An Automated Approach to Monitoring and Diagnosing Requirements

Yiqiao Wang
University of Toronto, Canada
yw@cs.toronto.edu

Sheila A. McIlraith
University of Toronto, Canada
sheila@cs.toronto.edu

Yijun Yu
The Open University, UK
y.yu@open.ac.uk

John Mylopoulos
University of Toronto, Canada
jm@cs.toronto.edu

ABSTRACT

Monitoring the satisfaction of software requirements and diagnosing what went wrong in case of failure is a hard problem that has received little attention in the Software and Requirement Engineering literature. To address this problem, we propose a framework adapted from artificial intelligence theories of action and diagnosis. Specifically, the framework monitors the satisfaction of software requirements and generates log data at a level of granularity that can be tuned adaptively at runtime depending on monitored feedback. When errors are found, the framework diagnoses the denial of the requirements and identifies problematic components. To support diagnostic reasoning, we transform the diagnostic problem into a propositional satisfiability (SAT) problem that can be solved by existing SAT solvers. We preprocess log data into a compact propositional encoding that better scales with problem size. The proposed theoretical framework has been implemented as a diagnosing component that will return sound and complete diagnoses accounting for observed aberrant system behaviors. Our solution is illustrated with two medium-sized publicly available case studies: a Web-based email client and an ATM simulation. Our experimental results demonstrate the feasibility of scaling our approach to medium-size software systems.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Diagnostics, Monitors

General Terms

Design, Theory, Verification

Keywords

Requirements monitoring, Diagnostics

ASE'07, November 4–9, 2007, Atlanta, Georgia, USA.

1. INTRODUCTION

Monitoring software for requirements compliance is necessary for any operational system. Yet, design of runtime monitoring and diagnostic components has received little attention in the Requirement Engineering (hereafter RE) literature. The aim of our research is precisely this: to develop a tool-supported methodology for designing and running monitoring and diagnostic components.

In this paper, we propose an adaptive monitoring framework, complemented by a SAT-based diagnostic framework adapted from artificial intelligence (AI) theories of action and diagnosis. Software requirements are represented as goal models that can be either reverse engineered from source code using techniques we presented in [24], or provided by requirements analysts. In addition, we assume that traceability links are provided, linking source code and requirements in both directions.

The monitoring component monitors requirements and generates log data at different levels of granularity that can be tuned adaptively at runtime depending on monitored feedback. A diagnosing component analyzes generated log data and identifies errors corresponding to aberrant system behaviors that lead to the violation of system requirements. When errors are found, the diagnostic component identifies root causes.

Propositional satisfiability (SAT) is the problem of determining whether a propositional formula admits any truth assignments to its literals that render it true. Recent advances in SAT solver technology have encouraged SAT-based applications to software engineering problems. We transform the problem of diagnosing software systems into a SAT problem by encoding goal model relations and log data into propositional formulae that can be used by an off-the-shelf SAT solver to conjecture possible diagnoses.

The diagnostic component is implemented with two diagnosing algorithms. The first algorithm is sound and complete with respect to the diagnoses it generates. Each SAT truth assignment corresponds to a correct diagnosis that represent requirements satisfaction and denial, If system's requirements are denied, problematic components are identified. We find a complete set of such diagnoses by having the SAT solver return all possible truth assignments. The second algorithm improves efficiency when incomplete log data is present by making the SAT solver only return a subset of possible truth assignments. This second algorithm is sound and complete with respect to participating diagnos-

tic component sets. We preprocess the log data to create a compact propositional encoding that scales with problem size, making our approach more amenable to industrial-sized applications.

We illustrate and evaluate our framework on two medium-sized publicly available case studies: a Web-based email client (Squirrel Mail) [2], and an ATM simulation [1]. These case studies demonstrate the feasibility of scaling our approach to medium-size software systems with medium-size requirement models.

The rest of the paper is organized as follows. Section 2 gives theoretical foundations to our work. Section 3 presents our proposed adaptive monitoring and SAT-based diagnostic framework. In Sections 4 and 5, we discuss implementation issues and evaluate our framework on the SquirrelMail and ATM simulation case studies. We discuss related work in Section 6 and conclude in Section 7.

2. PRELIMINARIES

2.1 Goal Models

Requirements Engineering (RE) is a branch of software engineering that deals with elicitation and analysis of system requirements. In recent years, goal models have been used in RE to model and analyze stakeholder objectives [22]. Functional and non-functional requirements are represented as hard goals and soft goals respectively [16]. 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 and diagnosis. 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 [8], apart from decomposition links, hard goals 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 meaning of links $++D$ and $--D$ are dual w.r.t. to $++S$ and $--S$ respectively by inverting satisfiability and deniability. Links $++$, and $--$ are shorthand for the $++S$, $++D$, and $--S$, $--D$ relationships respectively, and they represent strong MAKE($++$) and BREAK($--$) contributions between hard goals. In this paper, the partial (weaker) contribution links HELP($+$) and HURT($-$) are not included between hard goals because we do not reason with partial evidence for goal satisfaction and denial. These weaker links proceed from hard goals to softgoals in our work. The class of goal models used in our work has been formalized in [8], where sound and complete algorithms are provided for inferring whether a set of root-level goals can be satisfied.

As an extension, 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) that must be true before and after (respectively) a goal is satisfied or a task is successfully executed. Monitoring switches can be switched on/off to indicate whether the corresponding goal/task is to be monitored.

2.2 SAT Solvers

The propositional satisfiability (SAT) problem is concerned with determining whether there exists a truth assignment μ to variables of a propositional formula Φ 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.

The earliest and most prominent SAT algorithm is DPLL (Davis-Putnam-Logemann-Loveland) [4], which uses backtracking search. Even though the SAT problem is inherently intractable, there have been many improvements to SAT algorithms in recent years. Chaff [15], BerkMin [9] and Siege [20] are among the fastest SAT solvers available today. For our work, we use SAT4J [12], an efficient SAT solver that inherits a number of features from Chaff.

