Autom Softw Eng
DOI 10.1007/s10515-008-0042-8

Monitoring and diagnosing software requirements

Yiqiao Wang · Sheila A. McIlraith · Yijun Yu · John Mylopoulos

Received: 29 September 2008 / Accepted: 16 October 2008

Abstract We propose a framework adapted from Artificial Intelligence theories of action and diagnosis for monitoring and diagnosing failures of software requirements. Software requirements are specified using goal models where they are associated with preconditions and postconditions. The monitoring component generates log data that contains the truth values of specified pre/post-conditions, as well as system action executions. Such data can be generated at different levels of granularity, depending on diagnostic feedback. The diagnostic component diagnoses the denial of requirements using the log data, 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. The framework returns 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 scalability of our approach.

Keywords Requirement monitoring · Diagnostics

Y. Wang (✉) · S.A. McIlraith · J. Mylopoulos
Department of Computer Science, University of Toronto, Toronto, Canada
e-mail: yw@cs.toronto.edu

S.A. McIlraith
e-mail: sheila@cs.toronto.edu

J. Mylopoulos
e-mail: jm@cs.toronto.edu

Y. Yu
Computing Department, Faculty of Maths & Computing, The Open Univ., Walton Hall, Milton Keynes, UK
e-mail: y.yu@open.ac.uk

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 Software Engineering (hereafter SE) literature. This paper presents a monitoring and diagnosis framework with an adaptive monitoring component and a SAT-based diagnostic component 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 Yu et al. (2005), or provided by requirements analysts. In addition, we assume that traceability links are provided, linking source code and requirements in both directions.

In Wang et al. (2007), we presented a monitoring component that monitors requirements and generates log data at different levels of granularity. The monitoring component is based on AspectJ technologies (Kiczales et al. 2001). Monitoring granularity can be tuned adaptively depending on diagnostic feedback. The diagnosing component of our framework analyzes generated log data and identifies failures corresponding to aberrant system behaviors that lead to the violation of system requirements. When failures are found, the diagnostic component identifies root causes.

The propositional satisfiability (SAT) problem is concerned with determining whether there exists a truth assignment to variables of a propositional formula that makes the formula true. In Wang et al. (2007), we transformed 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.

This paper presents a complete account of our proposed monitoring and diagnosis framework, along with extensions and improvements. In particular, (1), we propose the concept of multi-layered monitoring and diagnosis to extend the applicability of our framework to multi-layered socio-technical systems, such as systems that have adopted service-oriented architectures (SOA); (2) We update and extend one of the diagnostic algorithms (Algorithm 4); (3), we discuss optimizations of our algorithms and their implementation that make the framework scalable.

We illustrate and evaluate our framework on two medium-sized publicly available case studies: Squirrel Mail, a Web-based email client (Castello 2007), and an ATM simulation (Bjork 2007). These case studies demonstrate the feasibility of scaling our approach to software systems with medium to large requirement models.

2 Preliminaries

2.1 Goal models

Requirements Engineering (RE) is a branch of SE that deals with the elicitation and analysis of system requirements. In recent years, goal models have been used in RE to model and analyze stakeholder objectives (Dardenne et al. 1993). Software systems' functional requirements are represented as hard goals, while their non-functional requirements are represented as soft goals (Mylopoulos et al. 1992). 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 Giorgini et al. (2002), 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 meaning 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. A $++$ link (respectively a $--$ link) represents both $++S$ and $++D$ (respectively $--S$ and $--D$) relationships. These $++$ and $--$ links represent the strong MAKE($++$) and BREAK($--$) contributions between hard goals/tasks.

In this paper, 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/task satisfaction and denial. These weaker links may proceed from hard goals/tasks to soft goals. The class of goal models used in our work has been formalized in Giorgini et al. (2002), 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 are boolean flags that can be switched on/off to indicate whether the corresponding goal/task is to be monitored.

We use the SquirrelMail (Castello 2007) 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 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 $g1$ (*send email*) is AND-decomposed into task $a1$ (*load login form*), goal $g2$ (*process send mail request*), and task $a7$ (*send message*). Goal $g1$ is satisfied if and only if $a1$, $g2$, and $a7$ are satisfied. $G2$ is OR decomposed into task $a6$ (*report IMAP error*) if the email IMAP server is not found and goal $g3$ (*get compose page*) if otherwise. $G2$ is satisfied if and only if either $g3$ or $a6$ is satisfied. A BREAK ($--$) contribution link proceeds between goal $g3$ and task $a6$, meaning that $g3$ is satisfied/denied if and only if $a6$ is denied/satisfied respectively. The satisfaction of

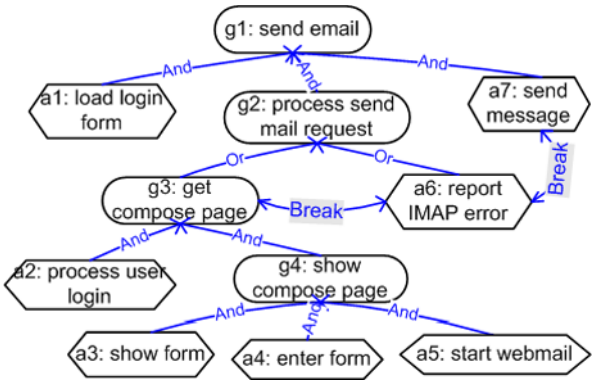

Fig. 1 Squirrel mail goal model

task $a6$ also "breaks" the satisfaction of task $a7$: $a7$ is denied/satisfied if and only if $a6$ is satisfied/denied respectively. $G3$ is decomposed into task $a2$ (*process user login*), and goal $g4$ (*show compose page*), which is further AND-decomposed into three tasks: $a3$ (show form), $a4$ (*enter form*), and $a5$ (*start webmail*).

