, and 4 of Table 4.4) and the goal model size (the x-axis - the values of column 1 of Table 4.4). The three curves in Figure 4.4 show that the diagnostic component scales to the size of the goal model when using algorithms 3 and 5\(^3\), and our approach can be applied to industrial software applications with medium-sized requirement graphs.

To compare the efficiency between the two encoding Algorithms 2 and 3, we performed this second set of experiments using also Algorithm 2, which encodes \textit{without} log file preprocessing. Figure 4.5 depicts the relationships between the total time taken (in seconds) for encoding and diagnostic reasoning, and the goal model size, using the two encoding algorithms. Figure 4.6 depicts the relationships between the size of $\Phi$ (the total number of literals and clauses) generated by the two encoding algorithms and the goal model size. Encoding \textit{without} log preprocessing gives exponential growth in the size of $\Phi$ with respect to the size of the goal model; an “out of memory” error was returned.

\(^{3}\)There are other dimensions to performance than the size of the goal model, such as the number of participating diagnostic components returned. A SAT solver is revoked for as many times as the number of total participating diagnostic components. Therefore, the diagnostic reasoning time is also affected by the total number of participating diagnostic components.
with experiments on goal models containing more than 400 goals/tasks. In contrast, the experiments using encoding with log file preprocessing scaled well to the goal model size. These experimental results are consistent with our claim that our diagnostic framework scales to the size of the relevant goal models, provided log file preprocessing is used, and all participating diagnostic components (instead of all diagnoses) are returned.

4.6 Multi-layer Monitoring and Diagnosis

Today’s software systems are components of complex socio-technical systems consisting of business processes, applications and computing infrastructure. Service-Oriented Architectures (SOA) constitute a popular example of such complex, multi-layered systems. To further enhance the scalability of our framework, we introduce the concept of hierarchical monitoring and diagnosis. Hierarchical monitoring and diagnosis enables the framework to analyze a software system at different layers in isolation for scalability.

SOA was defined in the late 1990’s, and it presents a loosely coupled, multi-tier architecture. Under SOA, software applications are encapsulated as services with well-
defined interfaces. In order to support software interoperability and a heterogeneous environment, the interfaces follows web-service standard [W3C02]. SOA offers three abstraction layers containing the business process layer (the top layer), the component layer (the middle layer), and the infrastructure layer (the back-end layer). The business process layer treats services as black boxes. The component layer gives the business logic of the services in the business process layer. The back-end infrastructure that the services depend on resides in the infrastructure layer.

To monitor requirement satisfaction of systems that have adopted a SOA, the requirements at each layer are represented in a goal model. The correct functioning at each layer depends on the correct functioning of the layer beneath it. As a result, leaf level tasks at an upper layers decompose to the root goals in the layer beneath it. Figure 4.7 illustrates the ATM case study in terms of the 3 layers of SOA. On top is the business process layer, with the highest level of abstraction. Here, software services are portrayed as black boxes with well defined interfaces. We represent black boxes as leaf level atomic tasks in the goal graph.

In Figure 4.7, the business process layer consists of three composite services depicted as goals: issue customers new card service \( (g_2) \), replace lost/stolen card service \( (g_3) \), and cancel card service \( (g_4) \). The issue customers new card service can be further decomposed into four sub services depicted as tasks: receive customers’ applications for new cards \( (a_1) \), issue customers temporary cards \( (a_2) \), issue customers permanent cards \( (a_3) \), and provide customers ATM service \( (a_4) \).

The middle layer, or the component layer, offers a medium level of abstraction. The leaf level tasks from the business process layer are “zoomed into”; here they are viewed as components with internal requirements that can be reasoned with. For example, the atomic task \( a_4 \) (provide customers ATM service) from the business process layer decomposes into the root goal \( g_5 \) (Manage ATM) in the component layer. \( G_5 \) is then further decomposed to the goal graph presented in Figure 4.3 (representing the requirements for
the ATM service), and the task $a_5$ (*Provide CPU*) (representing the requirements of the underlying infrastructure). The infrastructure layer, the third and bottom layer, represents the underlying servers, hardware devices, databases etc. required for the correct functioning of the upper layers. The entire infrastructure level goal graph can be treated as a black box at the component layer. In practice, each leaf level task at the component level depends on different parts of the infrastructure, and all these parts together form the entire infrastructure level goal graph. Thus satisfaction of the infrastructure level root goal $g_6$ (*provide CPU*) depends on the availability and the correct functioning of the physical ATM ($g_7$), the underlying connection between the ATM and the bank ($g_{11}$), and central bank ($g_{12}$).
Our framework can hierarchically monitor requirement satisfaction on each layer in isolation, or on all the layers as a connected global goal graph. The tradeoff lies between scalability and diagnostic precision. Monitoring at the business process layer offers the highest level of scalability because each component is treated as a black box. The framework infers requirement denials of black boxes as whole and does not look for root causes within them. Monitoring at the component level is less scalable, since each atomic task at the business process layer is treated here as a decomposable root goal. The benefit of monitoring at the component level is more precise diagnoses, pinpointing the source of the problem within the denied component. Monitoring at the infrastructure level is the least scalable. Here the framework not only analyzes denials of subcomponents, but also failures with the underlying infrastructure the component depends on. The benefit is that the diagnoses capture failures at a fine grained infrastructure level. Note that the domain expert specifies which layers are to be monitored by giving the framework their corresponding goal graphs.

4.6.1 Evaluation

In this section we discuss the scalability of our framework to goal model size at each layer of the SOA. The goal model shown in Figure 4.7 is a partial goal graph representing the requirements of 1 business process, provide ATM service, with its 3 SOA layers. We conducted 20 sets of experiments on 20 progressively larger goal models representing 1 to 20 business processes. Each of the 20 sets of experiments contained 3 experiments on 3 different goal graphs representing the 3 layers of SOA, for a total of 60 experiments. We generated the larger goal models by cloning the goal graph to itself. Table 4.5 lists the numbers of goals and tasks for all 60 experiments. Each row corresponds to one set of experiments. Column 1 lists the numbers of business processes. Columns 2 through 4 list the numbers of goals/tasks in the business process layer, the component layer, and the infrastructure layer respectively. Observe that the total numbers of goals/tasks increase
<table>
<thead>
<tr>
<th>#Business Processes</th>
<th>#Goals/Tasks at Business Process Level</th>
<th>#Goals/Tasks at Component Level</th>
<th>#Goals/Tasks at Infrastructure Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>50</td>
<td>173</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>100</td>
<td>346</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>150</td>
<td>519</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>200</td>
<td>692</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>250</td>
<td>865</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>300</td>
<td>1038</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>350</td>
<td>1211</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>400</td>
<td>1384</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>450</td>
<td>1557</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>500</td>
<td>1730</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>550</td>
<td>1903</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>600</td>
<td>2076</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>650</td>
<td>2249</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>700</td>
<td>2422</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
<td>750</td>
<td>2595</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>800</td>
<td>2768</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>850</td>
<td>2941</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
<td>900</td>
<td>3114</td>
</tr>
<tr>
<td>19</td>
<td>20</td>
<td>950</td>
<td>3287</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>1000</td>
<td>3460</td>
</tr>
</tbody>
</table>
as the level of abstraction decreases. All the experiments were performed with log file preprocessing (Algorithm 3). We injected one error into the log files of each experiment. In each experiment, all the tasks were monitored, and Algorithm 5 was used for diagnosis.

Figure 4.8 reports the results of the experiments. It depicts the relationship between the total time taken for diagnostic reasoning (in seconds) and the size of the goal model at the different layers of SOA (the x-axis). The figure shows that the framework is most efficient (takes the least amount of time) at the business process level, and is least efficient at the infrastructure level. Our data further confirm our claim that our framework scales to the size of the relevant goal graph and the number of diagnostic results it returns.

These experiments review that monitoring and diagnosing at higher layers of SOA can be effective. When the system is running correctly, monitoring at the business process level is advisable. Only if the system is not running correctly should the framework monitor at lower layers of SOA. Multi-layered monitoring and diagnosis allows our framework to model and analyze software systems at different levels of granularity, as appropriate in the circumstances. This enables our framework to be applied effectively to larger scale software systems.
4.7 Discussions

This chapter presents our work on requirement monitoring and diagnosis. Our approach is based on several important assumptions. Firstly, we assume that the system’s goal model is available. In addition, we assume that the goal model is correct and complete. The goal model specifies the requirements a system is supposed to fulfill. We verify that a system is in compliance with its requirements model. The diagnoses inferred by our approach would not be very useful if these specified requirements are not correct or complete. We infer denials of goals/tasks using truth values of its occurrence, precondition and effect. We assume that goals’/tasks’ associated preconditions and effects are also correct and complete. We expect domain experts to provide such an annotated goal model. In addition, we expect domain experts to provide traceability links that link between source code and goal model. These traceability links give us information on where the monitors should be inserted in the source code.

It is often difficult to provide a fine-grained goal model with traceability links for large software applications. To alleviate this problem, one can model a system at a high granularity level with few goals and tasks. When this is the case, traceability links link between high level goals/tasks to large-scaled subsystems, instead of simple methods and functions.

We also assume that it is possible for the instrumentation component to generate all the log data needed for diagnosis, and the log data are correct. Our diagnostic component analyzes log files that contain log traces for one execution session at a time. [Zho08] provided an AspectJ based approach to instrumenting a system using goal models. It is beyond the scope of this thesis to verify that the log data are correct. Finally, we have assumed that the software systems that we are interested in are diagnosable.

We have evaluated the performance of our diagnostic component. Our experimental results show that our diagnostic approach scales to the size of the goal model when encoding is done with log file preprocessing and when we infer participating diagnostic
components (instead of core diagnoses). Participating diagnostic components are weaker forms of diagnoses than the ones that are commonly understood in the AI literature. In AI literature, a diagnosis usually specifies, for each system component and/or action, whether it is normal or faulty. Our definition of a core diagnosis (specifies for each task whether it is denied or not) is consistent with this notion. The advantage of finding core diagnoses is that they contain more diagnostic information such as which tasks can or cannot fail together. In contrast, participating diagnostic components are individual tasks that might have failed. We lose all other diagnostic information. Clearly, the disadvantage of finding core diagnoses is that it scales poorly to the size of the system model. When deciding between finding core diagnoses and participating diagnostic components, a compromise need to be made between scalability and diagnostic precision.

Our diagnostic component is off line because it uses a SAT solver, and SAT solvers are off line in nature. The SAT solver takes as input log data and a model of the system. We need to wait for a pool of log data to be collected before invoking the solver. Each log file contains log traces for one execution session on the entire goal model. We can reduce the size of the log file to contain only log traces for a partial goal model, and we infer whether or not all goals/tasks in the partial goal model are satisfied. We reduce the time interval between calls to the solver, but increase the number of calls. By doing this, we make the diagnostic component more “online like” - it analyzes data and gives feedback more frequently at smaller time intervals. Of course, this does not change the off line nature of the diagnostic component. We need to explore other diagnostic techniques to make diagnosis completely online.

Our axiomatization is based on propositional logic. Therefore our approach is restricted by the reasoning power of propositional logic. The correct behavior of goals/tasks are described through their preconditions and effects in CNF. We can not easily describe temporal relations in these preconditions and effects. For example, consider task a’s with effect: “a’s execution time must be less than 5 seconds”. The only way to describe
this constraint using propositional logic is to introduce a predicate such as `respondless-than5seconds` that is either true or false. We rely on the instrumentation component to instrument the software in such a way that the predicate’s correct truth value is logged at runtime. The advantage of using propositional logic is that we can use a SAT solver for analysis. SAT solvers are efficient in searching through a relatively large knowledge base with many clauses and literals.

Finally, our approach is restricted by the expressiveness of the goal model itself. Goal models describe requirements of a system such as which tasks need to be executed to fulfill a goal. They cannot express the order of task executions, neither can they express the number of times a goal/task needs to be executed. The benefit that comes from goal model’s relatively simple notation is that domain experts can learn how to use goal models with little effort.
Algorithm 5 Find All Participating Diagnostic Components

\textbf{find\_all\_participating\_diagnostic\_components}(\Phi) \{
    \text{while (\Phi is satisfiable) \{} \\
    \hspace{1em} \mu = \text{satisfying assignments for all variables in } \Phi \\
    \hspace{1em} //\text{map SAT result to diagnostic instance} \\
    \hspace{1em} \text{oneDiagnosis} = \text{decodeToDiagnosis}(\mu) \\
    \hspace{1em} //\text{complete failure configuration containing session} \\
    \hspace{1em} //\text{level task satisfactions and denials in oneDiagnosis} \\
    \hspace{1em} \text{oneCoreDiag} = \text{session level task satisfactions and denials in oneDiagnosis} \\
    \hspace{1em} //\text{partial failure configuration containing only task denials} \\
    \hspace{1em} \text{partialCoreDiag} = \text{task denials in oneCoreDiag} \\
    \hspace{1em} \text{allCoreDiag} = \text{allCoreDiag} \land \text{oneCoreDiag} \\
    \hspace{1em} //\text{calculate which part of } \mu \text{ to add back to the SAT solver} \\
    \hspace{1em} \text{boolean negateCompleteConfig} = \text{false}; \\
    \hspace{1em} \text{for (each } FD(task_i, s) \text{ in oneCoreDiag)} \\
    \hspace{1em} \hspace{1em} \text{if (} task_i \text{ is associated with a contribution link) and} \\
    \hspace{1em} \hspace{1em} \hspace{1em} \text{if (} FD(task_i, s) \text{ is not already part of a core diagnosis)} \\
    \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \text{negateCompleteConfig} = \text{true}; \\
    \hspace{1em} \text{if (negateCompleteConfig)} \\
    \hspace{1em} \hspace{1em} //\text{add to } \Phi \text{ the negation of complete failure configuration} \\
    \hspace{1em} \hspace{1em} \Phi = \Phi \land \lnot\mu_{\text{oneCoreDiag}}; \\
    \hspace{1em} \text{else //add to } \Phi \text{ the negation of partial failure configuration} \\
    \hspace{1em} \hspace{1em} \Phi = \Phi \land \lnot\mu_{\text{partialCoreDiag}}; \} \\
    \text{AllParticipatingComps} = \text{filter(allCoreDiag);} \}
Chapter 5

Monitoring and Diagnosis Using Statecharts

5.1 Preliminaries

5.1.1 State Charts

Statechart [Har87, Har96] is a graphical and semi-formal language for describing “reactive systems” - systems that respond to events rather than transforming an input to an output. The statechart language is compact and expressive both because it uses notions of concurrency and also because its hierarchical representations allow states to be refined. It can describe complex software systems at different levels of abstraction, permitting more comprehensive specifications of large systems.