3. OUR FRAMEWORK

Figure 1 provides an overview of our proposed framework. The input to the framework is the monitored program's source code, and its corresponding goal model representing the system's requirements. The goal model can be either reverse engineered from the program using techniques we presented in [24] or it can be modeled by requirement analysts. Requirement analysts annotate the goal model with monitoring switches, preconditions and effects for goals and tasks. When these switches are enabled, the satisfaction of the corresponding goals and tasks is monitored at run time. From the input goal model, the *parser* component obtains goal/task relationships, goals and tasks to be monitored, and their preconditions and effects. The *parser* then passes this data to the *instrumentation* and *SAT encoder* components in the monitoring and diagnostic layers respectively.

In the monitoring layer, the *instrumentation* component inserts software probes into the monitored program at the appropriate places. At run time, the *instrumented program* generates log data that contains program execution traces and values of preconditions and effects for monitored goals and tasks. Offline, in the diagnostic layer, the *SAT encoder* component transforms the goal model and log data into a propositional formula in CNF which 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 and diagnostic component sets.

In our framework a diagnosis specifies for each goal and task, whether it is fully denied or not. If a denial of system requirements is found, it is traced back to the source code to identify the problematic components. The *diagnosis analyzer* analyzes the returned diagnoses, and may increase monitoring granularity by switching on monitoring switches for subgoals of a denied parent goal. When this is done, subsequent executions of the instrumented program generate more complete log data. More complete log data means fewer and more precise diagnoses, due to a larger SAT search

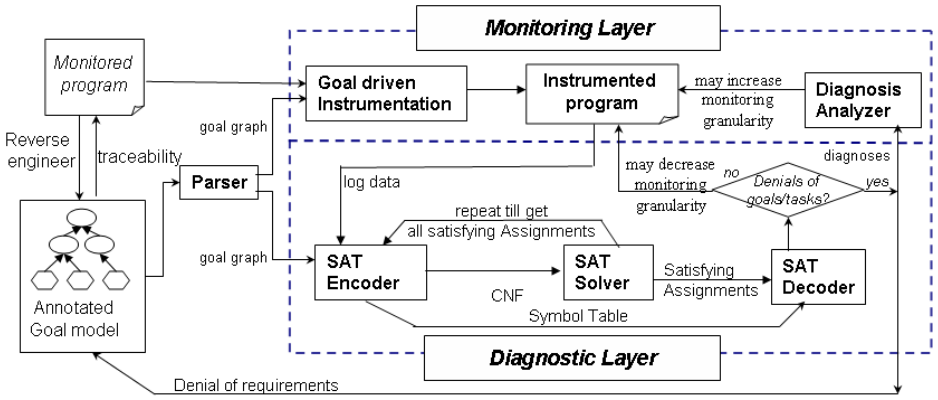

Figure 1: Framework Overview

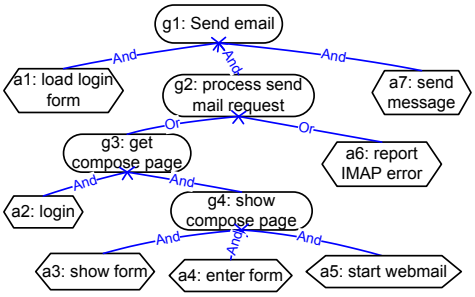

Figure 2: Squirrel Mail Goal Model

space with added constraints.

If no diagnoses are found, the system is running correctly. Monitoring granularity may also be decreased to monitor fewer (higher level) goals in order to reduce monitoring overhead. The steps described above constitute one execution session and may be repeated.

3.1 A Running Example

We use the SquirrelMail [2] case study as a running example throughout this paper to illustrate how our framework works. SquirrelMail is an open source email application that consists of 69711 LOC written in PHP. Figure 2 presents a simple, high-level goal graph for SquirrelMail with 4 goals and 7 tasks, shown in ovals and hexagons, respectively.

The root goal $g1$ is AND decomposed into task $a1$, goal $g2$, and task $a7$. $g2$ is OR decomposed into task $a6$ if the email IMAP server is not found and goal $g3$ if otherwise. $g3$ is decomposed into task $a2$, and goal $g4$, which is further AND decomposed into three tasks: $a3$, $a4$, and $a5$.

3.2 Adaptive Goal Monitoring

Satisfaction of software system requirements can be monitored at different levels of granularity depending on monitoring requirements, objectives, knowledge of the software system, and monitoring resources available. The finest level of monitoring granularity is at the functional level where all leaf level tasks are monitored. In this case, complete log data is generated, and a single precise diagnosis can be inferred. Of course, the disadvantage of complete monitoring is high monitoring overhead and the possible degradation of system performance. Coarser levels of granularity only monitor higher-level goals in a goal model. In this case, less complete log data is generated, leading to less precise diagnoses. Clearly, the advantage of coarse-grain monitoring is reduced monitoring overhead and complexity.

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 to allow 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 [14]. Errors may be introduced if specified preconditions and effects for goals and tasks do not completely or correctly capture the software system's dynamics. Detecting or dealing with discrepancies between a system's implementation and its goal model is beyond the scope of this paper, and we assume that both the goal model and its associated preconditions and effects are correct and complete.

Each task occurrence is associated with a specific logical timestep t. We introduce predicate $occ_a(a_i, t)$ to specify occurrences of tasks a_i at timestep t. We say a goal has occurred in an execution session s if and only if all the tasks in its decomposition have occurred in s, and we associate two timesteps, t_1 and t_2, to goal occurrences representing the timesteps of the first and the last executed task in the goal's decomposition in execution session s. We introduce predicate $occ_g(g_i, t_1, t_2)$ to specify occurrences of goals g_i that start and end at timesteps t_1 and t_2 respectively.