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, identifying the satisfying assignments of variables.

The earliest and most prominent SAT algorithm is DPLL (Davis-Putnam-Logemann-Loveland) (Davis et al. 1962), which uses backtracking search. Even though the SAT problem is inherently intractable, there have been many improvements to SAT algorithms in recent years. Chaff (Moskewicz et al. 2001), Berk-Min (Goldberg and Novikov 2002) and Siege (Ryan 2004) are among the fastest SAT solvers available today. For our work, we use SAT4J (Le Berre 2007), an efficient SAT solver that inherits a number of features from Chaff.

3 Our framework

3.1 Overview

Figure 2 provides an overview of our 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 Yu et al. (2005) or it can be modeled by requirement analysts. Requirement analysts annotate the goal model with monitoring switches, and 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

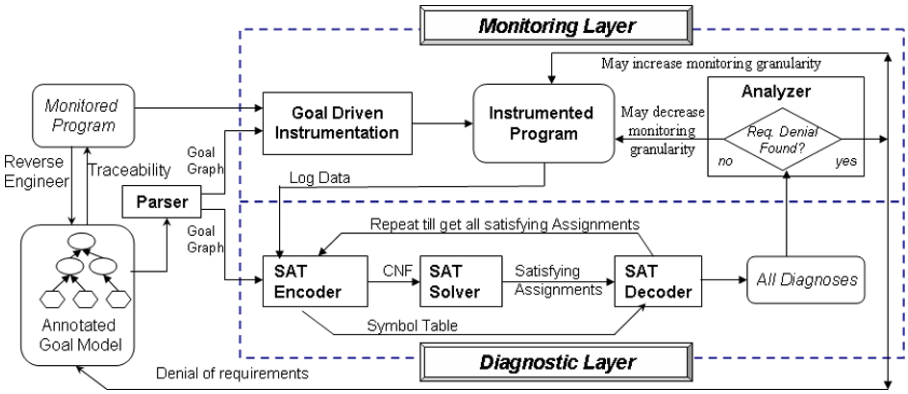

Fig. 2 Framework overview

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 using AspectJ (Zhou 2008). 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. In our framework a diagnosis specifies for each goal and task whether it is fully denied or not. 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 *analyzer* analyzes the returned diagnoses, and checks if a denial of system requirements is found. If denials of system requirements are found, they are traced back to the source code to identify the problematic components. The diagnosis analyzer may then 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 space with added constraints. If no system requirements are denied, 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. All the above described components have been implemented.

3.2 Monitoring and diagnosis

Satisfaction of software system requirements can be monitored at different levels of granularity. 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

Table 1 Squirrel mail annotated goal model

Goal/ task	Monitor switch	Precondition	Effect
$a1$	On	URL entered	Correct form loaded
$a2$	On	¬wrongIMAP ∧ correct form loaded	Correct key entered
$a3$	Off	Correct key entered	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 loaded ∨ wrongIMAP	Webmail started ∨ error reported
$g3$	Off	Correct form loaded ∧ ¬wrongIMAP	Webmail started
$g4$	On	Correct key entered	Webmail started

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.

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 McIlraith and Fadel (2002).

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 this paper. We assume that both the goal model and its associated preconditions and effects are correctly specified for the application.

Table 1 lists the details of each goal/task in the SquirrelMail goal model (Fig. 1) with its monitoring switch status (column 2), and associated precondition and effect (columns 3 and 4). In this example, the satisfaction of goal $g4$, and tasks $a1$, $a2$, $a6$, and $a7$ are monitored.

In the log data, 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. The following is an example of log data from the SquirrelMail case study:

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

The log data 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 diagnostic component analyzes the log data and infers that the goal $g4$ and the task $a7$ are denied.

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. For the SquirrelMail case study, the diagnostic component infers $FD(g4, s)$ and $FD(a7, s)$ as diagnoses.

4 Formal foundations

This section presents the formal foundations of our framework. The axiomatizations generated for diagnostic reasoning (presented in Sects. 4.3 and 4.4) are adaptations of the theoretical diagnostic framework proposed in Reiter (1987); De Kleer et al. (1992); McIlraith (1998).

4.1 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:

$$\Phi := \Phi_{LOG} \wedge \Phi_{deniability} \wedge \Phi_{goal}[\wedge \Phi_{domain\ constraints}] \tag{1}$$

The first component Φ_{LOG} represents log data generated by monitors as specified in Definition 1 (Sect. 4.2). The second component $\Phi_{deniability}$ encodes denials of tasks

and goals (Sects. 4.3 and 4.4). The third component Φ_{goal} encodes goal relations and forward and backward propagation (Sect. 4.5). The last component, which is optional, $\Phi_{domain\ constraints}$, encodes any domain constraints and relations that are not represented in the goal graph.

4.2 Log data

Log data consists of truth values of observed domain literals and the occurrences of tasks, each associated with a specific timestep t. A log is made of a sequence of log instances.