A statechart contains a finite collection of states and transitions. There are three types of states in a statechart: AND-states, OR-states, and basic states. AND- and OR-states are hierarchical states with substates. AND-states have orthogonal components (substates) related to each other by “and”, meaning that all substates can be active concurrently. OR-states have substates related to each other by “exclusive-or”, meaning that only one substate can be active at a time. Basic states sit at the bottom of the state
hierarchy and are not decomposable. Consider the example given in Figure 5.1. States $S$, $C$, $D$, and $E$ are hierarchical states with substates. All other states are basic states. State $C$ is an AND-state (with a “swim-lane” dividing its substates) because both of its substates $D$ and $E$ can be active concurrently. States $S$, $D$ and $E$ are OR-states because only one of their substates can be active at a time.

There are two kinds of transitions/actions: state transition actions that link states, and actions that reside inside states. A state transition action $e[c]/a$ is a tuple with an event $e$, a condition $c$, and an action $a$. It relates a source state to a destination state. A state transition action is enabled when the system is in its source state, event $e$ is generated, and condition $c$ is true. When this happens, action $a$ is carried out and the system transitions from the source state to the destination state. A special boolean condition $in(S)$ indicates whether the system is in some state $S$. A status of a statechart consists of two components: a set of states in which the computation currently resides (also called active states) and a set of currently present events. The set of all active states are referred to as a system configuration. An event is generated after each action execution. Therefore, action occurrences can be treated as events.

Transitions can also occur within a state. Our work deals with two such transitions: state entry and state exit actions. State entry and state exit transitions are actions executed when a system enters and exits a state respectively. A statechart describes how
a system “reacts” to changes and events in its environment. The statechart semantics [Har96] defines that: (1) events that occur in a timestep $t$ can only be sensed at the subsequent timestep $t+1$; and (2) calculations in a timestep $t$ take account solely of the situation in the previous timestep $t-1$.

Consider the example in Figure 5.1. Assume the system is in state $A$ at timestep $t$, event $e1$ occurred at $t$, and condition $c1$ is true at $t$. The system responds as follows: the transition $t4$ becomes enabled; action $a4$ is executed; the system exits $A$ and enters $S$ and $B$; the special condition $in(A)$ becomes false; and conditions $in(S)$ and $in(B)$ become true. Any exit actions associated with $A$ and any entry actions associated with $S$ and $B$ are executed.

Domain experts can associate transitions with preconditions, postconditions, and monitoring switches. Preconditions and postconditions are propositional formulae in Conjunctive Normal Form (CNF) that must be true before and after (respectively) a transition is successfully executed. Monitoring switches can be switched on/off to indicate whether the transition is to be monitored. We also associate states with state condition formulas and monitoring switches. State condition formulas are propositional formulas in CNF that express domain constraints and assumptions that must hold if the system is in a given state. Monitoring switches can be turned on/off to indicate whether the corresponding state is to be monitored at run time. We assume the availability of traceability links between a system’s source code and statechart.

### 5.2 A Running Example

We use an ATM simulation [Bjo] case study as a running example to illustrate our framework. This medium-size public domain software system illustrates the OO design used in a software development class at Gordon College. The application simulates an ATM which performs withdraw, deposit, transfer, and balance inquiry transactions. The source
code contains 36 Java Classes with 5000 LOC. Figure 5.2 shows a high level statechart for the ATM which contains three high level states: Off, Idle, and Serving Customer. These three states represent, respectively, the states in which the ATM is switched off, is switched on but idle, and is serving a customer. The statechart describes the following dynamic behaviors of the ATM. The ATM’s initial state is Off. Upon entering Off, the machine displays the message “ATM not available”. If the event switch_on occurs while the machine is in Off, transition tr1 is enabled. The ATM then performs the perform_start_up action, exits Off, and enters Idle. On entering Idle, the machine displays
the message “please insert bank card”. If the event \texttt{card\_inserted} occurs while the machine is in \textit{Idle}, transition \textit{tr3} is enabled. The ATM then executes the \texttt{create\_session} action, exits \textit{Idle}, and enters the \textit{Serving Customer} state.

\textit{Serving Customer} is a hierarchical OR-state with substates. The ATM reads the customer’s card number, reads the pin, asks the customer to choose a transaction and performs the chosen transaction. In the \textit{Perform Transaction} substate, the ATM checks that the entered pin number and the chosen transaction specifics are valid, deals with any invalid pin, completes all transactions that are valid, and ejects the bank card.

### 5.3 Monitoring

Our framework monitors for successful executions of transitions as well as satisfactions of domain constraints associated with states. States and transitions are annotated with monitoring switches that can be turned \textit{on} or \textit{off} to indicate whether the states/transitions are monitored at run time. Monitored states need to be associated with condition formulas. Domain experts can (but do not have to) associate transitions with pre- and post- conditions that are propositional formulas in CNF.

Thanks to the hierarchical nature of statecharts, the satisfaction of domain constraints (specified as state conditions) can be monitored at different levels of granularity. The finest granularity monitors all basic states. We can infer more precise diagnoses (which pinpoint denied basic states) because more complete log data are generated. Unfortunately this entails high monitoring overhead. Coarser levels of granularity only monitor higher-level hierarchical states. We generate less precise diagnoses (corresponding to denied hierarchical states). The advantage is reduced monitoring complexity and overhead.

We introduce predicate \textit{occ\_trans}(tr_i, t) to express occurrences of a transition \textit{tr}_i at timestep \(t\). When a transition occurs, three kinds of actions associated to the transition must occur: (1) transition actions that are in \textit{tr}'s \textit{e[c]/a} tuple - they are executed when
tr is enabled; (2) exit actions of tr’s source states; and (3) entry actions of tr’s target states. We introduce a predicate \( occ(a_i, t) \) to express the occurrence of an actions or an events \( a_i \) at timestep \( t \). We use predicates \( enabled(tr_i, t) \) to express enabledness of transition \( tr_i \) at timestep \( t \). A transition is enabled if its source states are active, its triggering events have occurred, its guarding condition is true, and any domain expert added precondition is true. We use predicate \( effect(tr_i, t) \) to represent effect of \( tr_i \) at \( t \). Tr’s effect is true if the system’s state changes are correct, and any domain expert added post-condition is true.

Statechart semantics [Har96] uses a special boolean condition \( in(S) \) to indicate that the system is in state \( S \). We extend this predicate with a time parameter \( t \), introducing predicate \( in(S, t) \) to indicate that the system is in state \( S \) at timestep \( t \). The monitored system’s runtime behavior is traced and recorded as log data consisting of (1) truth values of observed domain literals (specified in state condition formulas and transitions’ guarding conditions, preconditions and effects), (2) the occurrence of transitions, actions, and events, and (3) state information predicates (in the forms of \( in(S_i, t_j) \)) indicating which state(s) the system was in at a give timestep. Each log instance is associated with a specific timestep \( t \). A log comprises a sequence of log instances, where log instances are defined as follows:

**Definition 7 (Log instance)** A log instance is either the truth value of an observed literal at a specific timestep \( t \), or the occurrence of a transition, an event, or an action at \( t \), or the truth value of a state information predicate indicating which state the system is in at \( t \).

For example, if literal \( l \) was true at timestep 1, transition \( tr_1 \) and action \( a_1 \) occurred at timestep 2, and the system was in state \( S \) at timestep 3, their respective log instances are: \( l(1) \), \( occ_{trans}(tr_1, 2) \), \( occ(a_1, 2) \), and \( in(S, 3) \).
5.3.1 ATM Example Log Data

<table>
<thead>
<tr>
<th>State</th>
<th>Switch</th>
<th>State Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>on</td>
<td>¬ atm_on</td>
</tr>
<tr>
<td>Idle</td>
<td>on</td>
<td>atm_on ∧ ¬ serving_customer</td>
</tr>
<tr>
<td>Serving Customer</td>
<td>on</td>
<td>serving_customer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Action</th>
<th>Switch</th>
<th>Precondition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>tr1</td>
<td>on</td>
<td>¬ atm_on</td>
<td>atm_on ∧ connection_to_bank</td>
</tr>
<tr>
<td>tr3</td>
<td>on</td>
<td>atm_on</td>
<td>session_created</td>
</tr>
<tr>
<td>tr4</td>
<td>on</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Tables 5.1 and 5.2 list respectively details of monitored states and transitions in the ATM statechart (Figure 5.2). Table 5.1 lists monitored states with their monitoring switches (Column 2) and associated state condition formulas (Column 3). Table 5.2 lists monitored transitions with their monitoring switches (Column 2), and domain expert added preconditions and effects (Columns 3 and 4). High level states (Off, Idle, and Serving customer) and state transitions (tr1, tr3, and tr4) are monitored in this example. Note that neither precondition nor effect is specified for transition tr4.

We analyze the successful execution of a transition using information about its occurrence, occurrences of its associated actions, its source and destination states, the truth values of its trigger conditions, and its domain expert added preconditions and effects. The following example log data consists of (1) truth values of domain literals, (2) occurrences of transitions, triggering events, and actions, and (3) state information predicates that records which states the system was in:
in(Off, 1), ¬atm_on(1), occ(switch_turned_on, 1), occ_trans(tr1, 2), 
occ(perform_start_up, 2), occ(a2, 2), connection_to_bank(3), in(Idle, 3), 
atm_on(3), ¬serving_customer(3), occ(card_inserted, 3), occ(create_session, 
4), occ_trans(tr3, 4), ¬session_created(5), in(Serving customer, 5), serv-
ing_customer(5), session_completed(6), in(Idle, 7), atm_on(7), serv-
ing_customer(7).

Note that we use occ predicates to specify the occurrences of both trigger events 
and transition actions. For example, in the above log, occ(switch_turned_on, 2) indicates 
that transition a1’s trigger event switch_turned_on occurred at timestep 2. Similarly, 
occ(perform_start_up, 2) indicates that action perform_start_up also occurred at timestep 
2.

5.4 Diagnosis

This section presents the formal foundations of our diagnostic component when it is based 
on statecharts. In [Har96], Harel gave a schematic description of a “basic step algorithm” 
that explains what is executed in a single step. Mikk et al. formalized this algorithm using 
Z notations [MLP97]. Our axiomatizations of transition denials are based on a subset of 
the semantics defined by Harel [Har96] and Mikk et al. [MLP97]. The axiomatizations 
generated for diagnostic reasoning (presented in Sections 5.4.2, 5.4.4, and 5.4.5) are 
adaptations of the theoretical diagnostic framework proposed in [Rei87, dKMR92, McI98].

We monitor for the satisfaction of states and transitions. We use FD predicates to 
express full evidence of transition or state denials at a certain timestep. An FD predicate 
is associated with two parameters. The first parameter is either a transition or state in 
the statechart, and the second is a timestep. For example, if state S is denied at timestep 
2, and transition tr is denied at timestep 5, the diagnostic component infers FD(S, 2) 
and FD(tr, 5) respectively. We use ¬FD predicates to express full evidence of a state or 
a transition is not denied at a certain timestep. For example, predicates ¬FD(S, 1) and
\neg FD(tr, 6) indicate that state S and transition tr are not denied at timesteps 1 and 6 respectively.

5.4.1 Basic formulation for SAT

We reduce the problem of searching for diagnoses to that of the satisfiability of a propositional formula $\Phi$. $\Phi$ is written in the form:

$$\Phi := \Phi_{LOG} \land \Phi_{deniability}$$

(5.1)

The first component $\Phi_{LOG}$ represents log data generated by the monitoring component as specified in Definition 7 (Section 5.3). The second component $\Phi_{deniability}$ encodes transition and state denials (Sections 5.4.2, 5.4.3, 5.4.4, and 5.4.5).

5.4.2 Axiomatization of Transition Denials

Axiom (4) describes deniability for transitions that link between states. Recall that a transition $tr$ is an $e[c]/a$ tuple. $Tr$ is enabled if its source state(s) are active, its triggering event(s) occurred, its guarding condition is true, and any domain expert added precondition is true. When this happens, $tr$’s associated action $a$ is carried out. The system exits $tr$’s source state(s) $S$, and enters $tr$’s target state(s) $T$. Any exit actions associated with $S$, and entry actions associated with $T$ are also executed.

Axiom 4 (Transition Denial Axiom) Consider a state transition $tr : e[c]/a$ with triggering event(s) $e$, guarding condition $c$, and transition action(s) $a$. Let $S$ be $tr$’s source state(s); let $T$ be $tr$’s target state(s); let exited be all exit actions of the states in $S$; and let entered be all the entry actions of the states in $T$. Let $p_{add}$ and $q_{add}$ be $tr$’s domain expert added precondition and effect.

Transition $tr$ is enabled at timestep $t$ if and only if the system is in $tr$’s source state(s) $S$ at $t$, $e$ occurred at $t$, $c$ is true at $t$, and $p_{add}$ is true at $t$ (Equation (5.2)). $Tr$’s effect
effect(tr) is true at t+2 if and only if the system is in tr’s target state(s) T at t+2, and qadd is true at t+2 (Equation 5.3). Occ_{trans}(tr, t+1) is true if and only if all transition actions associated to tr occurred at t+1, and all actions belonging to exited and entered occurred at t+1 (Equation 5.4). Tr is fully denied at timestep t+2 if and only if: (1) tr is not enabled at timestep t and its effect is true at timestep t+2; or (2) tr is enabled at timestep t, and it did not occur at timestep t+1; or (3) tr occurred at timestep t+1, but its is false at timestep t+2 (Equation 5.5).

\begin{align}
\text{enabled}(tr, t) & \leftrightarrow \bigwedge_i in(S_i, t) \land \text{occ}(e, t) \land c(t)[\land p_{add}(t)] \tag{5.2} \\
\text{effect}(tr, t+2) & \leftrightarrow \bigwedge_j in(T_j, t + 2)[\land q_{add}(t + 2)] \tag{5.3} \\
\text{occ}_{trans}(tr, t + 1) & \leftrightarrow \bigwedge_w \text{occ}(a_w, t + 1) \land \bigwedge_j \text{occ}(\text{exited}_m, t + 1) \land \bigwedge_n \text{occ}(\text{entered}_n, t + 1) \tag{5.4} \\
FD(tr, t + 2) & \leftrightarrow (\neg \text{enabled}(tr, t) \land \text{effect}(tr, t + 2)) \lor \\
\quad (\text{enabled}(tr, t) \land \neg \text{occ}_{trans}(tr, t + 1)) \lor (\text{occ}_{trans}(tr, t + 1) \land \neg \text{effect}(tr, t + 2)) \tag{5.5}
\end{align}

We formulate the denial of transitions in terms of the truth values of the predicates representing their enabledness, occurrences, and effects. Intuitively, if a transition tr is enabled at timestep t, tr fired at the subsequent timestep t+1 (meaning all of tr’s associated actions occurred at t+1), and if tr’s effect holds at t+2, then the transition is not denied at timestep t+2.

Three scenarios describe transition denial: (1) if a transition is not enabled at timestep t, but its effect is true (i.e. state update occurred) at timestep t+2; or (2) if a transition is enabled at timestep t, but the transition did not occur at t+1; or (3) if a transition’s associated actions occurred at timestep t+1, but its effect is false at the subsequent timestep t+2. Equation (5.5) under Axiom (4) captures all of these cases.