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.*

3.2.1 SquirrelMail Example Log Data

Table 1 lists the details of each goal/task in the SquirrelMail goal model (Figure 2) with its monitoring switch status (column 2), and associated precondition and effect (columns 3 and 4). The requirement analyst chooses which goals/tasks

Table 1: Squirrel Mail Annotated Goal Model

Goal/ Task	Monitor switch	Precondition	Effect
a1	on	URL entered	correct form
a2	on	¬wrongIMAP∧ correct form	correct key
a3	off	correct key	form shown
a4	off	form shown	form entered
a5	off	form entered	webmail started
a6	on	wrongIMAP	error reported
a7	on	webmail started	email sent
g1	off	URL entered	email sent ∨ error reported
g2	off	correct form ∨ wrongIMAP	webmail started ∨ error reported
g3	off	correct form ∧ ¬wrongIMAP	webmail started
g4	on	correct key	webmail started

are to be monitored at runtime and specifies for the goals and tasks their associated preconditions and effects. In this example, the satisfaction of goal $g4$, and tasks $a1$, $a2$, $a6$, and $a7$ are monitored. The following is an example log data consisting of truth values of monitored goals/tasks' preconditions and effects, and occurrences of tasks:

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

3.3 SAT-Based Goal Diagnosing

Executions of tasks in some order form a plan which if executed successfully leads to satisfaction of the root goal. We associate a unique execution session ID, s, with each session of a plan executed in fulfillment of the root goal. Goal satisfaction or denial may vary from one session to another. 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 introduce a distinct predicate FD to express full evidence of goal and task denial at a certain timestep or during a specific session. FD predicates take two parameters: the first parameter is either a goal or a task specified in the goal model and the second parameter is either a timestep or a session id.

3.3.1 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 tasks'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 the same timestep 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. All propositional literals are grounded to domain instances.

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 is false at $t+1$:*

$$FD(a, t+1) \leftrightarrow occ_a(a, t) \wedge (\neg p(t) \vee \neg q(t+1)) \quad (1)$$

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 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.*

$$FD(g, t_2+1) \leftrightarrow occ_g(g, t_1, t_2) \wedge (\neg p(t_1) \vee \neg q(t_2+1)) \wedge (t_1 \leq t_2) \quad (2)$$

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.*

$$FD(a, t) \rightarrow FD(a, s) \quad (3)$$

$$FD(g, t) \rightarrow FD(g, s) \quad (4)$$

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:

$$FD(a7, 2) \leftrightarrow occ_a(a7, 1) \wedge (\neg webmail\ started(1) \vee \neg email\ sent(2))$$

$$FD(g4, 2) \leftrightarrow occ_g(g4, 1, 1) \wedge (\neg correct\ key(1) \vee \neg webmail\ started(2))$$

$$FD(a7, 2) \rightarrow FD(a7, s)$$

$$FD(g4, 2) \rightarrow FD(g4, s)$$

3.3.2 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 retain the same values are often called frame axioms and they 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 [18] 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

[1]In many axiomatizations we assume that $occ_a(a, t) \rightarrow p(t)$.

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 is not denied at t.

Explanation Closure Axioms are described by axioms (5) and (6) 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 or one of the goals g_j, must have occurred by timestep t and is not denied at the subsequent timestep $t+1$.

If f is in a positive effect of tasks $a_1 \ldots a_n$ and goals $g_1 \ldots g_m$,

$$\begin{array}{l} \neg f(t) \wedge f(t+1) \leftrightarrow \\ \bigvee_i (occ_a(a_i,t) \wedge \neg FD(a_i,t+1)) \vee \\ \bigvee_j (occ_g(g_j,t_1,t) \wedge \neg FD(g_j,t+1) \wedge (t_1 \leq t)) \end{array} \quad (5)$$

If f is in a negative effect of tasks $a_1 \ldots a_n$ and goals $g_1 \ldots g_m$,

$$\begin{array}{l} f(t) \wedge \neg f(t+1) \leftrightarrow \\ \bigvee_i (occ_a(a_i,t) \wedge \neg FD(a_i,t+1)) \vee \\ \bigvee_j (occ_g(g_j,t_1,t) \wedge \neg FD(g_j,t+1) \wedge (t_1 \leq t)) \end{array} \quad (6)$$

In the SquirrelMail case study, according to Table 1, only the task $a7$ 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:

$$\neg email\ sent(1) \wedge email\ sent(2) \leftrightarrow occ_a(a7,1)\ \wedge \neg FD(a7,2)$$

3.3.3 Axiomatization of Label Propagations

Axioms (7) and (8) 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, if and only if at least one (or all) of the subgoals or tasks in its decomposition is (or are) denied in that session.

$(g_1 \ldots g_n, a_1 \ldots a_m) \xrightarrow{AND} g$:

$$FD(g,s) \leftrightarrow (\bigvee_i FD(g_i,s)) \vee (\bigvee_j FD(a_j,s)) \quad (7)$$

$(g_1 \ldots g_n, a_1 \ldots a_m) \xrightarrow{OR} g$:

$$FD(g,s) \leftrightarrow (\bigwedge_i FD(g_i,s)) \wedge (\bigwedge_j FD(a_j,s)) \quad (8)$$

Axioms (9) to (12) 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 diagnosis purposes, to have a goal being both denied and satisfied at the same time. Conflict tolerance in [21] is not supported within our diagnosing framework.

$$g_1 \xrightarrow{++S} g_2 : \neg FD(g_1,s) \rightarrow \neg FD(g_2,s) \quad (9)$$

$$g_1 \xrightarrow{--S} g_2 : \neg FD(g_1,s) \rightarrow FD(g_2,s) \quad (10)$$

$$g_1 \xrightarrow{++D} g_2 : FD(g_1,s) \rightarrow FD(g_2,s) \quad (11)$$

$$g_1 \xrightarrow{--D} g_2 : FD(g_1,s) \rightarrow \neg FD(g_2,s) \quad (12)$$

The following propagation axiom is generated for the goal $g4$ in the SquirrelMail example, stating that $g4$ is denied if and only if at least one of its subtasks $a3$, $a4$, or $a5$ is denied:

$$FD(g4,s) \leftrightarrow FD(a3,s) \vee FD(a4,s) \vee FD(a5,s)$$

3.3.4 Basic Formulation for SAT

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

$$\Phi := \Phi_{\text{goal}} \wedge \Phi_{\text{deniability}} \wedge \Phi_{\text{LOG}}[\wedge \Phi_{\text{domain constraints}}] \quad (13)$$

The first component Φ_{goal} is the conjunction of relation axioms (7) to (12) which encode goal relations and forward and backward propagation axioms. The second component $\Phi_{\text{deniability}}$ is the conjunction of axioms (1) to (6) that encode denials of tasks and goals and explanation closure axioms. The third component Φ_{LOG} represents log data generated by monitors as specified in Definition 1. The last component, which is optional, $\Phi_{\text{domain constraints}}$, encodes any domain constraints and relations that are not represented in the goal graph.