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 was true at timestep 1, and task a occurred at timestep 2, their respective log instances are: $l(1)$ and $occ_a(a, 2)$.

4.3 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 t; or (2) if the task occurred at timestep t, but its effect is false at the subsequent timestep $t+1$. Axiom (2) 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.

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 $t+1$.*

$$FD(a, t+1) \leftrightarrow occ_a(a, t) \wedge (\neg p(t) \vee \neg q(t+1)) \tag{2}$$

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,*

[1] In many axiomatizations it is assumed that $occ_a(a, t) \rightarrow p(t)$, where p is the precondition of a.

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 $t_2 + 1$.

$$FD(g, t_2 + 1) \leftrightarrow occ_g(g, t_1, t_2) \wedge (\neg p(t_1) \vee \neg q(t_2 + 1)) \wedge (t_1 \leq t_2) \tag{3}$$

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.*

$$FD(a, t) \rightarrow FD(a, s) \tag{4}$$

$$FD(g, t) \rightarrow FD(g, s) \tag{5}$$

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:

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

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

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

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

4.4 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 (Reiter 1991) 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 (6) and (7) 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 ($i \in [1, \ldots, n]$ and $j \in [1, \ldots, m]$),

$$\begin{aligned} &\neg f(t) \wedge f(t+1) \\ &\quad \leftrightarrow \bigvee_i (occ_a(a_i, t) \wedge \neg FD(a_i, t+1)) \\ &\qquad \vee \bigvee_j (occ_g(g_j, t_1, t_2) \wedge \neg FD(g_j, t_2+1) \wedge (t_1 \leq t \leq t_2)) \end{aligned} \tag{6}$$

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

$$\begin{aligned} &f(t) \wedge \neg f(t+1) \\ &\quad \leftrightarrow \bigvee_i (occ_a(a_i, t) \wedge \neg FD(a_i, t+1)) \\ &\qquad \vee \bigvee_j (occ_g(g_j, t_1, t_2) \wedge \neg FD(g_j, t_2+1) \wedge (t_1 \leq t \leq t_2)) \end{aligned} \tag{7}$$

For example, 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:

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

The conjunction of axioms (2) to (7) encodes the $\Phi_{deniability}$ component of the propositional formula Φ (equation (1)), and they represent goal and task denial relations.

4.5 Axiomatization of the goal model

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

$$(g_1 \ldots g_n, a_1 \ldots a_m) \xrightarrow{AND} g:$$
$$FD(g,s) \leftrightarrow \left(\bigvee_i FD(g_i,s)\right) \vee \left(\bigvee_j FD(a_j,s)\right) \tag{8}$$

$$(g_1 \ldots g_n, a_1 \ldots a_m) \xrightarrow{OR} g:$$
$$FD(g,s) \leftrightarrow \left(\bigwedge_i FD(g_i,s)\right) \wedge \left(\bigwedge_j FD(a_j,s)\right) \tag{9}$$

Axioms (10) to (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 Sebastiani et al. (2004) is not supported within our diagnostic framework.

$$g_1 \xrightarrow{++S} g_2 : \neg FD(g_1,s) \rightarrow \neg FD(g_2,s) \tag{10}$$
$$g_1 \xrightarrow{--S} g_2 : \neg FD(g_1,s) \rightarrow FD(g_2,s) \tag{11}$$
$$g_1 \xrightarrow{++D} g_2 : FD(g_1,s) \rightarrow FD(g_2,s) \tag{12}$$
$$g_1 \xrightarrow{--D} g_2 : FD(g_1,s) \rightarrow \neg FD(g_2,s) \tag{13}$$

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:

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

The conjunction of axioms (8) to (13) encodes the Φ_{goal} component of the propositional formula Φ (equation (1)). These axioms represent the AND/OR decompositions and contribution links in the goal model.

4.6 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, 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 Φ is satisfiable, and D is extracted from a satisfying assignment.*

Theorem 1 follows directly from Definition 2, and it establishes the soundness and completeness of our diagnostic approach (presented in Sect. 5). 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. 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 3 (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 4 (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.

5 Algorithms

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 components*, respectively.

The difference between the two encoding algorithms, Algorithms 1 and 2, lies in 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

Algorithm 1 Encode Φ without Log Preprocessing

```
encode_Φ_without_log_preprocessing (goal_model, total_timesteps) {
  for each task a {
    //encode denial axioms
    if (a is associated with p and q) {
      for each t_i ∈ [1, total_timesteps] {
        Φ = Φ ∧ encodeTaskDenialAxiom(a, t_i);
        Φ = Φ ∧ encodeTaskSessionDenialAxiom(a, t_i);}}}
  for each goal g {
    //encode denial axioms
    if (g is associated with p and q) {
      for each t_i ∈ [1, total_timesteps]
        for each t_j ∈ [t_i, total_timesteps] {
          Φ = Φ ∧ encodeGoalDenialAxiom(g, t_i, t_j);
          Φ = Φ ∧ encodeGoalSessionDenialAxiom(g, t_i, t_j);}}
    //encode goal model structure
    for each t_i ∈ [1, total_timesteps]
      Φ = Φ ∧ encodeLabelPropagation (goal_model, t_i)}
  for each fluent f {
    for each t_i ∈ [1, total_timesteps]
      Φ = Φ ∧ encodeExplanationClosureAxiom(f, t_i);}
  for each contribution link l {
    for each t_i ∈ [1, total_timesteps]
      Φ = Φ ∧ encodeContributionLink(l, t_i)}
  return Φ;}
```