Statechart semantics [Har96] requires that an action’s precondition, occurrence, and effect lie within three consecutive timesteps. This follows from the facts that changes at

\footnote{In many axiomatizations it is assumed enabled(tr) → occ_{trans}(tr), where enabled(tr) represents tr is enabled and occ_{trans}(tr) represents tr occurs
t can only be sensed at the subsequent timestep \( t+1 \), and that calculations at \( t \) only take account of values from the previous timestep \( t-1 \). The domain expert added preconditions and effects are in CNF. All propositional literals are grounded to domain instances. For example, a propositional literal \( a \), representing an action, may be grounded to action instance \textit{read\_bank\_card} in an ATM domain.

Consider transition “\textit{tr1: switch on / perform start up}” from the ATM case study (Figure 5.1). \textit{Tr1}’s source and destination states are \textit{Off} and \textit{Idle} respectively. Its domain expert added precondition and effect are \( \neg\text{atm\_on} \) and \( \text{atm\_on} \land \text{connection\_to\_bank} \) respectively (Table 5.2). The following axioms are generated for \( \textit{tr1} \) for timesteps 1, 2, and 3:

\[
\begin{align*}
\text{enabled}(\textit{tr1}, 1) & \iff \text{in}(\textit{Off}, 1) \land \text{occ}(\textit{switch\_on}, 1) \land \neg\text{atm\_on}(1) \\
\text{effect}(\textit{tr1}, 3) & \iff \text{in}(\textit{Idle}, 3) \land \text{atm\_on}(3) \land \text{connection\_to\_bank}(3) \\
\text{occ}_{\text{trans}}(\textit{tr1}, 2) & \iff \text{occ}(\textit{perform\_start\_up}, 2) \land \text{occ}(a2, 2) \\
\text{FD}(\textit{tr1}, 3) & \iff (\neg\text{enabled}(\textit{tr1}, 1) \land \text{effect}(\textit{tr1}, 3)) \vee \\
(\text{enabled}(\textit{tr1}, 1) \land \neg\text{occ}_{\text{trans}}(\textit{tr1}, 2)) \lor (\text{occ}_{\text{trans}}(\textit{tr1}, 2) \land \neg\text{effect}(\textit{tr1}, 3))
\end{align*}
\]

5.4.3 Foundations of Statechart Semantics

Harel’s definition of statechart semantics [Har87, Har96] is informal in the sense that semantics are defined in natural language instead of formal notations. Harel described a “basic step algorithm” that describes what is executed in a single step. The “basic step algorithm” underlines statechart semantics and it lies at the heart of the dynamic analysis tools of STATEMATE. In [MLP97], Mikk et al. formalized the algorithm using Z notations. We show in this section that our axiomatizations of deniability of transitions are based on a subset of the semantics defined in the “basic step algorithm”. Specifically, the subset of statechart semantics that we base our work on is the semantics on transition firing, and changes in the system due to transition firing (such as state changes, and
executions of actions associated to the transition). We do not deal with conflicts, priority, nondeterminism, or concurrency\(^2\).

Harel’s work specifies that a transition \( tr \) is enabled if the system is in \( tr \)’s source state(s), \( tr \)’s triggering event(s) occur(s), and \( tr \)’s guarding condition is true. Figure 5.3 shows Mikk et al.’s formalization of the enabledness of a transition \( tr \) w.r.t. configuration \( conf \) and event set \( env \)[MLP97]. \( Conf \) represents all active states, and \( env \) represents all present events. In [MLP97], occurrences of a transition’s associated actions are treated as events. Lines 3 to 5 in Figure 5.3 defines that a transition \( tr \) is enabled if and only if the transition’s source state(s) are active (the system is in \( tr \)’s source state(s))(line 3), \( tr \)’s triggering events \( tr.label.event_expr \) evaluate to true (line 4), and \( tr \)’s guarding condition \( tr.label.condition \) evaluates to true (line 5). We encode the enabledness of a transition in Equation (5.2) under Axiom (4). Our respective formalism on transition enabledness is equivalent.

Figure 5.4 lists Mikk et al.’s formalization of Harel’s “basic step algorithm” that describes what is executed in a single step. The following is a citation from [MLP97], and it describes the algorithm in Figure 5.4.

- Compute the set of enabled transitions (correspond to the set \( ET \) ) (line 1 of Figure 5.4).

\[^2\]Our work verifies the successful executions of each concurrent transitions, but it does not verify that these transitions did occur concurrently.
Figure 5.4: Basic Step Algorithm[MLP97]

- Remove from $ET$ all transitions that are in conflict with an enabled transition of higher priority (corresponds to the set $HPT$) (line 2).

- Split the set of enabled transitions into maximal non-conflicting sets (corresponds to the set $MNS$) (lines 3 and 4).

- If there are no enabled transitions then the step is empty (line 5) else choose one of the sets nondeterministically for execution. Let $EN$ be the choice (line 6).

- For each transition $Tr$ in $EN$ let $Exited$ be the set of states exited and $Entered$ be the set of states entered by $Tr$ (lines 7 and 8).
  
  - update the set of active states (correspond to the set $csts$) by removing $Exited$ and adding $Entered$ (line 9).

  - update the event list (correspond to the set of events) by adding executions of actions associated to $Tr$. These actions are: (1) transition actions labeled on $Tr$’s $e[c]/a$ tuple (line 10), (2) exit actions associated to states in $Exited$ (line 11), and (3) state entry actions associated to states in $Entered$ (line 12).

- increment the step (line 13).
We use a subset of the semantics defined in Figure 5.4. Specifically, we do not deal with conflict, priority, nondeterminism or concurrency. Therefore lines 2 to 4 of the algorithm collapses. Hence, HPT and MNS is the same as ET (set of enabled transitions). The subset of the semantics that we deal with includes transition firing, state change due to transition firing, and action executions due to transition firing (lines 5 to 13 of the algorithm).

The basic step algorithm states that if a transition is enabled, then the sets of active states change and a set of actions are carried out. Again, these actions are the actions associated to the transition and to the states that are entered and exited. If no transitions are enabled, nothing should happen. Mathematically, we have:

\[(\text{enabled} \rightarrow \text{occ}_{\text{trans}} \land \text{effect}) \land (\neg\text{enabled} \rightarrow \neg\text{occ}_{\text{trans}} \land \neg\text{effect})\]

\[\equiv (\neg\text{enabled} \lor (\text{occ}_{\text{trans}} \land \text{effect})) \land (\text{enabled} \lor (\neg\text{occ}_{\text{trans}} \land \neg\text{effect}))\]

\[\equiv (\text{enabled} \land \neg\text{enabled}) \lor (\neg\text{enabled} \land \neg\text{effect} \land \neg\text{occ}_{\text{trans}})\]

\[\lor (\text{enabled} \land \text{effect} \land \text{occ}_{\text{trans}}) \lor (\text{effect} \land \text{occ}_{\text{trans}} \land \neg\text{effect} \land \neg\text{occ}_{\text{trans}})\]

\[\equiv (\text{enabled} \land \text{occ}_{\text{trans}} \land \text{effect}) \lor (\neg\text{enabled} \land \neg\text{occ}_{\text{trans}} \land \neg\text{effect})\]

We establish that for every transition firing, the formula \((\text{enabled} \land \text{occ}_{\text{trans}} \land \text{effect}) \lor (\neg\text{enabled} \land \neg\text{occ}_{\text{trans}} \land \neg\text{effect})\) should evaluate to true. If this formula is false, then a transition failure has occurred.

\[-FD \leftrightarrow (\text{enabled} \land \text{occ}_{\text{trans}} \land \text{effect}) \lor (\neg\text{enabled} \land \neg\text{occ}_{\text{trans}} \land \neg\text{effect})\]

The above formula can be represented by the Karnaugh map [Kar] shown in Table 5.3. The Karnaugh map contains three variables: enabled, effect, and occ\(_{\text{trans}}\). On the top left side of the grid, the first “00” represents variables enabled and effect are both false. In the cell to its right, the “01” values represents enabled is false and effect is true and so forth. There are 8 permutations out of the three variables, and thus 8 outputs in

\(^3\text{In Mikk et al.’s work, an event is generated after action execution. Therefore, action executions are treated as events.}\)

\(^4\text{We are omitting parameters to predicates for readability}\)
the Karnaugh map. Each output represents the truth value of \( FD \) predicate (full evidence a transition is denied) under the given values for \( enabled \), \( occ_{trans} \) and \( effect \). A “0” means \( \neg FD \), and a “1” means \( FD \).

Table 5.3: Karnaugh Map for Transition Denial

<table>
<thead>
<tr>
<th>( occ_{trans} )</th>
<th>enabled</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

There are two minimal covers for this Karnaugh map. The first minimal cover is: 
\[
(\neg enabled \land occ_{trans}) \lor (\neg occ_{trans} \land effect) \lor (enabled \land \neg effect).
\]
The second minimal cover is: 
\[
(\neg enabled \land effect) \lor (enabled \land \neg occ_{trans}) \lor (occ_{trans} \land \neg effect).
\]
Our transition action denial axiom (Axiom (5.5)) is the second minimal cover with predicates expanded with two parameters: a transition id and a timestep. We have shown in this section that our axiomatization of transition denials is based on exiting statechart semantics provided by Harel [Har96] and Mikk et al [MLP97].

5.4.4 Axiomatization of state denials

**Axiom 5 (State Denial Axiom.)** A state \( S \) with state condition \( c \) is denied at timestep \( t + 1 \) if and only if the system is in \( S \) at \( t \), and \( c \) is false at \( t \).

\[
FD(S, t + 1) \leftrightarrow in(S, t) \land \neg c(t) \tag{5.6}
\]

We associate domain constraints to states and monitor their satisfaction when the system in a given state. We formulate state denials much as we do transition denials: a state \( S \) is denied at \( t + 1 \) if and only if \( S \) “occurred” at \( t \) (expressed as \( in(S, t) \)), and its condition \( c \) is false at \( t \). Recall that state conditions are propositional formulas in CNF.
In the ATM case study, state *Off* is associated with condition “¬*atm_on*”. The following axiom is generated for *Off* at timesteps 1 and 2:

\[ \text{FD}(\text{Off}, 2) \iff \text{in}(\text{Off}, 1) \land \text{atm_on}(1) \]

State denial axiom (Axiom (5)) verifies satisfaction of domain constraints/assumptions associated with states as state conditions formulas.

### 5.4.5 Explanation closure axioms

*Fluents* are propositional literals whose values may vary from timestep to timestep. For example, literal *atm_on* from the ATM domain is a fluent. If fluent \( f \) is not mentioned in the effect of an action executed at timestep \( t \), we would not know the value of \( f \) at \( t + 1 \). We therefore need axioms to specify which fluents remain unaffected when an action is performed. *Frame axioms* are formulas that specify unaffected fluents retain the same values. The number of necessary frame axioms often fails to scale with the size of the Knowledge Base (KB), since frame axioms are needed for every fluent, action, and timestep in the KB.

We adopt Explanation Closure Axioms [Rei91] to address this problem. We assume that the effects specified for actions characterize all conditions under which fluents change their truth values. Therefore, if the value of fluent \( f \) changes at timestep \( t \), then one of the transitions that has \( f \) in its effect must have occurred at a previous timestep \( t - 1 \) and not have been denied at \( t \).

Axioms (5.7) and (5.8) are Explanation Closure Axioms which state that, for any fluent \( f \) that is in a positive (or negative) effect of transition \( tr_1, \ldots, tr_n \), if \( f \) does not hold (or does hold) at timestep \( t \), but holds (or does not hold respectively) at \( t + 1 \), then one of the transitions \( tr_i \) must have occurred at \( t \) and not have been denied at \( t + 1 \).

If \( f \) is in a positive effect of transitions \( tr_i \ (i \in [1 \ldots n]) \),

\[ \neg f(t) \land f(t + 1) \iff \bigvee_{i}(\text{occ}_{\text{trans}}(tr_i, t) \land \neg \text{FD}(tr_i, t + 1)) \quad (5.7) \]
If $f$ is in a negative effect of transition actions $tr_i \ (i \in [1\ldots n])$,

$$f(t) \land \neg f(t + 1) \Leftrightarrow \bigvee_i (occ_{trans}(tr_i, t) \land \neg FD(tr_i, t + 1)) \tag{5.8}$$

For example, in the ATM case study, only transition action $tr3$: card_inserted/create_session can change the truth value of fluent session_created from being false to being true. We say that fluent session_created is in $tr3$’s positive effect. We generate the following explanation closure axiom for that fluent for timesteps 1 and 2:

$$\neg session\_created(1) \land session\_created(2) \leftrightarrow occ_{trans}(tr3, 1) \land \neg FD(tr3, 2)$$

The special “in state” predicates, $in(S,t)$, are also fluents, because their truth values can change from timestep to timestep. Consider again the ATM case study. Two transitions allow the system to transit into state Idle: $tr1$ transits from Off to Idle, and $tr4$ transits from Serving customers to Idle. If the system was not in Idle at $t$, but is in Idle at $t+1$, then either $a1$ or $a4$ must have been executed and not denied at $t$ and $t+1$ respectively. The following explanation closure axiom is generated for timesteps 1 and 2:

$$\neg in(Idle, 1) \land in(Idle, 2) \leftrightarrow (occ_{trans}(tr1, 1) \land \neg FD(tr1, 2)) \lor (occ_{trans}(tr4, 1) \land \neg FD(tr4, 2))$$

The conjunction of the axioms presented in Sections 5.4.2, 5.4.4 and 5.4.5 encodes the $\Phi_{deniability}$ component of the propositional formula $\Phi$ (Equation 5.1): they describe deniability of transitions and states.

### 5.4.6 Characterizing diagnoses

**Definition 8 (Diagnosis)** A Diagnosis $D$ for a software system is a set of $FD$ and $\neg FD$ predicates over all the transitions and states in a statechart, indexed with respect to timesteps, such that $D \cup \Phi$ is satisfiable.

For example, consider a statechart with two states $W$ and $U$, and one transition action $a1$ that links between them. Supposing a total of 3 timesteps, and that $a1$ is denied at
timestep 3, the system’s diagnosis would contain: \( \neg FD(W, 1) \), \( \neg FD(W, 2) \), \( \neg FD(W, 3) \), 
\( \neg FD(U, 1) \), \( \neg FD(U, 2) \), \( \neg FD(U, 3) \), \( \neg FD(a1, 1) \), \( \neg FD(a1, 2) \), \( FD(a1, 3) \).

5.4.7 Algorithms

This section discusses the two main algorithms of our diagnostic component: an encoding algorithm (Algorithm 6) that encodes the SAT propositional formula, \( \Phi \), and a diagnostic algorithm (Algorithm 7) that finds a correct diagnosis.