3.3.5 Characterizing Diagnoses

Definition 2. *(**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, such that $D \cup \Phi$ is satisfiable.*

Theorem 1 establishes the soundness and completeness of our approach.

Theorem 1. *D is a diagnosis for a software system if and only if $D \cup \Phi$ is satisfiable.*

Proof. (**If:**) It follows straightforwardly from Definition 2 that if D is a diagnosis for a software system, $D \cup \Phi$ is satisfiable.

(**Only if:**) For the sake of contradiction, assume that D is not a diagnosis for a software system. It follows from the assumption that $D \cup \Phi$ is not satisfiable. We reach a contradiction because $D \cup \Phi$ is satisfiable. Therefore, D must be a diagnosis for the software system. □

According to the theorem, the diagnostic component finds a complete set of correct diagnoses defined in Definition 2, representing all the possible denied and satisfied goals and tasks, that can 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 2.

Definition 3. *(**Core Diagnosis**) A core diagnosis D for a software system is a set of FD and $\neg FD$ predicates over all the tasks in the goal graph such that $D \cup \Phi$ is satisfiable.*

Corollary 1. *Our diagnostic approach finds all the core diagnoses to the software system, as specified in 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. 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 component set* that contains individual task denial predicates that participate in core diagnoses. Task denial predicates that participate in the same *participating diagnostic component set* represent tasks that fail together. Any combinations of *participating diagnostic component sets* are possible core diagnoses. Therefore, instead of returning all core diagnoses that represent all the possible combinations of task denials, the diagnostic component returns all *participating diagnostic component sets* and leaves out their combinations.

DEFINITION 4. *(***Participating Diagnostic Component Set***) A participating diagnostic component set P for a software system is a set of FD predicates over tasks such that $P \cup \Phi$ is satisfiable.*

COROLLARY 2. *Our diagnostic approach finds all the* participating diagnostic component sets *to the software system, as specified in Theorem 1.*

4. IMPLEMENTATION

This section discusses the four main algorithms of our framework, namely two encoding algorithms (Algorithms 1 and 2) for encoding an annotated goal model into the propositional formula, Φ, and two diagnostic algorithms (Algorithms 3 and 4) for finding all core diagnoses and all *participating diagnostic component sets*, respectively.

The difference between the two encoding algorithms, Algorithms 1 and 2, is based on whether the algorithm preprocesses the log data when encoding the goal model into Φ. Algorithm 1 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 Φ with the size of a goal model. Algorithm 2 addresses this problem by generating all necessary axioms while keeping the growth of the size of Φ polynomial with respect to the size of the goal model. We present and compare experimental results using these two algorithms in Section 5.

Algorithm 1 Encode Φ Without Log Preprocessing

encode_Φ_without_log_preprocessing (goal model) {
  for each monitored task a
    for each $t_i \in$ [1, total timesteps] {
      $\Phi = \Phi \wedge$ *encodeTaskDenialAxiom*(a, t_i);
      $\Phi = \Phi \wedge$ *encodeTaskSessionDenialAxiom*(a, t_i); }
  for each monitored goal g
    for each $t_i \in$ [1, total timesteps]
      for each $t_j \in [t_i$, total timesteps] {
        $\Phi = \Phi \wedge$ *encodeGoalDenialAxiom*(g, t_i, t_j);
        $\Phi = \Phi \wedge$ *encodeGoalSessionDenialAxiom*(g, t_i, t_j); }
  for each fluent f
    for each $t_i \in$ [1, total timesteps]
      $\Phi = \Phi \wedge$ *encodeExplanationClosureAxiom*(f, t_i);
  return Φ; }

For each monitored task a in the goal model, Algorithm 1 generates a task denial axiom, and a task session denial axiom (axioms (1) and (3)) for all the timesteps during the execution session. These axioms cover all the possible task occurrence and denial timesteps. Similarly, for each monitored goal g, goal denial axiom and goal session denial axiom (axioms (2) and (4)) are generated for all possible combinations of timesteps t_i and t_j ($t_i \leq t_j$). These axioms cover all possible goal occurrence and denial timesteps. In addition, explanation closure axioms (axioms (5) and (6)) 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. The SAT solver input formula Φ is a conjunction of all the generated axioms. The size of Φ grows exponentially with the size of the goal model under Algorithm 1.

Algorithm 2 Encode Φ With Log Preprocessing

encode_Φ_with_log_preprocessing(goal model, log) {
  for each monitored task a {
    t_{occ} = task occurrence time during s
    $t_p = \max_{p(t)|p(t)\in\log}\{t_p \leq t_{occ_a}\}$
    $t_q = \min_{q(t)|q(t)\in\log}\{t_q > t_{occ_a}\}$
    $\Phi = \Phi \wedge$ *encodeTaskDenialAxiom*(a, t_p, t_{occ}, t_q)
    $\Phi = \Phi \wedge$ *encodeTaskDenialSessionAxiom*(a, t_q) }
  for each monitored goal g {
    $t_1 = \min_{occ_a(a,t)|occ_a(a,t)\in log \wedge a\in\text{descendents}(g)}\{t\}$
    $t_2 = \max_{occ_a(a,t)|occ_a(a,t)\in log \wedge a\in\text{descendents}(g)}\{t\}$
    $t_p = \max_{p(t)|p(t)\in\log}\{t_p \leq t_1\}$
    $t_q = \min_{q(t)|q(t)\in\log}\{t_q > t_2\}$
    $\Phi = \Phi \wedge$ *encodeGoalDenialAxiom*(g, t_p, t_1, t_2, t_q)
    $\Phi = \Phi \wedge$ *encodeGoalSessionDenialAxiom*(g, t_q) }
  return Φ; }

To address the scalability issue, for each monitored task a, Algorithm 2 finds in the log three timesteps: t_{occ}: a's occurrence timestep during the execution session s, t_p: 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 task denial axioms and task session denial axioms using these recorded timesteps. 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 monitored goal g, 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 goal denial axioms and goal session denial axioms, using the goal's occurrence, observed precondition and effect timesteps. Therefore, Algorithm 2 generates goal/task denial axioms only for the timesteps at which the goals/tasks actually occur as recorded in the log. As will be illustrated in Section 5, Algorithm 2 allows polynomial growth in the size of Φ with respect to the corresponding goal model, and allows the diagnostic component to scale to larger goal models.