Algorithm 2 Encode Φ with Log preprocessing

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
      $t_p = \max_t\{t \leq t_{occ_a}\ and\ p(t) \in \log\}$
      $t_q = \min_t\{t > t_{occ_a}\ and\ q(t) \in \log\}$
      if ($t_p \leq t_{occ_a} < t_q$) {
        $\Phi = \Phi \wedge FD(a, s) \leftrightarrow occ_a(a, t_{occ_a}) \wedge (\neg p_{t_p} \vee \neg q_{t_q})$
  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 $t2$
      $t_1 = \min_t\{occ_a(a, t) \in \log\ and\ a \in \text{descendents}(g)\}$
      $t_2 = \max_t\{occ_a(a, t) \in \log\ and\ a \in \text{descendents}(g)\}$
      $t_p = \max_t\{t \leq t1\ and\ p(t) \in \log\}$
      $t_q = \min_t\{t > t2\ and\ q(t) \in \log\}$
      if ($t_p \leq t1 \leq t2 < t_q$) {
        $\Phi = \Phi \wedge FD(g, s) \leftrightarrow occ_g(g, t_1, t_2) \wedge (\neg p_{t_p} \vee \neg q_{t_q})$
    //encode goal model structure
    $\Phi = \Phi \wedge$ encodeLabelPropagation (goal model, s)}
  for each contribution link l {
    $\Phi = \Phi \wedge encodeContributionLink(l, s)$}
  return Φ;}

model. We present and compare experimental results using these two algorithms in Sect. 7.

For each task a in the goal model that is associated with a precondition p and an effect q, Algorithm 1 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 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 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.

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 2 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 Φ an axiom stating that if a occurred at timestep t_{occ_a} 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 Φ 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 2 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 Sect. 7, 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.

Theorem 2 *Let Φ' be the Φ computed by Algorithm 2. 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 2.

Algorithm 3 finds all possible core diagnoses accounting for aberrant system behaviors recorded in the log. If the input formula Φ is satisfiable, the algorithm decodes the solver result μ[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

Algorithm 3 Find All Core Diagnoses

```
find_all_core_diagnoses(Φ) {
  while (Φ is satisfiable) {
    μ = satisfying assignments for all variables in Φ
    //map μ to diagnostic instance
    oneDiagnosis = decodeToDiagnosis(μ)
    //obtain a new oneCoreDiagnosis containing session
    //level task satisfaction and denial predicates
    oneCoreDiagnosis =
        session level task satisfactions and denials in oneDiagnosis
    //add to Φ the negation of both session level and timestep
    //level task denials and satisfactions in oneDiagnosis
    Φ = Φ ∧ ¬μ_(task denials and satisfactions in oneDiagnosis)
}}
```

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

Algorithm 4 Find All Participating Diagnostic Components

```
find_all_participating_diagnostic_components(Φ) {
  while (Φ is satisfiable) {
    μ = satisfying assignments for all variables in Φ
    //map SAT result to diagnostic instance
    oneDiagnosis = decodeToDiagnosis(μ)
    //complete failure configuration containing session
    //level task satisfactions and denials in oneDiagnosis
    oneCoreDiag =
      session level task satisfactions and denials in oneDiagnosis
    //partial failure configuration containing only task denials
    partialCoreDiag = task denials in oneCoreDiag
    allCoreDiag = allCoreDiag ∧ oneCoreDiag
    //calculate which part of μ to add back to the SAT solver
    boolean negateCompleteConfig = false;
    for (each FD(task_i, s) in oneCoreDiag)
      if (task_i is associated with a contribution link) and
        if (FD(task_i, s) is not already part of a core diagnosis)
          negateCompleteConfig = true;
    if (negateCompleteConfig)
      //add to Φ the negation of complete failure configuration
      Φ = Φ ∧ ¬μ_oneCoreDiag;
    else //add to Φ the negation of partial failure configuration
      Φ = Φ ∧ ¬μ_partialCoreDiag;}
  AllParticipatingComps = filter(allCoreDiag);}
```

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 μ that encodes task denials and satisfactions (both at the session level and at the timestep level) in oneDiagnosis is negated and added back to Φ. The solver is invoked again to solve the new Φ. 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, 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 4 returns all *participating diagnostic components* defined in Definition 4, instead of all core diagnoses. As with Algorithm 3, Algorithm 4 decodes the solver result μ into a core diagnosis if Φ is satisfiable. And as with Algorithm 3, Algorithm 4 negates part of μ and adds it back to Φ, to have the SAT solver return another satisfying truth assignment, until Φ becomes unsatisfiable. However, the key difference between the two algorithms lies in which part of μ the algorithm negates and adds back to the solver. Algorithm 3 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 4 negates and adds back to the solver ei-

ther task denials only, or both task denials and satisfactions, depending weather any denied task is associated with contribution links. Consequently, under Algorithm 4 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 *negateCompleteConfig* is set to true. When this happens, the algorithm negates and adds to Φ the complete session level task failure configuration in μ (corresponding to both task denials and satisfactions, i.e., the *oneCoreDiagnosis*). Negating and adding to the solver the complete failure configuration in μ, 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 Φ becomes unsatisfiable because of these contradictions before all the participating diagnostic components can be found. On the other hand, if the boolean flag *negateCompleteConfig* is false, all the denied tasks in μ 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 μ that encodes task denials, guiding the solver to move on quickly to other denied tasks.