Our previous work [WMYM07, WMYM09] demonstrates that the encoding algorithm must encode with log file preprocessing in order for \( \Phi \) to scale. Otherwise the encoding algorithm generates a complete set of axioms for all timesteps. This prevents the size of \( \Phi \) from scaling well with respect to the size of the input model. In contrast, when encoding makes use of log file preprocessing, the algorithm only generates all necessary axioms using timesteps that appear in the log. This keeps the growth of \( \Phi \) polynomial with respect to the size of the input model. For this reason Algorithm 6 incorporates log file preprocessing.

Algorithm 6 identifies all occurrence timesteps for each monitored transition \( tr \). One transition can occur multiple times in the log, and we infer the success or failure of each occurrence independently. For each of \( tr \)’s occurrence timesteps \( t_{occ} \), the algorithm calculates two additional timesteps: those at which \( a \)'s enabledness (precondition) and \( a \)'s effect ought to appear in the log. Our framework requires that the timesteps for an action’s precondition, occurrence, and effect occur consecutively. That is to say, if \( a \) occurs at timestep \( t_i \), the truth values of \( a \)'s precondition and effect must appear in the log at timesteps \( t_i-1 \) and \( t_i+1 \), respectively. This follows from the fact, mentioned earlier, that statechart semantics provide that events “live” for one timestep only. Calculations at a timestep only take account of values from the previous timestep. The algorithm generates transition denial axioms only for the timestep at which the action actually occurred (as recorded in the log).
Algorithm 6 Encode Φ with log preprocessing

```
encode_Φ_with_log_preprocessing(statechart, log) {
  for each monitored transition tr {
    occ_trans = timesteps at which tr occurred in log
    for each timestep t_occ in occ_trans {
      //timesteps of enabledness and effect
      t_enabled = t_occ - 1
      t_effect = t_occ + 1
      Φ = Φ ∧ encodeTransitionDenialAxiom(tr, t_enabled, t_occ, t_effect)
    }
  }
  for each monitored state S {
    cond = S’s state condition formula
    in_state_times = timesteps at which system is in S in log
    for each timestep t_in in in_state_times {
      Φ = Φ ∧ encodeStateDenialAxiom(S, t_in, cond)
    }
    return Φ
  }
}
```

Algorithm 6 also verifies that the state condition formula of each monitored state S is true when the system is in S. It first obtains the timesteps at which the system is in S. Then it generates state denial axioms (Axiom (5)) for those timesteps that appear in the log. Because it uses log file preprocessing, the algorithm doesn’t need to generate explanation closure axioms. Algorithm 6 scales well to the size of the statechart (as Section 6.2 will illustrate).

Algorithm 7 returns a single diagnosis for the system. The diagnosis specifies whether or not each transition and state in the statechart is denied. The diagnosis pinpoints denied transitions and states if failures have occurred. And if not, if the system is running correctly, it indicates that no transitions and states are denied. The algorithm calls Algorithm 6 to encode formula Φ using log preprocessing. If Φ is satisfiable, Algorithm
Algorithm 7 Find diagnosis

\[
\text{diagnose}() \{
\Phi = encode_{\Phi \_with \_log \_preprocessing}(\text{statechart, log});
\text{solver.solve}(\Phi);
\text{if (\Phi is satisfiable) } \{
\mu = \text{SAT result};
\text{//map SAT result to diagnostic instance}
\text{diagnosticInstances = decode(}\mu\text{)}
\text{//filter the diagnostic instances to contain only}
\text{//satisfactions and-denials of actions and states}
\text{diagnosis = filter(diagnosticInstances) } \}
\text{return diagnosis}
\]

7 decodes the solver result \(\mu\) into diagnostic instances. It then filters the diagnostic instances into a diagnosis containing \(FD\) and \(\neg FD\) predicates over all transitions and states in the statechart. We demonstrate in Section 6.2 both that Algorithm 7 returns correct diagnoses, and that it scales to the size of the statechart.

5.5 Evaluation

This section evaluates the performance of our framework. We used the ATM simulation case study [Bjo] as a running example to illustrate how our framework works. We then performed 2 sets of experiments, containing 20 and 10 experiments respectively, on randomly generated and progressively larger statecharts. Our results show that our solution scales to the size of the statechart, and can therefore be applied to industrial software applications. All experiments were performed on a machine with a Pentium 4 CPU with 1 GB of RAM.
5.5.1 The ATM Running Example

The ATM log data (from Section 5.3.1) contains two errors: ¬session\_created(5) and serving\_customer(7). The first error indicated that action a3:create\_session’s effect (session\_created) was false at timestep 5 (after the action was executed at timestep 4). The second error indicates that state Idle’s condition formula (atm\_on ∧ ¬serving\_customer) was false at timestep 7 (while the system was in Idle at 7).

The diagnostic component calls Algorithm 6 (Section 5.4.7) to preprocess the log data and encode the statechart, log, and denial axioms into a SAT propositional formula Φ. The diagnostic component then follows Algorithm 7 to infer that action create\_session and state Idle were denied at timesteps 5 and 7 respectively. The diagnostic component returns a diagnosis that contains FD(create\_session, 5) and FD(Idle, 7). For readability, we don’t list ¬FD predicates in the diagnosis.

5.5.2 Performance Evaluation

We performed two sets of experiments to evaluate the scalability of our framework with respect to statechart size. The first contains 20 experiments on 20 progressively larger statecharts. The smallest statechart contains 50 states and 100 transitions, and the largest statechart contains 1000 states and 2000 transitions. We demonstrate the scalability of our framework when the numbers of states and transitions increase at the same proportional rate (maintaining a transition to state ratio of 2:1). The second set of experiments contains 10 experiments on 10 progressively larger statecharts. These statecharts all contain the same number of states (1000 states), but the number of transitions increases from 1000 to 10,000. We demonstrate the scalability of our framework when the number of states remains constant and the number of transitions increases quickly.

Our program generated all these statecharts randomly. For each generated state, our program also randomly generates: (1) its state condition formula containing up to 2
clauses, (2) 0 to 1 entry action, and (3) 0 to 1 exit action. For each generated transition, our program randomly selects its source and destination states, then randomly generates for the transition: (1) 0 to 2 triggering events, (2) 0 to 2 guarding conditions, (3) 0 to 1 domain expert added precondition formula, and (4) 0 to 1 domain expert added effect formula.

Table 5.4 reports the results of the first set of experiments. All states and transitions were monitored in each of these experiments. We injected an error into a randomly chosen transition action $a_i$, with the goal of finding a diagnosis that contains $FD(a_i, t)$ for some timestep $t$. Columns 1 and 2 in Table 5.4 ($#S$ and $#T$) list the number of states and transitions in the statechart. Columns 3 and 4 ($#L$ and $#C$) give the total numbers of literals and clauses in the propositional formula, $\Phi$, encoded for the SAT solver, using log preprocessing (Algorithm 6). Column 5, $T_{en}$, lists the time taken (in seconds) for the diagnostic component to encode $\Phi$ using log preprocessing. Column 6, $T_{read}$, gives the time the SAT solver took (in seconds) to read $\Phi$ into its internal memory. Column 7, $T_{diag}$, gives the time (in seconds) taken by the SAT solver to solve $\Phi$ ($T_{solve}$) and the time taken by our diagnostic component to decode $\Phi$ into a diagnosis ($T_{decode}$) (Equation 5.9). The last column, $T_{sum}$, represents the total time taken (in seconds) for diagnostic reasoning, calculated as the sum of the time values listed in Columns 5, 6, and 7 (Equation 5.10).

$$T_{diag} = T_{solve} + T_{decode} \quad (5.9)$$

$$T_{sum} = T_{en} + T_{read} + T_{diag} \quad (5.10)$$

Figure 5.5 depicts the relationship between the total time taken for diagnostic reasoning (the y-axis - the values in columns 5, 6, 7, and 8 of Table 5.4) and the numbers of states and transitions in the statechart (the x-axis - the values of columns 1 and 2 of Table 5.4). The four curves in Figure 5.5 show that the diagnostic component scales to the size of the statechart. Interestingly, the biggest time component is the time taken
by the SAT solver to read in and parse the propositional formula \( \Phi \) \( (T_{\text{read}}) \). Because we treat our SAT solver (SAT4J [LB05]) as a black box, we don’t consider performance optimizations of the solver itself.

We performed the second set of experiments to run an additional check on the per-
performance of our diagnostic component. All statecharts contain a fixed number of states (1000). They contain from 1000 to 10,000 transitions. Table 5.5 reports our results. The first column lists the numbers of transitions. Columns 2 and 3 list numbers of generated literals and clauses. Columns 4 ($T_{en}$), 5 ($T_{read}$) and 6 ($T_{diag}$) represents respectively the
time taken (in seconds) to encode $\Phi$, read $\Phi$, solve $\Phi$ and decode the SAT result into a diagnosis. The last column, $T_{\text{sum}}$, gives the total time (in seconds). Figure 5.6 gives the relationship between the total time taken for diagnostic reasoning (the values in columns 4, 5, 6, and 7 of Table 5.5) and the numbers of transitions in the statechart (Column 1 of Table 5.5).

These results demonstrate that when encoding is done with log file preprocessing, it is feasible for our approach to scale well with larger industrial software applications.

5.6 Discussions

This chapter presents our work on monitoring and diagnosis using statecharts. We chose to extend our diagnostic framework presented in Chapter 4 with statecharts because the Unified Modeling Language (UML) (including statecharts) has become industry standard for modeling software applications. By extending our framework presented in Chapter 4 with statecharts, we can apply our approach to a wider range of systems. This section discusses the main assumptions, limitations, and contributions of our work on monitoring and diagnosis using statecharts.
We base our approach on several important assumptions. Firstly, we assume that a statechart is available and it is correct. Domain experts need to prepare a system statechart that specifies the intended/correct system behaviors. Our work deals with a subset of the statechart semantics - we verify that a system's run time behavior complies with a subset of the semantics described in its statechart. Hence, our work assumes and depends on the correctness of the statechart itself. This assumption includes assuming any domain expert added preconditions and effects to transitions are also correct.

Secondly, we assume that traceability links are available and they link between a system’s source code and statechart. The instrumentation component needs this traceability information to know where in the source code to insert monitors so that information regarding monitored states and transitions can be collected. These traceability links need also be provided by domain experts. One way to alleviate the difficulties of providing a system statechart and traceability links is to prepare them at a high abstraction level. A system can be modeled at a high abstraction level with few states and transitions. When this is the case, traceability links link between high level statechart elements to larger scaled system sub-components rather than to individual methods and functions.

Thirdly, we assume that it is possible for the instrumentation component to generate all the log data needed for diagnosis. In [Zho08], Zhou provided a first implementation of the instrumentation component that is based on requirement goal models. This component needs to be extended to work with statecharts. We also assume the diagnosability of monitored systems.

Our work has several limitations. Most importantly, it deals with a (rather small) subset of the statechart semantics. Specifically, the subset that we deal with includes transition firing, and changes in the system due to transition firing. Our work currently does not deal with more complex concepts such as concurrency, history, conflicts, priority, nondeterminism, hierarchy, or temporal relationships. For example, our work does not keep track of the number of times a certain action is repeated (lack of notions of history
and time). Therefore cannot verify patterns of events such as “action X must be repeated \( n \) times within 5 seconds of each other and before action Y takes place”. Concurrency is an important aspect of statechart. Our work monitors the successful executions of each concurrent transition, but we can not verify that these concurrent transitions did occur concurrently. Our work verifies a system’s runtime behavior complies with a subset of its statechart specification. Meaning that it cannot verify a system is correct against its complete specification. Much future work is required to achieve this.

Our evaluation uses randomly generated statecharts, instead of statecharts from real world applications. We randomly generated statecharts of increasing size to test scalability. The largest statechart we experimented on contained 1000 states and 10,000 transitions. It would have been difficult to obtain real world applications with statecharts of this scale. Therefore we chose to generate these statecharts to test the feasibility of scaling our solution to large specifications. More studies are needed to test the feasibility of applying our approach on real life applications, especially large scale applications.
Chapter 6

Reconfiguration and Execution

6.1 Reconfiguration Algorithms

High variability software systems deliver their functionalities in multiple ways by reconfiguring their components. If one of these ways fails, the system can switch to an alternative. Consider an Automated Teller Machine (ATM) system that may either print or display a receipt at the end of a transaction. If the ATM’s printer malfunctions, the ATM may display a receipt instead. High variability has become an important concern for software designers. It underlies the notions of product family, software personalization, and service-oriented architecture. Our proposed framework exploits high variability in a software system to deliver self-repairing capabilities through reconfiguration. The idea of using high variability to support autonomicity was sketched in [YLL+08].

In this section, we present reconfiguration algorithms for computing a set of best system reconfigurations for repair. We define a configuration as follows:

**Definition 9 (Configuration)** A configuration consists of a set of tasks from a goal model which, when executed successfully in some order, lead to the satisfaction of the root goal (in the goal model).

The example goal model in Figure 6.1 contains the following five configurations:
Chapter 6. Reconfiguration and Execution

Figure 6.1: Example Goal Model

Configuration 1: \{a1, a3, a4, a5, a8, a9, a10, a11\}
Configuration 2: \{a1, a3, a4, a6, a8, a9, a10, a11\}
Configuration 3: \{a2, a3, a4, a5, a8, a9, a10, a11\}
Configuration 4: \{a2, a3, a4, a6, a8, a9, a10, a11\}
Configuration 5: \{a7, a8, a9, a10, a11\}

This section discusses the three main reconfiguration algorithms (Algorithms 8, 9, and 11) that find best reconfiguration(s) free of failures. Algorithm 8 finds best global reconfiguration(s) by searching the entire goal model. Algorithm 9 finds best local reconfiguration(s) by searching only locally on a sub-graph of the goal model. In worst cases, the performance of both Algorithms 8 and 9 is exponential to the size of the goal model they search on. Algorithm 11 is a greedy algorithm that a best global reconfiguration. It computes sub-configuration for each node in the model once, then add these sub-configurations to the final reconfiguration. The number of sub-configurations computed is linear to the size of the goal model.

One or multiple best global reconfigurations may exist. If there is one best reconfig-
uration, both Algorithms 8 and 11 return it. If there are multiple, Algorithm 8 returns them all, and Algorithm 11 returns one of them. All three algorithms assume that tasks in a goal model are independent from each other. For example, in the goal model shown in Figure 6.1, we assume that the decision of selecting task $a_1$ vs. $a_2$ in a configuration $c_i$ is independent of the decision of selecting $a_5$ vs $a_6$ for $c_i$. This is to say that we ignore any contribution links between hard goals and tasks. Any approach that takes into account these contribution links would exhibit exponential behavior - there is no easy solution.