Algorithm 3 finds all possible core diagnoses accounting for aberrant system behaviors recorded in the log. The algorithm calls Algorithm 2 to encode formula Φ using log preprocessing. If Φ is satisfiable, the algorithm decodes the

Algorithm 3 Find All Core Diagnoses

```
find_all_core_diagnoses() {
  Φ = encode_Φ_with_log_preprocessing(goal model, log);
  solver.solve(Φ);
  while (Φ is satisfiable) {
    μ = SAT result;
    //map SAT result to diagnostic instance
    oneDiagnosis = decodeToDiagnosis(μ)
    //filter the diagnosis to contain only denials of tasks)
    oneCoreDiagnosis = filter(oneDiagnosis)
    numberDiagnoses++;
    //add to Φ the negation of part of μ that encodes tasks
    Φ = Φ ∧ (¬μ_tasks);
    solver.solve(Φ); }}
```

solver result μ[2] into diagnostic instances that constitute a diagnosis. The diagnosis is then filtered into a core diagnosis that contains only FD predicates over tasks. To have the SAT solver search only on predicate symbols that encode the denial of tasks, the part of μ that corresponds to the denials of tasks is then negated and added back to Φ. The solver is invoked again to solve $\Phi \wedge \neg\mu_{tasks}$. When satisfied, a new μ 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 3 finds a complete set of core diagnoses, which is worst-case exponential in number to the goal graph size, and may not scale to large goal models.

Algorithm 4 Find All Participating Diagnostic Component sets

```
find_all_participating_diagnostic_component_sets() {
  Φ = encode_Φ_with_log_preprocessing(goal model, log);
  solver.solve(Φ);
  while (Φ is satisfiable) {
    μ = SAT result;
    oneCoreDiagnosis = filter(decodeToDiagnosis(μ))
    for each denial predicate of unmonitored task,
       FD(task_i), in oneCoreDiagnosis {
      oneDiagnosticSet =
       FD(task_i) ∧ ⋀_j FD(montitored task_j)
      numberDiagnosticSets++; }
    //add to the solver the negation of part of μ
    //that corresponds to denials of unmonitored tasks
    Φ = Φ ∧ (¬ μ_unmonitored tasks);
    solver.solve(Φ) ; }}
```

To address the scalability problem, Algorithm 4 returns all *participating diagnostic component sets* defined in Definition 4, instead of returning all core diagnoses. As with Algorithm 3, Algorithm 4 encodes the goal graph into Φ, and finds one core diagnosis to the system if Φ is satisfiable. Tasks can be denied in one of two ways, depending on if they are monitored. If a task is monitored, the truth values of its precondition and effect are known. Therefore, the task's satisfaction or denial is inferred through the task denial axiom (axiom (1)). Algorithm 4 adds the conjunction of the denial predicates for monitored tasks in the core diagnosis to the *participating diagnostic component set*. If a task is *not* monitored, its satisfaction or denial label can be propagated from its parent goal through the propagation axioms (axiom (7) and (8)). If a parent goal is denied, any combination of task denials for tasks under the goal's decomposition can account for the denial of the goal. Algorithm 4 infers individual task denials (i.e. participating diagnostic components) and leaves out their combinations. Therefore, the algorithm generates all *participating diagnostic component sets* by adding from the core diagnosis: (1) each individual task denial predicate for unmonitored tasks, and (2) the conjunction of task denial predicates for monitored tasks. The algorithm adds to Φ the negations of the part of μ that corresponds to the denial of unmonitored tasks. Hence, SAT solver searches only on predicate symbols that encode the denial of unmonitored tasks, and returns all *participating diagnostic component sets*. Under complete monitoring (i.e. task level monitoring), the two diagnosing algorithms conjecture the same set of task denial predicates since Algorithm 3 returns one core diagnosis under complete monitoring, which Algorithm 4 uses to parse into one *participating diagnostic component set*.

[2]Without loss of generality we treat the set as a conjunction of its elements.

5. EVALUATION OF OUR FRAMEWORK

We applied our framework to two medium-size public domain software systems to evaluate its correctness and performance: SquirrelMail [2], a Web-based email client, and an ATM (Automated Teller Machine) simulation [1]. 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 our solution can scale up to the goal model size and can be applied to industrial software applications with medium-sized requirements. All experiments reported were performed on a machine with a Pentium 4 CPU with 1 GB of RAM.

5.1 The SquirrelMail Running Example

The SquirrelMail log data (Section 3.2.1) contains two errors ($\neg$*webmail started(10)*, and occ_a*(a7, 11)*): (1) the effect of $g4$ (*webmail started*) was false, at timestep 10, after all the tasks under $g4$'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 encoding component preprocesses the log data as described in Algorithm 2. The diagnostic component infers that the goal $g4$ and the task $a7$ are denied during the execution session s. Then it infers that at least one of $g4$'s subtasks, $a3$, $a4$, $a5$, must have been denied to account for the denial of $g4$. Following Algorithm 4, the following three *participating diagnostic component sets* are returned:

Diagnostic Components Set 1: FD(a3); FD(a7)
Diagnostic Components Set 2: FD(a4); FD(a7)
Diagnostic Components Set 3: FD(a5); FD(a7)

5.2 Performance Evaluation with ATM

The ATM simulation case study is an illustration of OO design used in a software development class at Gordon College [1]. The application simulates an ATM performing customers' *withdraw*, *deposit*, *transfer* and *balance inquiry*