The SAT solver solves the new Φ and returns another μ, which is decoded to another core diagnosis, if Φ is satisfiable. This process repeats till Φ 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 4 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 4 outperforms Algorithm 3 because Algorithm 3 finds all core diagnoses, whereas Algorithm 4 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 3 instead of Algorithm 4.

6 Implementation

6.1 General considerations

The presented monitoring and diagnostic framework has been implemented using the Java programming language. The source code contains 11 Java classes with about 5000 LOC. We use SAT4J (Le Berre 2007), an efficient SAT solver, by including its jar file in the framework's compile path.

Monitoring specifications and instrumentation code can be generated semi-automatically using AspectJ (Kiczales et al. 2001). Aspect Oriented Programming (AOP) is a technology that implements crosscutting concerns, such as logging, in modular units. AspectJ is an aspect oriented extension to Java. The *Goal Driven Instrumentation* component (in Fig. 2) obtains monitoring information such as "what" to monitor (from the goal model), and "where" the monitors should be inserted (from the traceability links). Using this information, the instrumentation component can semi-automatically generate monitoring specifications in AspectJ terminologies. The AspectJ compiler can then use these specifications to automatically instrument the software at its byte code level. Interested readers can refer to Zhou (2008) for a complete account of the monitoring component.

6.2 System 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 2 and 4, instead of Algorithms 1 and 3, enhance scalability. In this section, we discuss how Algorithm 4 works differently from Algorithm 3.

The performance comparison of Algorithms 3 and 4 is straightforward when no MAKE/BREAK contribution links are present. Algorithm 3 finds all core diagnoses corresponding to possible combinations of task denials. Algorithm 4 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 3 returns up to 2^n core diagnoses. Algorithm 4 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 2 that compare the efficiency of the two algorithms when contribution links are present. All 8 experiments use the same goal model with 27 tasks and 23 goals. The 8 experiments feather different numbers and types of contribution links between the tasks in the goal model. For each experiment, the program

Table 2 Optimization of Algorithm 4 over Algorithm 3

#Contr links	*#Core*	*Time$_C$* (s)	*#Core to obtain all PDC*	*#PDC*	*Time$_{PDC}$* (s)	*%Improv*
0	n/f	n/f	27	27	1.391	≈100%
1	n/f	n/f	42	27	2.047	≈100%
10	n/f	n/f	53	27	2.782	≈100%
15	4096	7318.00 (>2 h)	91	27	5.219	97.78%
20	299	62.40	18	27	1.276	94.08%
22	128	17.02	17	27	1.286	86.46%
25	107	15.28	17	27	1.307	84.06%
27	16	1.38	10	27	0.953	37.50%

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 3. 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 4. Column 4 (*#Core to Obtain All PDC*) lists the total number of core diagnoses returned. Column 5 (*#PDC*) lists the number of *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 4 over Algorithm 3. Observe that Algorithm 4 generally returns a small fraction of the core diagnoses returned by Algorithm 3. 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 4 (Column 4) by the number of core diagnoses returned by Algorithm 3 (Column 2) (equation (14)).

$$\%Improv = 1 - \frac{\#Core\ Diagnoses\ to\ \ Obtain\ All\ PDC\ (Algorithm\ 4)}{\#Core\ Diagnoses(Algorithm\ 3)} \tag{14}$$

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 3 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 3 to find within a reasonable period of time. In contrast, Algorithm 4 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 3 took 7318 seconds (more than 2 hours) to return 4096 core diagnoses. Algorithm 4 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 3 decreased from 299 to 16. In contrast, Algorithm 4's returns decreases from 18 to 10. In each case, Algorithm 4 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 3 is able to finish more quickly, narrowing its performance gap.

Where the goal graph contains many contribution links, one might decide to use Algorithm 3 instead of Algorithm 4. 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 2) in addition to preprocessing log data. We replaced "String" objects with "StringBuilder" objects with large buffers during the encoding of the propositional formula Φ 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.

7 Evaluation

We applied our framework to two medium-size public domain software systems to evaluate its correctness and performance: SquirrelMail (Castello 2007), a Web-based email client, and an ATM (Automated Teller Machine) simulation (Bjork 2007). 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.

7.1 The SquirrelMail running example

The encoding component preprocesses the SquirrelMail log data (Sect. 3.2) as described in Algorithm 2. The diagnostic component infers that the goal $g4$ and the task $a7$ are denied during 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 $g4$. Algorithm 3 returns the following 7 core diagnoses:

Core Diagnosis 1*:* $FD(a3, s)$*;* $FD(a7, s)$
Core Diagnosis 2*:* $FD(a4, s)$*;* $FD(a7, s)$
Core Diagnosis 3*:* $FD(a5, s)$*;* $FD(a7, s)$
Core Diagnosis 4*:* $FD(a3, s)$*;* $FD(a4, s)$*;* $FD(a7, s)$
Core Diagnosis 5*:* $FD(a3, s)$*;* $FD(a5, s)$*;* $FD(a7, s)$
Core Diagnosis 6*:* $FD(a4, s)$*;* $FD(a5, s)$*;* $FD(a7, s)$
Core Diagnosis 7*:* $FD(a3, s)$*;* $FD(a4, s)$*;* $FD(a5, s)$*;* $FD(a7, s)$