Both Algorithms 8 and 9 first compute a set of system reconfigurations free of failures. If multiple reconfigurations exist, both algorithms choose one or more best reconfigurations that contribute most positively to system’s soft goals, and that reconfigure the system the least from its current configuration. The difference between the two algorithms lies within the first step where the algorithms find a set of reconfigurations free of failures. Algorithm 8 searches globally on the entire goal model and finds all system configurations that are free of failures. Obviously, the number of global configurations increases exponentially with the size of the goal model. Algorithm 9 alleviates this problem by searching only locally on a goal model and finding only the reconfigurations to goals and tasks that had failed. Its performance is improved, but the number of local reconfigurations can still be exponential to the size of the sub-goal model it searches on. Algorithm 11 does not compute a set of configurations free of failures, then chose therefrom a best. Instead, it calculates contribution scores for individual goals and tasks. Whenever there is a choice (an OR decomposition), it chooses goals/tasks that contributes more positively to soft goals. We describe our three algorithms in this section. We evaluate their performance in Section 6.2.2.

Algorithm 8 takes as input a goal model and failed goals/tasks. It calculates *globally* all reconfigurations that do not include any failed goals/tasks. If more than one such reconfiguration exists, it applies our two selection criteria to compute the best reconfiguration(s). The first criterion selects configurations that contribute most positively to
Chapter 6. Reconfiguration and Execution

Algorithm 8 Find Best Global Reconfigurations

\textbf{findGlobalReconfig} \,(goal\_model, failures) \{

\hspace{1em} //find all global reconfigurations

allReconfigs = all reconfigures that are free of failures

\hspace{1em} //apply the first selection criterion

if (goal\_model contains softgoals) \{

\hspace{2em} for each reconfig in allReconfigs

\hspace{3em} \textbf{score} = \textbf{calConfigScore}(goal\_model, config)

\hspace{3em} bestConfigs = reconfigs with the highest score

\}

\hspace{1em} //apply the second selection criterion

if (multiple reconfigs tie on their scores)

\hspace{2em} bestReconfigs = configs that minimally reconfigures

\hspace{2em} the system from its current configuration

\hspace{2em} return bestReconfigs

\}

soft goals. For each reconfiguration, the algorithm calls Algorithm 10 (\textit{calConfigScore})
to calculate a score that indicates how positively the reconfiguration contributes to soft
goals. The algorithm chooses the reconfiguration(s) with the highest scores. If multiple
“best reconfigurations” tie, it uses the second selection criterion to further prune the list.
Criterion two selects configurations that minimally reconfigure the system by choosing
configurations that reuse the greatest number of tasks from system’s current configura-
tion. Algorithm 8 searches for best global reconfigurations. It may replace goals/tasks
that did not fail with alternatives that contribute more positively to soft goals. We show
in Section 6.2.2 that the Algorithm 8 scales poorly to goal model size.

Algorithm 9 finds local reconfigurations that replace only denied goals and tasks
with their alternatives, and selects therefrom the best reconfigurations. The algorithm is
Algorithm 9 Find Best Local Reconfigurations

findLocalReconfig (goal_model, failures) {
    //find all local reconfigurations that are free of failures
    for each denied goal and task {
        reconfigableGoal = the closest OR-decomposed
                        ancestor goal to the denied goal/task
        localReconfigs = configs contained under
                        reconfigableGoal that are free of failures
    }
    allLocalReconfigs = configs in all localReconfigs
    //apply the first selection criterion
    if (goal_model contains softgoals) {
        for each reconfig in allReconfigs
            score = calConfigScore(goal_model, config)
            bestConfigs = reconfigs with the highest score
    }
    //apply the second selection criterion
    if (multiple reconfigs tie on their scores)
        bestReconfigs = configs that minimally reconfigures
                        the system from its current configuration
    return bestReconfigs
}

proposed to alleviate the poor scalability suffered by Algorithm 8. However it can still
exhibit exponential behavior depending on the size and structure of the sub-goal graph it
searches on. For each denied goal/task, the algorithm finds its immediate, or closest, OR-
decomposed ancestor goal. Local reconfigurations free of failures are contained under this
ancestor goal. Take for example the goal model in Figure 6.1. Goal \( g_2 \) is the closest OR-
decomposed ancestor to task $a_3$. If $a_3$ fails, the algorithm searches for reconfigurations under the decomposition of $g_2$ that are free of failures. Task $a_7$ is chosen as the only local reconfiguration to $a_3$. If no such OR decomposed ancestor goals exist within the goal model, no reconfigurations exist. If more than one local reconfiguration is found, the algorithm selects the best reconfiguration(s) using the same criteria as Algorithm 8. Algorithm 9 only finds local best reconfigurations that replace denied goals/tasks. The number of such reconfigurations depends on where in the goal model the failures have occurred, together with the structure of the goal model. In worst case, this number is exponential to the size of the partial goal model it searches on.

\[
\begin{align*}
    hG \overset{++S}{\longrightarrow} sG : \neg FD(hG, s) & \rightarrow FS(sG, s) \quad (6.1) \\
    hG \overset{--S}{\longrightarrow} sG : \neg FD(hG, s) & \rightarrow FD(sG, s) \quad (6.2) \\
    hG \overset{+S}{\longrightarrow} sG : \neg FD(hG, s) & \rightarrow PS(sG, s) \quad (6.3) \\
    hG \overset{-S}{\longrightarrow} sG : \neg FD(hG, s) & \rightarrow PD(sG, s) \quad (6.4) \\
    FS(sG, s) & \rightarrow PS(sG, s) \quad (6.5) \\
    FD(sG, s) & \rightarrow PD(sG, s) \quad (6.6)
\end{align*}
\]

To calculate each configuration’s contribution score to soft goals, we first propagate satisfaction labels from hard goals and tasks in the given configuration to soft goals in a goal model. In other words, we want to compute satisfaction labels for soft goals in a goal model assuming that all hard goals and tasks in a given configuration are satisfied. We do not consider contribution links between hard goals and tasks. Axioms 6.1 to 6.4 describe label propagations from hard goals and tasks ($hG$) to soft goals ($sG$) of a goal model. These axioms specify that if link $++S$ (or $--S$, $+S$, $-S$) proceeds from $hG$ to $SG$, and if $hG$ is fully satisfied in execution session $s$, then $sG$ is fully satisfied (or fully denied, partially satisfied, partially denied respectively) in $s$. The $++D$, $--D$, $+D$, $-D$ cases are dual w.r.t. $++S$, $--S$, $+S$, $-S$ respectively. Axioms 6.5 to 6.6 specify that for
any soft goal $sG$, if it is fully satisfied (or denied) in $s$, it is also partially satisfied (or
denied respectively) in $s$.

Algorithm 10 Calculate a Reconfiguration’s Contribution to Soft Goals

```python
calConfigScore (goal_model, config) {
    FSValue=1, PSValue=0.5, FDValue=-1, PDValue=-0.5
    score = 0
    propagate satisfaction labels following axioms 6.1 to 6.6
    for each softgoal $s_i${
        priority = $s_i$’s priority value
        for each of $s_i$’s satisfaction label {
            if (label == FS)
                score += priority $\times$ FSValue
            else if (label == PS)
                score += priority $\times$ PSValue
            else if (label == FD)
                score += priority $\times$ FDValue
            else if (label == PD)
                score += priority $\times$ PDValue
        }
    }
    return score
}
```

Algorithm 10 calculates a contribution score for a given configuration. A configura-
tion’s total contribution to all the soft goals is the sum of its contributions to each soft
goal. The algorithm propagates satisfaction labels from hard goals/tasks in the given
configuration to soft goals by following Axioms 6.1 to 6.6. A soft goal has one to four
satisfaction labels: $FS$, $FD$, $PS$, and $PD$. There may be both full and partial evidence
that a soft goal is both satisfied and denied in the same execution session $s$. Therefore, a
soft goal can have one or more or even all of the four satisfaction labels associated to it in
We assign numeric values: 1, 0.5, -1, and -0.5 to $FS$, $PS$, $FD$, and $PD$ respectively. In addition, a soft goal is assigned one of three priority values: high, medium or low. Numeric values of 3, 2, and 1 are used to represent them respectively.

For each such goal $s_i$, the algorithm obtains its priority value, as well as all of its satisfaction labels. For each of $s_i$’s satisfaction label, the algorithm increases the configuration’s score by the product of $s_i$’s priority and the label’s numerical value. For example, consider a soft goal $sG$ with high priority and satisfaction labels $FS$ and $PD$. $sG$’s score is calculated as follows: $FS\text{Value} \times \text{priority} + PD\text{Value} \times \text{priority}$, which is $1 \times 3 + (-0.5) \times 3 = 1.5$. Note that soft goals with satisfaction labels of $FS$ and $FD$ only or $PS$ and $PD$ only have satisfaction scores of 0, because the evidences of their satisfaction and denial cancel each other out.

Algorithm 11 calls a recursive algorithm (findBestSubConfig) to greedily compute a best global configuration. The algorithm is effective when all contribution links are to soft goals. In this case, a sub configuration can be chosen for any part of the model below some given node without looking at other parts because of the linearity of the scoring algorithm, which is a summation over all chosen goals and tasks. The impact of making a decision for the part of a goal model below a certain goal is not impacted by and does not impact decisions made in other parts of the goal model, and thus can be made greedily.

Algorithm findBestSubConfig contains two steps. In the first step, it recursively traverses down the goal model one level from a given goal. Traversal stops when leaf nodes are reached. A data structure (subConfig) is created for each node (goal/task) traversed. SubConfig for a node contains information on whether the node has failed, the best partial configuration under the node, and its contribution score to soft goals. This contribution score is the sum of the contribution score of the node itself and the scores calculated for all of its children selected to participate in the configuration. For a leaf node, the structure is filled when traversal reaches that node. The node is checked
Algorithm 11 Greedy Algorithm to Find a Best Global Reconfiguration

```java
findGreedyGlobalReconfig (goal_model, failures) {
    config=findBestSubConfig(goal_model.root_goal, failures)
}

class SubConfig {
    score, failed, node, subelements
    SubConfig(leaf,failures) {
        node=leaf
        failed=is leaf in failures
        score=computePartialScore(leaf))
    }

SubConfig findBestSubConfig (g, failures) {
    configOfG = new SubConfig()
    subconfigs=new SubConfig[g.children.size()]
    for (i=0; i<children.size(); i++) {
        if (child[i] is not leaf node) subconfigs[i]=findBestSubConfig(child[i],failures)
        else subconfigs[i]=new SubConfig(child[i],failures)
    }
    if g is and-decomposed
        configOfG.subelements=subconfigs
        if (any subconfigs are failed) configOfG.failed=true
    else
        find subconfig with highest score that is not failed
        if found then configOfG.subelements=that subconfig
        else configOfG.failed=true
    configOfG.node=g
    configOfG.score=computePartialScore(g)+\sum_{i} configOfG.subelements(i).score
    return configOfG
}
```
against the list of failed tasks and the score is computed based on only that leaf node. For non-leaf nodes, the structure is filled in the second step.

In the second step, the algorithm computes the best configuration given all of the children’s configurations. A (partial) configuration is calculated for each goal $g$. This partial configuration contains the best configuration that fulfills $g$. For an AND-decomposed goal $g$, the best and only configuration is to add all of the partial configurations calculated for $g$’s children. If any of $g$’s children’s configuration contains a failure, then $g$’s configuration fails (boolean flag failed is marked true for the configuration). For an OR-decomposed goal, the best configuration is the configuration of one of its children such that the child’s configuration does not contain a failure and has the highest score\(^1\).

The contribution score of a goal $g$ is calculated as the sum of $g$’s own contribution score and the scores of all $g$’s children that are chosen to participate in the configuration. We show in Section 6.2.2 that Algorithm 11 scales well to the size of a goal model.

### 6.1.1 Reconfiguring the Running Example

We illustrate our reconfiguration algorithms by applying them on the running example given in Figure 6.1. The goal model in the running example contains five possible configurations. The successful execution of all the tasks in each of the five configurations leads to the satisfaction of the root goal $g_1$. Again, these 5 configurations are:

- **Configuration 1**: $[a1, a3, a4, a5, a8, a9, a10, a11]$
- **Configuration 2**: $[a1, a3, a4, a6, a8, a9, a10, a11]$
- **Configuration 3**: $[a2, a3, a4, a5, a8, a9, a10, a11]$
- **Configuration 4**: $[a2, a3, a4, a6, a8, a9, a10, a11]$
- **Configuration 5**: $[a7, a8, a9, a10, a11]$

Consider if Configuration 1 is executed, and if $a1$ fails, Algorithm 9 finds one local

---

\(^1\) The first best configuration is chosen, although a different approach to tie breaks could be used.
Table 6.1: Finding a Best Global Reconfiguration

<table>
<thead>
<tr>
<th>Soft Goal Priority</th>
<th>Contribution Scores</th>
<th>Best Config.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sG_1=\text{medium}$, $sG_2=\text{medium}$</td>
<td>config3=1, config4=-2, config5=0</td>
<td>config3</td>
</tr>
<tr>
<td>$sG_1=\text{high}$, $sG_2=\text{low}$</td>
<td>config3=0, config4=-1, config5=2</td>
<td>config5</td>
</tr>
</tbody>
</table>

reconfiguration (Configuration 3) that replaces only the failed task $a_1$ with its alternative task $a_2$ (i.e. did not replace any task that did not fail). Since there is only one local reconfiguration, no further selection is necessary. Configuration 3 is returned as the best local reconfiguration.

Algorithm 8 is called to find a best global reconfiguration. The algorithm first finds three global reconfigurations that do not include $a_1$ (Configurations 3, 4, and 5). It then needs to choose a best configuration among the three. We illustrate the selection process through two experiments. Their experimental results are reported in Table 6.1. These two experiments assign different priority values to the soft goals, $sG_1$ and $sG_2$, in the goal model. In the first experiment (row 1 of Table 6.1), both $sG_1$ and $sG_2$ have medium priority. In the second experiment (row 2), $sG_1$’s priority is high, while $sG_2$’s priority is low. The table’s second column lists each global reconfiguration’s contribution score to soft goals. The last column lists the chosen configuration. In the first experiment, the scores for configurations 3, 4, and 5 are 1, -2, and 0 respectively. Configuration 3 is therefore chosen. In the second experiment, the contribution scores changed to 0, -1, and 2 respectively, making configuration 5 the best.

We now show how the scores of 0 and 2 were derived for configuration 5. Configuration 5 involves executions of task $a_7$ and goal $g_7$. One MAKE (shorthand for $++S$ and $++D$) and one BREAK (shorthand for $--S$ and $--D$) contribution link proceed from $a_7$ to soft goals $sG_1$ and $sG_2$, respectively. If $a_7$ is fully satisfied in execution session $s$, the labels for the soft goals are $FS(sG_1,s)$ and $FD(sG_2,s)$ after the label propagation process. Configuration 5’s contribution score is calculated as: $FS\text{Value} \times \text{priority}(sG_1)$
+ \( FDValue \times priority(sG2) \). In the first experiment, the score comes to \( 1 \times 2 + (-1) \times 2 = 0 \), as \( sG1 \) and \( sG2 \)'s priority values are 2. In the second experiment, the score comes to \( 1 \times 3 + (-1) \times 1 = 2 \), as \( sG1 \) and \( sG2 \)'s priority values are 3, and 1 respectively.