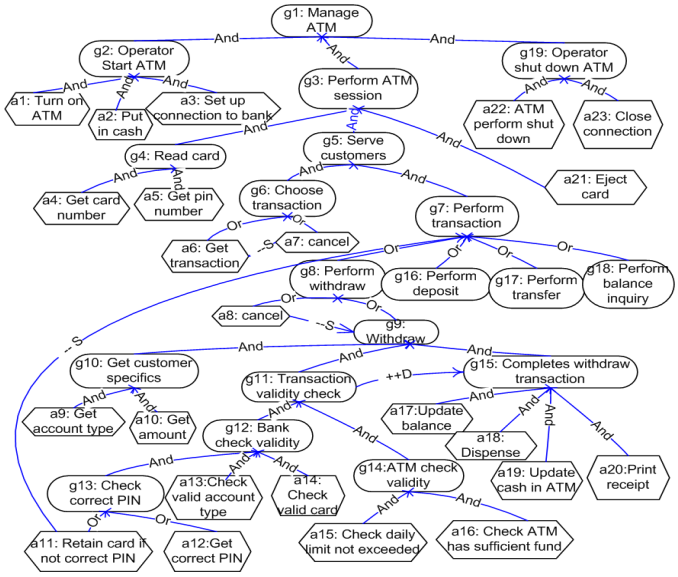

Figure 3: Partial ATM goal model

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 its partial goal graph with 19 goals and 23 tasks in Figure 3.

We report on two sets of experiments in this section. The first contains five experiments on the goal model shown in Figure 3, with increasing monitoring granularity. The goal graph is encoded in the SAT input formula Φ using the log preprocessing algorithm (Algorithm 2). We demonstrate and discuss the tradeoff between monitoring granularity and diagnostic precision. The second set reports 20 experiments on 20 progressively larger goal models that contain from 50 to 1000 goals and tasks. We obtain these larger goal models by cloning the ATM goal graph to itself. Furthermore, we performed this second set of experiments using both encoding algorithms 1 and 2 to compare their efficiency on larger goal graphs. In the both sets of experiments, the diagnostic component follows Algorithm 4 to return all *participating diagnostic component sets* for scalability.

The second set of experiments shows that our diagnostic framework scales to the size of the goal model, when the encoding is done *with* log file preprocessing (Algorithm 2), and when the diagnostic component returns all *participating diagnostic component sets* (Algorithm 4). As a result, our approach can be applied to industrial software applications with medium-sized requirement models.

Table 2 reports the results of the first set of experiments. We injected an error into the implementation of task $a17$, *update balance*, with the goal of pinning down a single precise participating diagnostic component set that contains $FD(a17)$. Column 1 in Table 2 lists the number of monitored goals/tasks in the goal graph. Column 2 lists the number of *participating diagnostic component sets* returned by the diagnostic component. Columns 3 and 4 give the total numbers of literals and clauses in the propositional formula, Φ, encoded for the SAT solver, using log preprocessing (Algorithm 2). Column 6 gives the total time (in seconds) that the diagnostic component took in finding all participating diagnostic component sets. T_{sum} is the sum of the time taken

Table 2: Tradeoff Between Monitoring Overhead and Diagnostic Precision (First Set of Experiments)

#Mon	#Diag	#Lit	#Clauses	T_{avg}(s)	T_{sum}(s)
1	20	62	62	0.051	1.032
3	15	67	72	0.058	0.876
6	11	72	82	0.071	0.781
8	4	76	92	0.132	0.531
11	1	79	107	0.392	0.392

to *encode* the goal graph into Φ (T_{encode}), and the time taken to find all diagnostic component sets, calculated by multiplying the time taken to find one diagnostic component set ($T_{diagnose}$) by the total number of returned diagnostic component sets ($\#Diag\ Set$) (Equation 14). $T_{diagnose}$ includes the time taken by the SAT solver to *solve* the propositional formula Φ (T_{solve}), and the time taken to *decode* the SAT result into one diagnostic component set (T_{decode}) (Equation 15). Column 5 lists the average time (T_{avg}) the solver took to find one participating diagnostic set (in seconds), calculated by diving T_{sum} by $\#Diag\ Set$ (Equation 16)

$$T_{sum} = T_{encode} + T_{diagnose} \times (\#Diag\ Set) \quad (14)$$

$$T_{diagnose} = T_{solve} + T_{decode} \quad (15)$$

$$T_{avg} = T_{sum}/(\#Diag\ Set) \quad (16)$$

In the first experiment (row 1 in Table 2), we monitored only the root goal $g1$ (highest level of monitoring granularity). The diagnostic component inferred that $g1$ was denied and at least one of the executed tasks under $g1$'s decomposition must have been denied to account for the denial of $g1$. A total of 20 participating diagnostic component sets were returned (column 2). The diagnostic framework took 1.032 seconds to find all diagnostic component sets, which averages to 0.051 second per diagnosis.

In experiments 2 to 5 (rows 2 to 5 in Table 2), 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 component sets 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 set increasing from 0.058 to 0.392 seconds. It's interesting to note that, even with this increase, the total amount of time the solver took to find *all* participating diagnostic component sets decreased from 1.032 to 0.392 seconds. This happened because the total number of core diagnoses decreased from 20 to 1.