Algorithm 3 returns all core diagnoses which are possible combinations of tasks denials for tasks $a3$, $a4$, and $a5$—the tasks which account for the denial of goal $g4$. In contrast, Algorithm 4 returns individual task denials under one denied parent goal, leaving out their possible combinations. The following 4 *participating diagnostic components* were returned by Algorithm 4:

Diagnostic Component 1*:* $FD(a3, s)$
Diagnostic Component 2*:* $FD(a4, s)$
Diagnostic Component 2*:* $FD(a5, s)$
Diagnostic Component 3*:* $FD(a7, s)$

7.2 Performance evaluation with ATM

The ATM simulation case study is an illustration of OO design used in a software development class at Gordon College (Bjork 2007). 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 Fig. 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 Fig. 3. 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 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 1 and 2 to compare their efficiency on larger goal graphs. In both sets of experiments, the diagnostic component uses Algorithm 4 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 2) and the diagnostic component returns all *participating diagnostic components* (Algorithm 4). Our approach can therefore be applied to industrial software applications with medium-sized requirement models.

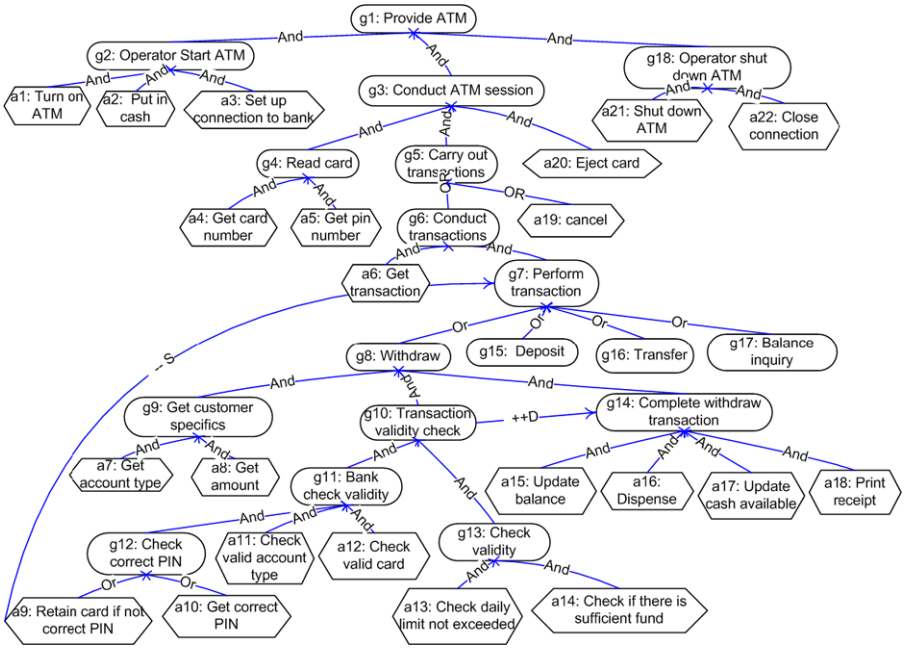

Fig. 3 Partial ATM goal model

Table 3 Tradeoff between monitoring overhead and diagnostic precision (first set of experiments)

#Mon	*#Diag*	*#Lit*	*#Clauses*	T_{avg} (s)	T_{sum} (s)
1	19	62	66	0.053	1.000
3	14	68	76	0.065	0.906
5	11	73	86	0.073	0.798
8	4	82	101	0.133	0.531
11	1	87	116	0.390	0.390

Table 3 reports the results of the first set of experiments. We injected an error into the implementation of task $a15$, *update balance*, with the goal of pinning down a single precise participating diagnostic component, namely $FD(a15)$. Column 1 in Table 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, Φ, encoded for the SAT solver, using log preprocessing (Algorithm 2). Column 6 gives the total time (in seconds) taken by the diagnostic component to find all *participating diagnostic components*. T_{sum} is the sum of the time taken to *encode* the goal graph into Φ (T_{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 ($T_{diagnose}$) by the total number of returned diagnostic components (*#Diag Set*) (equation (15)). $T_{diagnose}$ 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 (T_{decode}) (equation (16)). Column 5 lists the average time (T_{avg}) the solver took to find one participating diagnostic set (in seconds), calculated by dividing T_{sum} by *#Diag Set* (equation (17))

$$T_{sum} = T_{encode} + T_{diagnose} \times (\#Diag\ Set) \tag{15}$$

$$T_{diagnose} = T_{solve} + T_{decode} \tag{16}$$

$$T_{avg} = T_{sum}/(\#Diag\ Set) \tag{17}$$

In the first experiment (row 1 in Table 3), 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 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.