These two experiments show us that the priority values specify which soft goals are more important. These values can guide the reconfiguration algorithms to choose a set of best configurations that contribute most positively to the most important soft goals. Note that if any task under the decomposition of a maximal AND-subtree fails, there are no reconfigurations that can repair its failure. In this example, these “non-repairable” tasks are: \( a3, a4, a8, a9, a10, \) and \( a11 \).

For readability, we illustrate Algorithm 11 on the partial goal model starting at \( g3 \) in the goal model in Figure 6.1. Assume soft goals \( SG1 \) and \( SG2 \) both have a low priority value (numeric priority value is 1). Task \( a1 \) failed. The following shows the important recursive calls and calculations for finding a best configuration for the partial goal model rooted by \( g3 \):

```plaintext
findBestSubConfig(g3,\{a1\})
subconfigs[0]=findBestSubConfig(g4,\{a1\}){
    subconfigs[0]=new SubConfig(a1,\{a1\})
    subconfigs[1]=new SubConfig(a2,\{a1\})
    choose best unfailed one
    configOfG.subelements={subconfigs[1]}
    configOfG.score=0 + 0
}
subconfigs[1]=findBestSubConfig(g5,\{a1\}) {
    subconfigs[0]=new SubConfig(a3,\{a1\})
    subconfigs[1]=new SubConfig(a4,\{a1\})
    choose all
    configOfG.subelements={subconfigs[0],subconfigs[1]}
```
Figure 6.2: Data Structures Created When Calculating a Best Configuration for G3 in Figure 6.1

\[
\text{configOfG.score= 0 + 0 + 0}
\]

\[
\text{subconfigs[2]=findBestSubConfig(g6,{a1})}
\]

\[
\text{subconfigs[0]=new SubConfig(a5,{a1})}
\]

\[
\text{subconfigs[1]=new SubConfig(a6,{a1})}
\]

choose best unfailed one

\[
\text{configOfG.subelements={subconfigs[0]}}
\]

\[
\text{configOfG.score=0 + 1}
\]

choose all

\[
\text{configOfG.subelements={subconfigs[0],subconfigs[1],subconfigs[2]}}
\]

\[
\text{configOfG.node=g3 configOfG.score= -0.5 + 0 + 0 + 1 = 0.5}
\]

Chosen configuration = \{a2,a3,a4,a5\}
We call algorithm \textit{findBestSubConfig} on the root goal \(g3\) of the partial goal model. Figure 6.2 shows the important data structures (SubConfig) created along this process. Because \(g3\) is not a leaf node, the algorithm traverses to its three children: \(g4\), \(g5\), and \(g6\). Therefore, \(g3\)’s subconfigs array contains the best configurations calculated for \(g4\), \(g5\), and \(g6\). \(G4\) has two leaf nodes \(a1\) and \(a2\). We create a data structure \textit{SubConfig} for each of them. Since \(g4\) is OR-decomposed, we chose the best of its children that did not fail. \(A2\) is chosen because it is the only child of \(g4\) that did not fail. The configuration calculated for \(g4\) therefore contains \(a2\) (saved in \textit{configOfG.subelements}). There are no contribution links between \(g4\) and \(a2\) to soft goals. Therefore the contribution scores for \(g4\) and \(a4\) are both 0 (0 * 1 = 0, where 1 is the priority value). The partial configuration for \(g4\) containing \(a2\) is \(0 + 0 = 0\).

The calculation for a best partial configuration for \(g5\) is as follows. \(G5\) is AND-decomposed to leaf nodes \(a3\) and \(a4\). The best configuration for \(g5\) is to select both of its children in the configuration (both of the configurations calculated for its children, which include only themselves). Therefore, the configuration calculated for \(g5\) contains \(a3\), and \(a4\). Its contribution score is 0 because no contribution links between these nodes and soft goals.

Finally, we calculate the best configuration for \(g6\). \(G6\) is OR-decomposed to \(a5\) and \(a6\). We choose a best un-failed one. A MAKE link proceeds from \(a5\) to soft goal \(sG2\), meaning that if \(a5\) is fully satisfied, so is \(sG2\). Therefore the contribution score of \(a5\) is \(1 * 1 = 1\) (numeric value for \(FS\) is 1, and priority level is 1). A HURT link processes from \(a6\) to \(sG2\), meaning that if \(a5\) is fully satisfied, \(sG2\) is fully denied. The contribution score of \(a6\) is calculated as \(-1 * 1 = -1\) (numeric value for \(FD\) is -1). The algorithm chooses to add task \(a5\) in the configuration because it has a higher score. The partial configuration for \(g6\) therefore contains \(a5\). \(G6\)’s contribution score is 0. The configuration’s contribution score is \(0 + 1 = 1\).

Now we have find the best configurations for \(g3\)’s children, the recursive function
returns, and it is time to find a best configuration for $g_3$. $G_3$ is AND-decomposed, its best and only configuration is to add the best configurations calculated for all of its children. Again, configurations calculated for $g_4$, $g_5$, and $g_6$ are: $[a_2]$, $[a_3, a_4]$, and $[a_5]$. The final configuration for $g_3$ is $[a_2, a_3, a_4, a_5]$. A HURT link proceeds from $g_3$ to soft goal $sG_1$, meaning if $g_3$ is satisfied, $sG_1$ is partially denied. Therefore, $g_3$’s contribution score is $-0.5 \times 1 = -0.5$ (numeric value for $PD$ is -0.5). The scores of $g_4$, $g_5$, and $g_6$’ configurations are 0, 0, and 1 respectively. Therefore, the score for $g_3$’s final configuration is calculated as $-0.5 + 0 + 0 + 1 = 0.5$.

### 6.2 Evaluation

In Chapters 4 and 5, we evaluated the correctness and performance of our monitoring and diagnostic components. In this section, we evaluate our reconfiguration component. We illustrate Algorithms 8 and 9 by applying them to an ATM (Automated Teller Machine) simulation software [Bjo] and show that they transform the original ATM into an adaptive system with self-reconfiguration capabilities. We then evaluate and compare the performance of Algorithms 8, 9, and 11 by applying them to randomly generated goal models of increasing size (the largest goal model contains 1000 goals and tasks). The experimental results show the feasibility of scaling our approach to industrial software applications with medium-sized goal graphs. All experiments reported were performed on a machine with an AMD Sempron LE-1200 CPU with 1GB of RAM.

#### 6.2.1 ATM Case Study

The ATM simulation case study is an illustration of OO design used in a software development class at Gordon College [Bjo]. The application simulates an ATM performing customers’ withdraw, deposit, transfer and balance inquiry transactions. Its original source code contains 36 Java Classes with 5000 LOC. In this section, we illustrate recon-
We also show that our framework transforms the original ATM software into an adaptive system with self-reconfiguration capabilities.

The implementation of the original ATM was extended to include alternative behaviors for the purpose of making reconfiguration possible. Figure 6.3 shows a partial goal graph of the extended ATM, with 19 goals and 26 tasks. The priority levels for all soft goals are set to medium. Four major alternative behaviors were added: (1) the original ATM asks the operator to manually enter ATM cash amount (represented by task \(a_3\)); the extended ATM offers a cash sensor that automatically senses the cash amount (represented by task \(a_2\)). (2) The original ATM asks customers to insert their bank cards (\(a_7\)); the extended ATM allows a customer to manually enter her card number if she does not have the physical card with her (\(g_7\)). (3) The original ATM takes customers’ input from a full-keyed physical keypad (\(a_5\) and \(a_9\)); the extended ATM adds a backup two-key keypad so that it can be used if the full-keyed keypad fails (\(a_6\) and \(a_{10}\)). (4) The original ATM prints customers’ receipts (\(a_{21}\)); the extended ATM displays the receipt if its printer fails (\(a_{22}\)).
The framework first calls Algorithm 1 (inside the monitoring component) to compute the optimal monitoring granularity. Algorithm 1 selects goals $g_{13}$ and $g_{19}$, which are roots of maximal AND-subtrees, for monitoring. Tasks not under $g_{13}$ and $g_{19}$ are also monitored. The instrumentation component instruments the ATM at the appropriate places using AspectJ technologies [Zho08]. At run time, the ATM produces log data that are analyzed by the diagnostic component to infer requirement denials. Consider a scenario where a configuration containing goals/tasks $[a1, a2, a4, a7, a8, a9, a11, g_{13}, a_{21}, a_{23}, g_{19}]$ is executed. An error was injected into the execution of task $a7$. The diagnostic component returns a single diagnosis containing $FD(a7, s)$, indicating that $a7$
is fully denied in session $s$.

The framework calls Algorithm 8 to find all global reconfigurations. Table 6.2 reports the results. Algorithm 8 returns a total of 16 global reconfigurations that do not include the failed task. The first, second, and last columns of the table list respectively the reconfiguration number, the actual reconfiguration, and its contribution score. The listed reconfigurations contain both goals and tasks (instead of all tasks) for readability. Reconfigurations 1, 3, 9, and 11 tie on their contributions to soft goals, with scores of 1. The algorithm then applies the second selection criterion to choose a best reconfiguration. This second criterion seeks the reconfiguration that minimally reconfigures the system from its current configuration. The first configuration is selected because it replaces failed task $a_7$ with its alternative task $a_5$ and reuses all other goals/tasks from the current configuration.

Under the same failure scenario, Algorithm 9 finds 2 local reconfigurations: reconfigurations 1 and 2 that replace $a_7$ with $a_5$ and $a_6$ respectively. Reconfigurations 1 and 2 have a contribution score of 1 and 0 respectively. As a result, reconfiguration 1 wins out as the best alternative.

In this example, Algorithms 8 and 9 returned the same best reconfiguration because the best global and local reconfiguration happens to be the same. In its next execution, the ATM runs with a new configuration free of failed components.

### 6.2.2 Performance Evaluation

To evaluate the scalability of the reconfiguration component, we performed 20 sets of experiments on 20 progressively larger goal models containing from 50 to 1000 goals and tasks. Each of the 20 sets of experiments contains 10 runs on goal models of equal size. Our program generated these goal models randomly. For each node in the goal model, our program randomly decides: (1) whether it is a goal or a task, (2) whether it is AND- or OR- decomposed, and (3) which goal is its parent. Also, a number of soft goals and
contribution links are generated. All contribution links proceed from hard goals/tasks to soft goals. One task is chosen randomly to fail.

These experiments show that the reconfiguration component scales to the size of the goal model when it uses the greedy algorithm (Algorithm 11) or when it searches for the best local (instead of global) reconfigurations (Algorithm 9).

We performed these experiments using reconfiguration Algorithms 8, 9, and 11 to compare their efficiency. Figure 6.4 report experimental results. The x-axis represents the size of the goal model. The smallest goal model contains 50 goals/tasks, and the largest goal model contains 1000 goals/tasks. The y-axis shows the time to find reconfigurations in milliseconds. We show 4 curves in Figure 6.4: (1) the geometric-mean time taken to find best global reconfigurations (Algorithm 8), (2) the max time taken (worst case performance) to find best local reconfigurations (Algorithm 9), (3) the geometric-mean time taken to find best local reconfigurations (Algorithm 9), and (4) the geometric-mean time taken by the greedy algorithm to find best global reconfigurations (Algorithm 11). Each of the data point in curves (1), (3), and (4) is the geometric-mean of 10 experiments.

As can be seen from the graph, Algorithm 8 has the worst time performance. The number of best global reconfigurations is exponential to the size of the goal model and the size of the biggest OR-decomposed subtree of the goal model. Algorithm 8 ran out of memory on a goal model containing 300 goals and tasks. The mean/average time taken by Algorithm 9 to find best local reconfigurations grew reasonable well as the size of the goal model increases. However, in worst cases, Algorithm 9 is exponential to the subgraph it searches on. We show the max amount of time taken by Algorithm 9 to find local reconfigurations for all goal model sizes. As can been seen from the graph, Algorithm 9 is more sensitive to the size and structure of the partial goal model it computes on, than to the entire goal model size. Its performance depends on where failures have occurred and the structure of the model. Algorithm 11 produced the best data: it not only finds a best global reconfiguration, but also its performance is steady and scaled well to the
Because Algorithm 11 does not find all exhaustive combinations of goals/tasks, it is not that sensitive to the number of OR-decompositions in the model.

In all of the experiments, the worst case times taken for all goal model size are: 10.7 minutes taken by Algorithm 8 to find global reconfigurations (on a goal model with 250 goals/tasks); 22 seconds taken by Algorithm 9 to find local reconfigurations (on a goal model with 400 goals/tasks), and 46 milliseconds taken by Algorithm 11 to find a global reconfiguration (on a goal model with 250 goals/tasks). These data further confirms our observation that the worst time performance does not necessarily come from experimenting on the largest goal models. Rather, they depend on the structure of the goal model.

Our experimental results show that the greedy algorithm, Algorithm 11, is the most efficient in finding a best global reconfiguration. On a goal model with 1000 goals/tasks, Algorithm 11 finds a best global reconfiguration in an average of 27 milliseconds. Algorithm 11 is also the least sensitive to the structural of the goal model, and it scales well
as the goal model size increases. Therefore, it is feasible to apply our reconfiguration approach to industrial sized applications with medium-sized requirements when Algorithm 11 is used.

### 6.3 Execution

If failures occur, the *Reconfiguration and Compensation Executor* component first executes compensation actions to bring the system back to a consistent state. We adopt a model of database long lived transactions (LLTs) for executions of compensation actions [GM87]. Domain experts provide compensation actions $C_1, C_2, \ldots, C_n$ for all tasks in a goal model (or all transitions in a statechart) $T_1, T_2, \ldots, T_n$ respectively. The executer component ensures that either the sequence $T_1, T_2, \ldots, T_n$ or the sequence $T_1, T_2, \ldots, T_j, C_j, \ldots, C_2, C_1$ is executed for some $0 \leq j < n$ [GM87]. Executions of these compensation actions restore a system to its previous consistent state when failures occur. It is noteworthy that in many applications, it may make sense to compensate for only failed goals/tasks, rather than for all actions that are successfully completed. Because compensations are application and domain dependent, the domain experts need to decide on a case-by-case basis on what the appropriate compensation behaviors for a given software system are.

After it executes compensation actions, the executor component reconfigures the monitored system by tapping into the system’s own configuration API. The system runs under the new configuration in its next execution session.

### 6.4 Discussions

This chapter presents three reconfiguration algorithms: (1) Algorithm 8 that finds best global reconfigurations. Its worst case performance is exponential to the size of the goal model. (2) Algorithm 9 that finds best local reconfigurations. Its worst case performance
is exponential to the size of the partial goal model it searches on. And (3) Algorithm 11 which is a greedy algorithm that finds one best global reconfiguration. It computes the sub-configuration for each node in the model once, and the number of sub-configurations computed is linear to the size of the goal model.