This first set of experiments showed that the number of participating diagnostic component sets 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 set all increase. The benefit of monitoring at a high monitoring granularity is that we are able to infer fewer diagnostic component sets identifying a smaller set of possible faulty components. It is also noteworthy that the total amount of time taken to find all diagnostic component sets may not increase despite the fact that it takes longer to find one diagnostic component set. The reverse is true when monitoring granularity decreases: we have less monitoring and diagnostic overhead, but the number of participating diagnostic component sets increases if the system is behaving abnormally. However, if the system is running correctly

Table 3: Scalability to Goal Model Size with Log Preprocessing (Second Set of Experiments)

Goal Model Size	T_{sum} (s)	T_{encode} (s)	$T_{diagnose}$ (s)	#Lit	#Clauses
50	0.453	0.047	0.406	81	202
100	0.641	0.079	0.562	157	392
150	0.829	0.141	0.688	233	582
200	0.999	0.187	0.812	309	772
250	1.414	0.219	0.922	385	962
300	1.250	0.266	0.984	461	1152
350	1.390	0.343	1.047	537	1342
400	1.453	0.375	1.078	613	1532
450	1.593	0.437	1.156	689	1722
500	1.766	0.500	1.266	765	1912
550	1.906	0.578	1.328	841	2102
600	2.094	0.656	1.438	917	2292
650	2.266	0.797	1.469	993	2482
700	2.547	0.922	1.625	1069	2672
750	2.657	1.032	1.625	1145	2862
800	2.860	1.094	1.766	1221	3052
850	3.093	1.281	1.812	1297	3242
900	3.251	1.360	1.891	1373	3432
950	3.562	1.562	2.000	1449	3662
1000	3.750	1.672	2.078	1525	3812

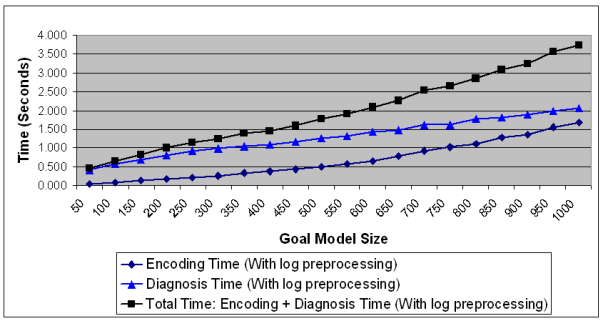

Figure 4: Scalability to Goal Model Size (Encoding with Log preprocessing)

and no requirements are denied, no diagnostic component set will be returned, so minimal monitoring is advisable.

Table 3 reports the results of the second set of experiments, performed with the log file preprocessing algorithm (Algorithm 2). 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 obtained these larger goal graphs by cloning the ATM goal graph (Figure 3) to itself. All the experiments are performed with complete (task level) monitoring, therefore, one diagnostic component set is returned for each experiment. Column 1 in Table 3 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 Φ with log file preprocessing. Column 4, $T_{diagnose}$ lists the time taken by the SAT solver to solve Φ and the time taken to decode the SAT result into a diagnostic component set. Column 2, T_{sum}, calculated by adding T_{encode} and $T_{diagnose}$, represents the total time taken (in seconds) for finding the diagnostic component set. The total numbers of literals and clauses in Φ are listed in Columns 5 and 6.

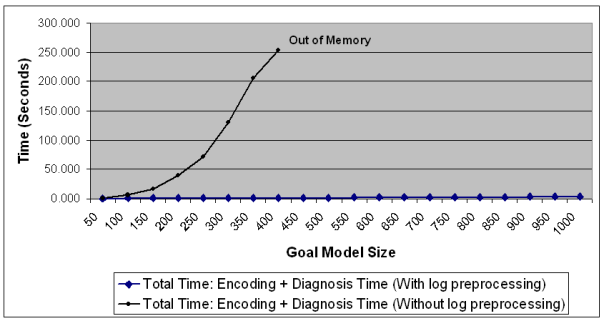

Figure 5: Scalability Comparison of Encoding With and Without Log Preprocessing

Figure 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 3) and the goal model size (the x-axis - the values of column 1 of Table 3). The three curves in Figure 4 shows that the diagnostic component scales to the size of the goal model when following algorithms 2 and 4, and our approach can be applied to industrial software applications with medium-sized requirement graphs.

To compare the efficiency between the two encoding Algorithms 1 and 2, we performed this second set of experiments using also the Algorithm 1, encoding *without* log file preprocessing. Figure 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. As discussed in Section 4, encoding *without* log preprocessing gives exponential growth in the size of Φ with respect to the size of the goal model; an "out of memory" error was returned 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 goal models with log file preprocessing, and when all *participating diagnostic component sets* (instead of all diagnoses) are conjectured. As a result, we contend that our framework can be applied to industrial software applications with medium-sized goal graphs.

6. RELATED WORK

6.1 Requirements Monitoring Systems

Requirement monitoring aims to track a system's runtime behavior so as to detect deviations from its requirement specification. Fickas' and Feather's work [6, 7] presents a run-time technique for monitoring requirements satisfaction. This technique identifies requirements, assumptions and remedies. If an assumption is violated, the associated requirement is denied, and the associated remedies are executed. The approach uses Formal Language for Expressing Assumptions (FLEA) to monitor and alert the user of any requirement violations at runtime. Reconciliation to system requirements generally involves either parameter tuning or switching to an alternative design. The main difference between our work and FLEA is that their proposal focuses on monitoring for changes in the domain, rather than malfunctions of the system. Moreover, there is no need for diagnostic reasoning in FLEA because the framework predefines

requirement/assumption/remedy tuples.

Robinson has also presented a requirement monitoring framework, ReqMon in [19]. Requirements are specified using KAOS [3, 22], and negations of a system's requirements are monitored. If an observed event is a satisfaction (or violation) event, satisfaction (or denial) status is updated for the requirement. The main difference between our research and Robinson's is that in ReqMon diagnostic formulae are generated manually using obstacle analysis [11], whereas in our work we make assumptions about what can fail and as a result, we can automatically infer diagnoses given a model of system requirements and log data. Moreover, in ReqMon the term "adaptive" refers to the ability of a monitor to take some actions after receiving an event. In our work, adaptive monitoring refers to the ability to tune monitoring granularity at runtime in response to diagnostic feedback.

More recently, Winbladh et al. [23] presented a goal-driven specification based testing prototype that aims to find mismatches between actual and expected system behaviors. This work is able to identify false positives (where the program achieves the correct results by following incorrect processes) and detect domain errors. (where the program follows correct processes but fails to achieve the correct result). To accomplish this, their monitoring component monitors software systems at the finest (i.e. leaf) level of monitoring granularity. The satisfaction of higher-level goals is inferred from the satisfaction of their leaf level functional subgoals. As a consequence, Winbladh's proposal may not scale to industrial sized applications. Our proposed monitoring component is able to monitor requirement satisfaction at different levels of granularity, which can be tuned dynamically as required. Furthermore, our diagnostic component is able to infer diagnoses with incomplete logging data, powered by existing SAT solvers.