In experiments 2 to 5 (rows 2 to 5 in Table 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 par-*

Table 4 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	0.469	0.044	0.425	81	207
100	0.647	0.066	0.581	157	411
150	0.819	0.100	0.719	233	615
200	1.006	0.119	0.887	309	819
250	1.134	0.128	1.006	385	1023
300	1.260	0.156	1.103	461	1227
350	1.384	0.200	1.184	537	1431
400	1.529	0.225	1.304	613	1635
450	1.650	0.241	1.410	689	1839
500	1.787	0.278	1.509	765	2043
550	1.969	0.312	1.656	841	2247
600	2.159	0.341	1.819	917	2451
650	2.316	0.375	1.941	993	2655
700	2.397	0.406	1.991	1069	2859
750	2.516	0.434	2.082	1145	3063
800	2.725	0.487	2.238	1221	3267
850	2.900	0.528	2.372	1297	3471
900	2.975	0.526	2.450	1373	3675
950	3.259	0.584	2.675	1449	3879
1000	3.444	0.628	2.816	1525	4083

ticipating 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 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 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 obtain these larger goal graphs by cloning the ATM goal graph structure (Fig. 3) to itself. All the experiments are per-

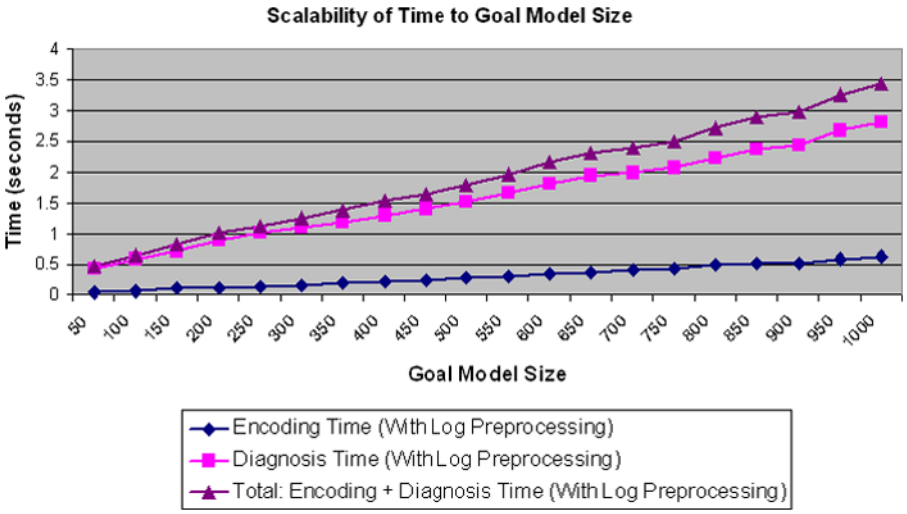

Fig. 4 Scalability to goal model size (encoding with log preprocessing)

formed with complete (task level) monitoring. Only 1 diagnostic component is therefore returned for each experiment. Column 1 in Table 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 Φ with log file preprocessing. Column 4, $T_{diagnose}$, lists the time taken by the SAT solver to solve Φ 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 Φ are listed in columns 5 and 6.

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 4) and the goal model size (the x-axis—the values of column 1 of Table 4). The three curves in Fig. 4 show that the diagnostic component scales to the size of the goal model when using 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 Algorithm 1, which encodes *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. Figure 6 depicts the relationships between the size of Φ (the total number of literals and clauses) generated by the two encoding algorithms and the goal model size. As discussed in Sect. 6, 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 relevant

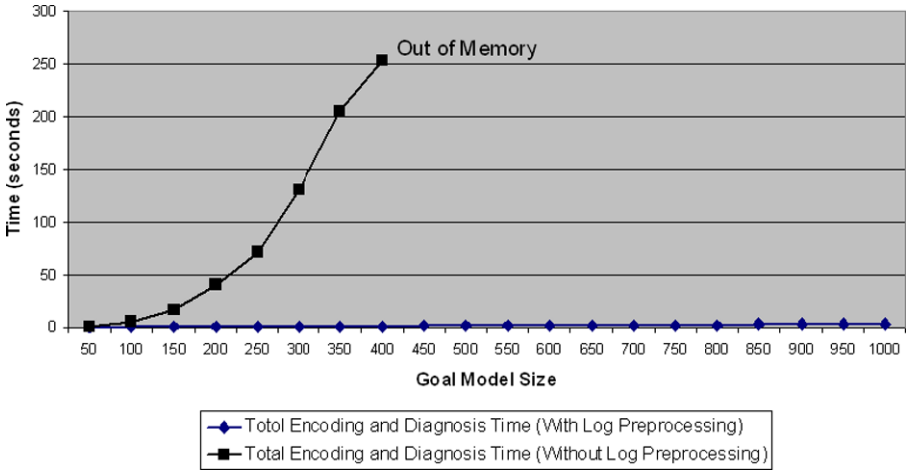

Fig. 5 Comparison of encoding and diagnostic time taken by the two encoding algorithms