Although the greedy algorithm has the best worst case performance, the disadvantage of the greedy algorithm is that it assumes that it does not deal with contribution links between hard goals and tasks in a goal model. Rather, the algorithm only looks at contribution links that proceed from hard goals and tasks to soft goals. With this assumption, we can decide which subgoals and tasks to choose to fulfill a goal independently of the rest of the goal model. This assumption allows the algorithm to have linear time performance. Consider the example goal model in Figure 6.1, the decision of choosing $a_1$ vs. $a_2$ to fulfil $g_4$ is independent from choosing $a_5$ vs. $a_6$ to fulfil $g_6$. Contribution links between hard goals and tasks impose constrains on which goals/tasks can or cannot participate in the same configuration. Any approach that fully takes into account these contribution links to find an exact solution will be expensive. We recognize that the reconfiguration our greedy algorithm computes may not be satisfiable if conflicting contribution links are present between the chosen hard goals and tasks in the configuration. In future work, we plan to investigate this issue and improve our greedy algorithm.

Best-first graph search algorithms may be used to assist in reconfiguration. A* is a best-first graph search algorithm that finds the lowest cost path from a given initial node to one goal node [A*]. A* creates a graph with nodes and edges between nodes. Edge weights represent distances to travel between nodes. A* uses a heuristic function to determine which nodes to visit the next in the graph. The heuristic function contains two parts: the distance already traveled from the initial node, and an estimation of the distance to the goal. In order for A* to be optimal, its heuristic function needs to be admissible, meaning that it never overestimates the actual minimal cost of reaching the goal. Therefore, one of the biggest challenges to use A* is to define an admissible heuristic
function.

If we want to use A* algorithm to assist in finding a best reconfiguration, we need to create a search graph from the goal model. One possible representation of a goal model’s configuration search space as a graph is to let each node represent a possible partial or complete configuration. The initial node would be an empty configuration and the goal node be a satisfiable and complete configuration. Each node is associated with a contribution score that represents how well it contributes to soft goals. The edge cost between two nodes is the delta of their scores. Because the edge weights are deltas, they can be negative values.

We note three issues with the above described approach: (1) A* does not handle negative edge weights. To use A*, we need to either modify our scoring algorithm, or representation of the goal model configuration space. (2) The number of possible configurations is exponential to the goal model size. Therefore, the configuration search space is exponential. For this reason, we should not build the entire graph, rather, only build what is searched through. (3) There is no obvious heuristic function since there is no clear indicator of distance to the goal. If there is no heuristic that works well, the search would have to visit all nodes, whose number is exponential.

To address these issues, we would need to change representation of the goal model configuration space and our scoring algorithm. It is important to note that it is difficult to find an accurate and admissible heuristic which does not overestimate the distance to the goal. This is because the heuristic needs to be conservative - it has to be no bigger than the smallest possible sum of the edge weights along a path. If there are big differences between edge weights (such as when some nodes have many contribution links to soft goals and some have none), the heuristic will likely be far from optimal. When this happens, the algorithm needs to search through a large portion of the graph, making it inefficient [YDA06].

A potential solution to overcome these issues is to use conflict-directed A* search
[SW06], which identifies plateaus, orders them in best-first order, and uses a SAT solver to look for solutions that satisfy all constraints with the highest score. In the worst case, this approach would be very slow, because there are as many plateaus as combinations of tasks. However, the approach should be fast if a small number of the best plateaus can be found quickly and one of them is satisfied. A good avenue of future work is to create techniques to identifying plateaus efficiently and evaluating the effectiveness of conflict-directed A* search on reconfiguration of goal models.

Another possible solution is to think of the reconfiguration problem as a weighted constraint satisfaction problem (WCSP) [LS03, dGHLZ]. The WCSP solver will look for a solution that satisfies the constraints in a way that minimizes the costs, and ensures that all hard constraint are satisfied. This approach can be applied to reconfiguration of goal models by specifying the goal model in terms of weighted constraints. Goals and tasks in a goal model can be directly assigned to variables for WCSP. Goal model decompositions and contribution links would be added as constraints. A good avenue of future work is to identify good ways of assigning weights to the constraints created from contribution links and to evaluate the effectiveness of mapping reconfiguration of goal models to WCSP.

This chapter also describes our execution component at a conceptual level. We assumed that domain experts provide compensation actions for all tasks in a goal model. These compensation actions are executed to bring the system back to a consistent state if failures occur. If a task can not be compensated for, its compensation action would be empty. It is necessary to assume that domain experts provide such compensation actions, because they are domain and application dependent. Traceability links are needed to link between compensation actions’ source code and the goal model. The difficulties of extracting and maintaining traceability links apply here as well. In future work, we plan to explore techniques in treating traceability links.
Chapter 7

Industrial Case Studies

We collaborated with our industrial partner, Computer Associate Inc. (CA) [CA], on a project named “Computer Assisted Root Cause Analysis”. Four universities participated in this project including: University of Alberta, University of Toronto, University of Victoria, and University of Waterloo. Our main objective is to help operators of distributed and complex systems to identify failures and discover their root causes quickly. To this end, we integrated different research tools from various universities into a diagnostic platform for computer assisted root cause analysis. The tools in this platform are complementary to each other because they are designed to catch different kinds of failures. Our requirement monitoring and diagnostic tool monitors for failures that occur within a system. We describe our two case studies (SAM and Day Trading) in the following sections.

7.1 Spectrum Availability Manager (SAM)

Spectrum Availability Manager (SAM) is a CA product that is formally known as Service Availability Manager [sam]. Under the traditional IT service management infrastructure, separate tools are used to manage and analyze events generated by services across many IT layers. Many events are generated, and it is hard to correlate and analyze them.
Consequently, it is difficult for the IT managers to spot and resolve problems quickly, and to ensure that all services are available 24 x 7. CA SAM resolves this problem by providing a unified view of services and events. It integrates event management across different IT platforms and unites products and services under a single point of control. SAM improves service availability as it ensures the quality of service delivery and proactively identifies key risks to that quality.

The “Computer Assisted Root Cause Analysis” project is inspired by CA SAM. We applied our requirement monitoring and diagnostic framework to example failure scenarios [CAD] that CA would like SAM to spot and resolve quickly. Through this case study, we show that our tool can catch both application and infrastructure level system failures, and it can potentially be integrated into SAM for diagnostic reasoning.

### 7.1.1 Case Study Failure Scenarios

We worked on web server related failures for an example online ordering system [CAD]. Figures 7.1 to 7.6 present SAM’s investigative user interfaces (UI) for diagnosing two server related failures of the online ordering system. Investigative work flow goes as follows: the system administrator, Matthew, sees that the online ordering system has an unusually high response time of 7 seconds (Figure 7.1), exceeding the normal threshold of 5 seconds. Satisfaction levels of Service Level Agreements (SLA) and performance requirements only researched 75% and 80% respectively. It is of high priority to resolve this problem quickly. Matthew then brought up the detailed information of this online ordering system (Figures 7.6 and 7.3). Figure 7.6 gives the system’s service topology showing that the system is backed by three apache servers. Servers 1 and 3 are highlighted in red and yellow boxes respectively, indicating that there are performance alerts for these servers. Figure 7.3 gives the system’s alert page with detailed alerts for servers 1 and 3. The first alert indicates that server 1 is unable to bind to its port 4949 because the port is already taken by another application. The failure with server 1 is then resolved.
Figure 7.1: SAM UI for the Online Ordering System: High Response Time

The second alert in the alert page indicates that server 3 breached a performance alert - its average disk time exceeds the warning threshold. Matthew needs to find out the root cause to this problem. Server 3’s performance report for the past 24 hours (Figure 7.5) shows that after 9:00am, the number of concurrent connections running on server 3 exceeds the normal threshold level. Consequently, server 3’s “health” dropped below threshold, and its response time increased above the threshold. Matthew brought up the troubleshooting page for the problem of “high sever response time” (Figure 7.6). This page states that all servers in a web farm should serve roughly equal number of requests that give them equal response time. If this is not the case, three possible explanations are: (1) not all servers in the web farm are started and able to receive requests. (2) Not enough servers in the web farm. An indication of this is all servers on the web farm serve high numbers of requests and have high response times. And (3) load balancer is not
working. A loader balancer balances requests among all servers, ensuring that no single server is overloaded. Specifically, if the *nlb.exe* process is no longer active or running flat at 0%, we know the load balancer is not working. Matthew checks all three possibilities and finds out that the load balancer started to run flat at 0% on server 3 since 9:00am (shown in the bottom right corner of Figure 7.5). This causes server 3 to receive high numbers of requests and have a high response time. With the help of SAM, Matthew discovered the root causes of this problem quickly.

### 7.1.2 Our Results

We apply our requirement monitoring and diagnostic framework to the online ordering system example. We show that our framework catches both discussed server failures:
Figure 7.3: SAM UI for the Online Ordering System: Server 1 not Bind to Port

(1) server 1 not bind to its port and (2) load balancer runs flat on server 3, causing server 3 to have a high response time. Figure 7.7 gives a partial goal model of the online ordering system. The root goal is $G_1$: *Run on line ordering system*. It is AND-decomposed to goals $G_2$ (*take orders*), $G_3$ (*process orders*), and $G_4$ (*ship product*). $G_3$ is further decomposed to goals $G_5$ (*run business logic*) and $G_6$ (*maintain low response time*). Goals $G_2$, $G_3$, $G_4$, and $G_5$ are application level goals - they represent system’s business logic at the application level. Since the failures we deal with are infrastructure level server failures, we do not show fully decomposed application level goals in Figure 7.7.

Goal $G_6$ (*maintain low response time*) is an infrastructure level goal and it represents the requirement for the online ordering system to serve requests with a low response time. $G_6$ is decomposed into goals $G_7$ (*Database running and respond with low delay*)
Figure 7.4: SAM UI for the Online Ordering System: Server 3 is Overloaded

and G8 (All servers on the web farm serve with equal response time). According to SAM’s knowledge on web servers (Figure 7.6), G8 is satisfied if (1) all servers on the web farm are started and running (represented by G9), (2) there are sufficient number of servers in a web farm (represented by G10), and (3) load balancer is working on all servers (represented by a7 and a11).

Goal G9 (All servers are started and running) is AND decomposed into goals G12, G13, and G14, which are then decomposed into tasks a1 to a6. These goals/tasks represent the requirements that all servers 1, 2, and 3 need to start, and bind to their respective ports. Goal G10 (Maintain sufficient number of servers in the web farm) is AND-decomposed into tasks a8 (server 1 not overloaded), a9 (server 2 not overloaded) and a10 (server 3 not overloaded). As noted before, if all servers are overloaded in a web farm, it is evidence enough that there are not enough servers in the web farm. Finally
Figure 7.5: SAM UI for the Online Ordering System: Load Balancer not Working on Server 3

the load balancer needs to be started (task $a7$), and be active on all servers (task $a11$).

Table 7.4 lists preconditions and effects for all monitored tasks in the goal model given in Figure 7.7. These preconditions and effects are propositional formulas in CNF which need to be true before and after their respective tasks are successfully executed. Table 7.2 contains the sample log data generated for all the tasks in Table 7.4. For each task $a$, the log data contains $a$’ occurrence and truth values of literals specified in $a$’s precondition and effect. Each log instance is associated to a timestep.

Our framework analyzes the log data and generates three Participating Diagnostic Components (PDCs) for a specific execution session $s$. Table 7.3 lists our diagnostic
Figure 7.6: SAM UI for the Online Ordering System: Diagnostic Knowledge Base results with explanations. The three PDCs \( FD(a2, s) \), \( FD(a10, s) \), and \( FD(a11, s) \) represent the failures of server 1 can not bind to its port, server 3 is overloaded, and load balancer runs flat on server 3, respectively. Our framework catches both server failures.

### 7.2 Day Trading Application

Day Trading is a stock trading application developed in University of Victoria\(^1\). The day trading application allows authenticated stock traders to check stock quotes, set buy stock and sell stock triggers, and buy and sell stocks. Figure 7.8 gives the high level goal model for Day Trading. The root goal \( g1 \) (provide day trading application) is AND-decomposed

\(^{1}\)The research team at the University of Victoria is lead by Dr. Hausi Muller. The author of the application is Lin Lei
Figure 7.7: Goal model for the Online Ordering System

Figure 7.8: High Level Goal Model of the Day Trading Application

to goals $g_2$ (authenticate user), $g_{10}$ (provide service to user), and $g_{48}$ (maintain good system health). $G_2$ (authenticate user) consists of getting user login information ($g_3$) and verifying the information ($g_4$). To accomplish the provide services to user goal ($g_{10}$), the application first asks a user to select a transaction ($a_{10}$), then perform the selected transaction ($g_{11}$). The decomposition of goal $g_{11}$ (perform transaction) is shown Figure
Figure 7.9: Decomposition of Goal "Perform Transaction" of Figure 7.8
Table 7.1: Annotated Goal Model for the Online Ordering System

<table>
<thead>
<tr>
<th>Task</th>
<th>Precondition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>¬server1 started</td>
<td>Server1 started</td>
</tr>
<tr>
<td>a2</td>
<td>Server1 started ∧ port 4949 available</td>
<td>Server1 bind to port 4949</td>
</tr>
<tr>
<td>a3</td>
<td>¬server2 started</td>
<td>Server2 started</td>
</tr>
<tr>
<td>a4</td>
<td>Server2 started ∧ port x2 available</td>
<td>Server2 bind to port x2</td>
</tr>
<tr>
<td>a5</td>
<td>¬server3 started</td>
<td>Server3 started</td>
</tr>
<tr>
<td>a6</td>
<td>Server3 started ∧ port x3 available</td>
<td>Server3 bind to port x3</td>
</tr>
<tr>
<td>a7</td>
<td>¬ load balancer started</td>
<td>Load balancer started</td>
</tr>
<tr>
<td>a8</td>
<td>Server1 bind to port x1</td>
<td>number of connections on Server1 &lt; 50% threshold</td>
</tr>
<tr>
<td>a9</td>
<td>Server2 bind to port x2</td>
<td>number of connections on Server2 &lt; 50% threshold</td>
</tr>
<tr>
<td>a10</td>
<td>Server3 bind to port x3</td>
<td>number of connections on Server3 &lt; 50% threshold</td>
</tr>
<tr>
<td>a11</td>
<td>Load balancer started</td>
<td>¬nlb.exe runs flat on server1 ∧ ¬nbl.exe runs flat on server2 ∧ ¬nbl.exe runs flat on server3</td>
</tr>
</tbody>
</table>

7.9. Goal \(g_{48}\) (maintain system health) is satisfied when the system periodically checks that: (1) average CPU queue length < 8 (a79), (2) average CPU usage ≤ 95% (a80), (3) RAM usage ≤ 80% (a81), (4) free RAM ≥ 10MB (a82), (5) page fault per second ≤ 180 (a83), (6) disk idle time ≥ 30% (a84), (7) swap usage ≤ 90% (a85), and (8) disk space usage ≤ 90% (a86). Goals \(g_2\) (authentic user) and \(g_{10}\) (provide service to user) are application level goals - they describe what the application does. Goal \(g_{48}\) (maintain system health) is an infrastructure level goal because it describes the requirements of infrastructure level hardware usage.
### Table 7.2: Sample Log Data for the Online Ordering System