None of the research discussed above [6, 7, 23] presented framework performance evaluations or discussed scalability issues. It is therefore difficult to compare our respective approaches in terms of performance and scalability.

6.2 AI Theories of Diagnosis

In this work, we have adopted theories of diagnosis from AI [17, 5, 13, 10]. Early AI research on diagnosis focused on static systems, determining which components of the system were behaving normally and which were behaving abnormally. Two widely accepted AI definitions of diagnosis are consistency-based diagnosis [17, 5], and abductive explanation [5]. Consistency based diagnosis looks for a set of abnormal components that is *consistent* with the union of system description, system input settings, and system observations. Abductive diagnosis looks for a minimal conjunction of abnormal and normal components, such that its union with system description and system input settings *entails* system observations.

Diagnosing dynamic systems has received more attention recently. McIlraith added a theory of action to traditional AI model-based diagnosis [17, 5] and proposed *explanatory diagnosis* [13]. Explanatory diagnosis conjectures a sequence of actions that lead to the system's aberrant behavior. McIlraith showed that conjecturing an explanatory diagnosis is analogous to AI planning. Iwan [10] further extended McIlraith's work and proposed *history based explanatory diagnosis* (in which the basic action theory was extended to take into account the possibility that some actions may not occur when they should, or occurred but did not achieve their intended effects).

We extend McIlraith's and Iwan's work in several important ways. The distinguishing feature of our approach is its ability to assess satisfaction of the system's requirements, goals as well as to diagnose atomic actions. This ability to diagnose at different levels of granularity is afforded by the richness and hierarchical structure of goal models. Moreover, the purpose of our diagnoses is to pin down which tasks have failed, whereas in McIlraith's and Iwan's work, the purpose of diagnosis is to find a sequence of actions that can account for aberrant system behaviors.

6.3 SAT-Based Goal Analysis

In [21] a SAT based qualitative framework is proposed for finding assignments of satisfaction/denial labels for a set of input goals that satisfies desired satisfaction/denial labels for a set of target goals using two SAT solvers. We adopted the goal model formalism used in [21], and extended it by associating with goals and actions their monitoring switches, preconditions, effects and occurrences. We concern ourselves with goal and task denial using AI theories of diagnosis. Goal/task denial is then propagated along the goal graph using SAT solvers. In contrast, the focuses of [21] is on satisfaction/denial label propagation only. In so far as satisfiability and deniably are two sides of the same coin, our work is in line with [21] with respect to label propagation.

7. CONCLUSION

This paper makes contributions to software requirement monitoring and diagnosis, by adapting AI theories of action and diagnosis. The aim of this research is to develop a tool-supported methodology for designing and running monitoring and diagnostic components in software-intensive systems. To the best of our knowledge, our proposed framework is the first SAT-based solution to the diagnosis of the software requirement satisfaction problem that is sound and complete.

Our framework has been evaluated with two medium-sized case studies, the results of which demonstrates the feasibility of scaling our approach to medium-size goal models, and can therefore be applied to industrial software applications. As future work, we plan to further evaluate our framework on large scale, industrial size applications. We also plan to design tool support for automating additional tasks, such as the generation of non-intrusive requirement monitors and the dynamic tuning of monitoring granularity. Finally, we plan to device monitors using statistical data to help identify recurring patterns of system failures.

8. REFERENCES

[1] R. Bjork. An example of object-oriented design: an atm simulation. http://www.cs.gordon.edu/courses/cs211/atmexample/index.html/.

[2] R. Castello. Squirrel mail. http://www.squirrelmail.org/.

[3] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[4] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Journal of ACM*, 5:394–397, 1962.

[5] J. De Kleer, A. K. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56(2-3):197–222, 1992.

[6] M. S. Feather, S. Fickas, A. Van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *IWSSD'98*, 1998.

[7] S. Fickas and M. Feather. Requirements monitoring in dynamic environments. In *RE'95*, 1995.

[8] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani. Reasoning with goal models. In *ER*, pages 167–181. Springer, 2002.

[9] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat-solver. In *DATE*, pages 142–149, 2002.

[10] G. Iwan. History-based diagnosis templates in the framework of the situation calculus. *AI Communications*, 15:31–45, 2002.

[11] A. Van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering*, 26:978–1005, 2000.

[12] D. Le Berre. A satisfiability library for java. http://www.sat4j.org/.

[13] S. McIlraith. Explanatory diagnosis: Conjecturing actions to explain observations. In *KR'98*, pages 167–179, 1998.

[14] S. McIlraith and R. Fadel. Planning with complex actions. In *NMR2002*, pages 356–364, 2002.

[15] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *Design automation*, pages 530–535. ACM Press New York, NY, USA, 2001.

[16] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: a process-oriented approach. *IEEE Trans. on Softw. Eng.*, 18(6):483–497, 1992.

[17] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[18] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. *Artificial Intelligence and Mathematical Theory of Computation*, pages 359–380, 1991.

[19] W. N. Robinson. Implementing rule-based monitors within a framework for continuous requirements monitoring. In *HICSS'05*, 2005.

[20] L. Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.

[21] R. Sebastiani, P. Giorgini, and J. Mylopoulos. Simple and minimum-cost satisfiability for goal models. In *CAiSE'04*, volume 4, pages 20–33. Springer, 2004.

[22] A. van Lamsweerde, R. Darimont, and P. Massonet. Goal-directed elaboration of requirements for a meeting scheduler: Problems and lessons learnt. In *RE'95*, page 194, 1995.

[23] K. Winbladh, T. A. Alspaugh, H. Ziv, and D. J. Richardson. An automated approach for goal-driven, specification-based testing. *ASE'06*, 2006.

[24] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. C. S. do Prado Leite. Reverse engineering goal models from legacy code. In *RE'05*, pages 363–372, 2005.