<table>
<thead>
<tr>
<th>Task</th>
<th>Log Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>¬server1 started(1); occ(a1, 2); server1 started(3);</td>
</tr>
<tr>
<td>a2</td>
<td>server1 started(3); ¬port 4949 available(4); occ(a2, 5); ¬server1 bind to port 4949 (6)</td>
</tr>
<tr>
<td>a3</td>
<td>¬server2 started(7); occ(a3, 8); server2 started(9);</td>
</tr>
<tr>
<td>a4</td>
<td>server2 started(9); port x2 available(10); occ(a4, 11); server2 bind to port x2(12)</td>
</tr>
<tr>
<td>a5</td>
<td>¬server3 started(13); occ(a5, 14); server3 started(15);</td>
</tr>
<tr>
<td>a6</td>
<td>server3 started(15); port x3 available(16); occ(a6, 17); server3 bind to port x3(18)</td>
</tr>
<tr>
<td>a7</td>
<td>¬load balancer started(19); occ(a7, 20); load balancer started(21);</td>
</tr>
<tr>
<td>a8</td>
<td>¬Server1 bind to port 4949(19); ¬occ(a7, 20); number of connections on server1 &lt; 50% threshold(21);</td>
</tr>
<tr>
<td>a9</td>
<td>Server2 bind to port x2(22); occ(a8, 23); number of connections on server 2 &lt; 50% threshold (24);</td>
</tr>
<tr>
<td>a10</td>
<td>Server3 bind to port x3(25); occ(a8, 26); ¬number of connections on server 3 &lt; 50% threshold(27);</td>
</tr>
<tr>
<td>a11</td>
<td>load balancer started(28); occ(a11, 29); ¬nlb.exe runs flat on server1(30); ¬nlb.exe runs flat on server2(31); nlb.exe runs flat on server3(32)</td>
</tr>
</tbody>
</table>

We chose to monitor the satisfaction of the high level goals of the Day Trading application (Figure 7.8): goals $g2$: authenticate user, $g10$: provide service, and $g48$: maintain system health. Table 7.4 lists their preconditions and effects. Literals specified in preconditions and effects can be further defined. Propositional formulas that define domain literals are added to $\Phi$ as domain constrains. For example, $g48$’s effect is pass_all_checks.
Table 7.3: Participating Diagnostic Components for the Online Ordering System

<table>
<thead>
<tr>
<th>PDCs</th>
<th>Explanations</th>
</tr>
</thead>
<tbody>
<tr>
<td>FD(a2, s)</td>
<td>Task a2 (server 1 bind to port) failed because it occurred at timestep 5</td>
</tr>
<tr>
<td></td>
<td>when its precondition (port 4949 available) was false at timestep 4.</td>
</tr>
<tr>
<td>FD(a10, s)</td>
<td>Task a10 (server 3 not overloaded) failed because its effect</td>
</tr>
<tr>
<td></td>
<td>(number of connections on server3 &lt; 50%) was false at timestep 27,</td>
</tr>
<tr>
<td></td>
<td>after its occurrence at timestep 26.</td>
</tr>
<tr>
<td>FD(a11, s)</td>
<td>Task a11 (load balancer active on all servers) failed because one</td>
</tr>
<tr>
<td></td>
<td>predicate in its effect (¬nbl.exe runs flat on server3) was false at</td>
</tr>
<tr>
<td></td>
<td>timestep 32, after its occurrence at timestep 29.</td>
</tr>
</tbody>
</table>

Table 7.4: Preconditions and Effects of Monitored Goals in the Day Trading Application

<table>
<thead>
<tr>
<th>Monitored Goals</th>
<th>Precondition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>g2:authenticate user</td>
<td>internet_connection</td>
<td>user_authenticated</td>
</tr>
<tr>
<td>g10:provide service</td>
<td>user_authenticated</td>
<td>service_provided ∨ service_canceled</td>
</tr>
<tr>
<td>g48:maintain system health</td>
<td>service_running</td>
<td>pass_all_checks</td>
</tr>
</tbody>
</table>

Literal pass_all_checks can be further defined as:

\[
pass\_all\_checks \leftrightarrow \neg nbl.exe\_runs\_flat\_on\_server3
\]

At run time, the instrumented program produces log data that contains truth values of the literals specified in monitored goals’ preconditions and effects, as well as task occurrences. The following is an example log data:
internet_connection(1), occ(a1, 2), occ(a2, 3), occ(a3, 4), occ(a4, 5), occ(a9, 6), user_authenticated(7), occ(a10, 8), occ(a12, 9), occ(a13, 10), service_provided(11), service_running(12), occ(79, 13), occ(80, 14), occ(81, 15), occ(82, 16), occ(83, 17), occ(84, 19), occ(85, 20), occ(86, 21), pass_all_checks(22).

Our framework analyzes the log data, and infers that no requirements are denied and the execution was successful. It then returns one core diagnosis containing \( \neg FD \) predicates over all the goals and tasks in the goal model, indicating that all requirements are satisfied.

## 7.3 Discussions

Our requirement monitoring and diagnosis research is useful for verifying that a system’s run time behavior (as recorded in the log) complies with its goal model. Our approach can be applied to real world applications if these three conditions hold: (1) we can model the correct behaviors of the monitored system using hard goals and tasks, (2) we can associate correct preconditions and effects for these hard goals/tasks, and (3) we can instrument the system in such a way that it produces log data that contain truth values of literals specified in these preconditions and effects, as well as task occurrences. Our framework monitors for failures of a system’s functional requirements. Other research tools participating in the “Computer Assisted Root Cause Analysis” project are complementary to our work as they are designed to monitor for other types of failures (such as patterns of failure events) [CAD].

CA researchers that we collaborated with recognize the value of our research. The main, and the only real concerns, are with the three conditions that we listed above. First of all, there is concern about the creation of the goal model. Manual generation of the goal model requires expert knowledge of the application and the process can be
time consuming. CA researchers think that ideally goal model generation should be fully automated. Or at least majority of the model should be created automatically and the rest should be created manually in a straight forward and easy to understand way. Associating correct preconditions and effects can be even more challenging than creating correct goal models. These concerns can be partially addressed by having a good reverse engineering tool that extracts such information from the source code. However, a domain expert must look over the extracted goal model to make sure that it is correct. This is because if the implementation is faulty, the goal model that is extracted from it would also be faulty. Our approach will end up verifying that the faulty system behavior do comply with its faulty specification.

There are also major concerns with instrumenting the software system to generate log data required for diagnosis. First of all, in many cases system’s source code is not available. The instrumentation component in [Zho08] does not modify system source code (it instruments at the byte code level). However, traceability links are required to link between source code and goal model, so that the instrumentation component knows what aspects should be generated. In addition, CA would like the generated log data to be in Common Base Event (CBE) format, and instrumentation able to deal with more complex issues such as multi-threading. The instrumentation component needs to be extended to address these issues. This thesis’s contribution on instrumentation is it defines what needs to in the log file and log data format. It is beyond the scope of this thesis to deal with implementation issues of the instrumentation component.
Chapter 8

Conclusions and Future Work

This thesis presents an autonomic architecture that enables high variability legacy systems to self-reconfigure when failures occur. The proposed framework consists of monitoring, diagnosis, reconfiguration, and execution components corresponding to the monitor, analyze, plan, and execution (MAPE) components of an autonomic system. We illustrate our framework with two medium-sized, publicly-available case studies: a Web-based email client (Squirrel Mail) [Cas07], and an ATM simulation [Bjo]. We evaluate the framework’s performance through a series of experiments on (randomly generated) medium to large size goal models and statecharts. The results demonstrate that our approach scales well to the size of the specification, and it is feasible to apply our approach to industrial applications with medium to large specifications.

This work advances the field in several respects. (1) It contributes to the research areas of requirements monitoring, diagnosis, and autonomic computing. Major research on requirement monitoring systems include Fickas’ and Feather’s runtime monitoring framework [FFLP98, FF95], Robinson’s ReqMon [Rob05b, Rob05c], and our work on requirement monitoring and diagnosis [WMYM07, WMYM09]. Our work differs from [FFLP98, FF95, Rob05b, Rob05c] in that we monitor for the malfunctions of the system rather than its environment or domain. In addition, we can automatically infer diagnoses
given a model of system requirements and log data. (2) We proposed the first SAT-based solution to the software diagnosis problem that is sound and complete. (3) We implemented a reconfiguration component [WM09] with an efficient reconfiguration algorithm (Algorithm 11). And (4), our framework is implemented and evaluated. Experimental results show that our framework scales well to the size of system specification.

We note several features and limitations of our work. Our framework depends on the availability of system specifications (either requirement goal models or statecharts) as well as traceability links. Both goal models and statecharts have a hierarchical and layered structure, since goals and states decompose into subgoals and sub-states. This structure enables us to model (and analyze) a software system at different levels of granularity. For example, it can describe a large-scale, complex software system using a small goal model or a small statechart at a high level of granularity. Consider the two cases studies discussed in this thesis. Although the SquirrelMail study (69711 LOC) is larger than the ATM simulation (5000 LOC), the SquirrelMail’s goal model (in Figure 4.1, with 11 goals/tasks) is smaller than that of the ATM simulation (in Figure 4.3, with 88 goals/tasks) because it is modeled at a coarser level of granularity. In cases where a fine-grained specification is not available, one can always model a system at a high level of abstraction.

Our approach requires traceability links to map denials of specification elements to a monitored system’s source code for purposes of pinpointing possible faulty components. Traceability links may also be represented at different granularity levels. Higher level traceability links map high level specification elements (goals/tasks or states/transitions) to larger-scaled software components, such as sub-systems and servers. Lower level traceability links map lower level specification elements to smaller-scaled components, such as one or several modules. Higher level traceability links can be used to relate specifications to larger-scaled sub-systems of the monitored system in the absence of detailed traceability links. The assumptions and limitations of our work on monitoring and diagnosis
are discussed in details at the end of Chapters 4 and 5.

We converted the problem of diagnosing software requirements into a SAT problem that can be solved by existing SAT solvers. A SAT solver is useful in our work because we need to do backward search from higher level goals to tasks in a goal model. Given a denied goal \( g \), we need to infer which of \( g \)'s subgoals and tasks that might have been denied to explain \( g \)'s denial. If any of \( g \)'s subgoals and tasks are associated with preconditions and effects, their truth values are also recorded in the log. These data, if available, constraints the search space further. SAT solvers are very efficient in performing such backward search, on a graph with a given a set of constraints, to find one satisfiable solution. Thus a SAT solver is useful when goals are monitored, and we need to do backward label propagation from goals to tasks. If all tasks are monitored, and we only need to do forward label propagation from tasks to goals, a SAT solver is not necessary. Although SAT solvers perform forward label propagation effectively as well. [GMNS02] presented an efficient algorithm for forward label propagation from tasks to goals in a goal model.

There are some similarities between our axiomatizations of goal/task denials in a goal model (presented in Chapter 4) and axiomatizations of transition denials in a statechart (presented in Chapter 5). We formulate the denial of goals/tasks in terms of the truth values of the predicates representing their occurrences, preconditions and effects. If a task's precondition is true and the task occurred at timestep \( t \), and if its effect holds at the subsequent timestep \( t + 1 \), then the task is not denied at timestep \( t + 1 \). Similarly, we formulate the denial of transitions in terms of the truth values of the predicates representing its occurrence, enabledness and effect. If a transition is enabled at timestep \( t \), the transition occurred (fired) at timestep \( t+1 \), and the transition's effect is true at \( t+2 \), then the transition is not denied at timestep \( t+2 \). Recall that the statechart semantics requires that a transition's enabledness, occurrence and effect to be observed in three consecutive steps.
The enabledness of a transition can be thought of as precondition being true for a task. The difference is a task does not have to occur when its precondition is true. But a transition has to fire if it is enabled. This is because the transition’s triggering events and conditions are encoded into its enabledness. We infer a transition has failed if it is enabled at timestep $t$ and it did not occur at $t+1$. We do not infer a goal/task has denied if its precondition is true and it did not occur in the same or subsequent timesteps. Occurrences of goals/tasks are similar to transition occurrences. We say a goal has occurred if all of the tasks fulfilling the goal have occurred. We say a transition has occurred if all the actions labeled on the transition have occurred, and all exit and entry actions associated to the transition’s source and target states have occurred. A transition’s effect is true if system’s active states are updated correctly. Such state change information can also be associated to the effects of goals/tasks in a goal model, if they are known.

To evaluate the performance of our work, we experimented on randomly generated goal models and statecharts, instead of specifications of real systems. The largest goal model we experimented on contained 3000+ goals and tasks, and the largest statechart contained 1000 states and 10,000 transitions. It would have been difficult to obtain real world goal models and statecharts of this scale. Our experimental results show that our approach scales with the size of the specification. However, more industrial case studies are needed to further evaluate our approach.

We plan, in the future, to extend our framework in three ways. (1) The current monitoring and diagnostic components can only find failures of a system’s functional requirements and failures within a system. We will extend the framework to monitor for other kinds of failures, such as violations of a system’s non-functional requirements and malicious attacks, etc. (2) The current repair/reconfiguration component only applies to systems with requirement models. It can’t repair/reconfigure systems with statechart specifications. The reconfiguration component also applies only to high variability
software systems that have alternative ways of fulfilling their requirements. We plan to extend the framework by designing a repair mechanism for systems with statecharts. We also intend to explore other repair methodologies, especially ones that don’t depend on reconfiguration. These extensions will allow us to design adaptive systems capable of performing a wider range of self-management tasks, including self-optimization, self-healing, and self-protection. (3) We plan to extend the framework so that it works with other types of software systems, web services in particular. Web services are the software of the future, since more and more software systems will interact with each other as services. How best to monitor for correct executions of web services and business processes will therefore remain a key research problem in the years ahead.

Along a somewhat different dimension, we plan to develop methodologies to address requirements evolution through requirements monitoring. Software systems evolve as a result of changing requirements, technologies, business objectives, domain assumptions, etc. Little attention has been devoted to requirements evolution - requirements changes that occur after a system has begun operation [EMW09]. Models of software systems often become obsolete quickly because of the lack of techniques and tools for model evolution. We plan to develop tools that address this problem. In particular, we plan to extend our framework’s monitoring and diagnostic components so that they can distinguish between failures and system evolutions, instead of treating all system behaviors that deviate from requirements as requirements denials. We intend to do this by developing techniques that propagate software evolutions to requirement goal models, thereby effecting model evolution. The framework can then use the updated requirement model as the basis for monitoring, diagnosis, repair, and compensation feedback loop.
Bibliography