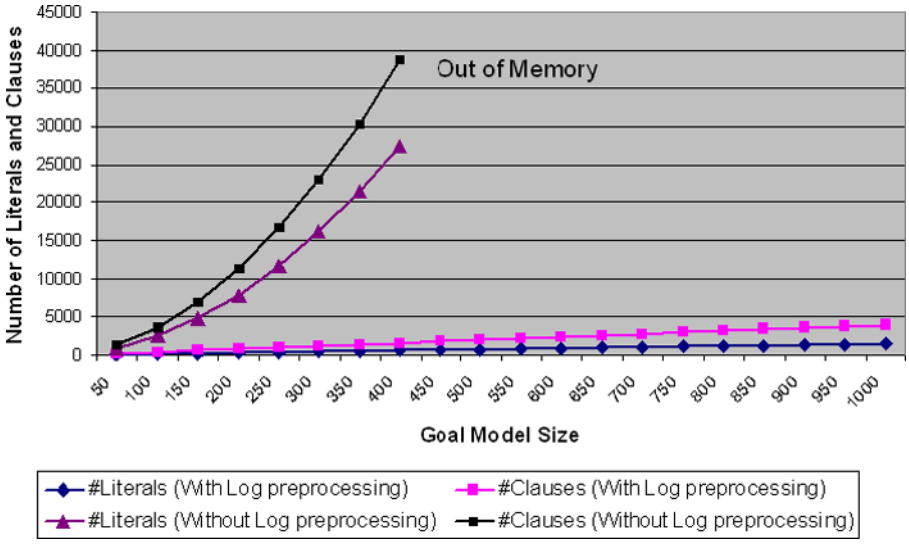

Fig. 6 Comparison of size of Φ generated by the two encoding algorithms

goal models, provided log file preprocessing is used, and all *participating diagnostic components* (instead of all diagnoses) are returned.

8 Multi-layer monitoring and diagnosis

8.1 Multi-layered architectures

Today's software systems are components of complex socio-technical systems consisting of business processes, applications and computing infrastructure. Service-

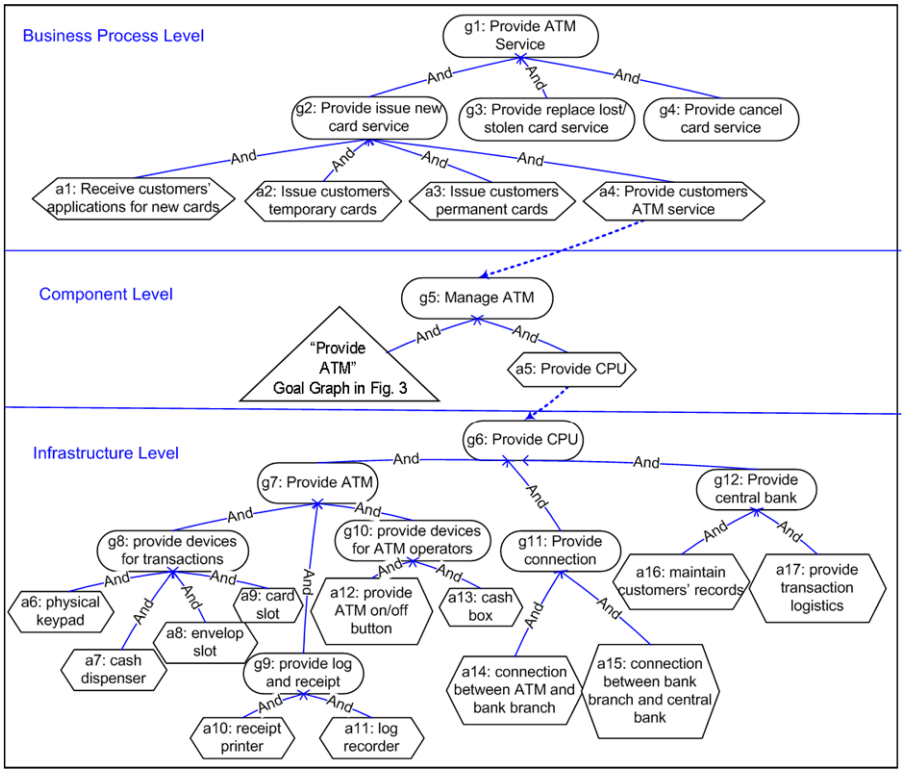

Fig. 7 ATM global goal model at the 3 layers of SOA

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 1990s, 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 (W3C 2002). 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 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 Fig. 7, the business process layer consists of three composite services depicted as goals: *issue customers new card service* ($g2$), *replace lost/stolen card service* ($g3$), and *cancel card service* ($g4$). The *issue customers new card service* can be further decomposed into four sub services depicted as tasks: *receive customers' applications for new cards* ($a1$), *issue customers temporary cards* ($a2$), *issue customers permanent cards* ($a3$), and *provide customers ATM service* ($a4$).

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 $a4$ (*provide customers ATM service*) from the business process layer decomposes into the root goal $g5$ (*Manage ATM*) in the component layer. $G5$ is then further decomposed to the goal graph presented in Fig. 3 (representing the requirements for the ATM service), and the task $a5$ (*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 uppers 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 $g6$ (*provide CPU*) depends on the availability and the correct functioning of the physical ATM ($g7$), the underlying connection between the ATM and the bank ($g11$), and central bank ($g12$).

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.

8.2 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 Fig. 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

Table 5 Number of goals and tasks at each SOA layer

#Business processes	#Goals/tasks at business process level	#Goals/tasks at component level	#Goals/tasks at infrastructure level
1	2	50	173
2	3	100	346
3	4	150	519
4	5	200	692
5	6	250	865
6	7	300	1038
7	8	350	1211
8	9	400	1384
9	10	450	1557
10	11	500	1730
11	12	550	1903
12	13	600	2076
13	14	650	2249
14	15	700	2422
15	16	750	2595
16	17	800	2768
17	18	850	2941
18	19	900	3114
19	20	950	3287
20	21	1000	3460

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 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 as the level of abstraction decreases. All the experiments were performed with log file preprocessing (Algorithm 2). We injected one error into the log files of each experiment. In each experiment, all the tasks were monitored, and Algorithm 4 was used for diagnosis.

Figure 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

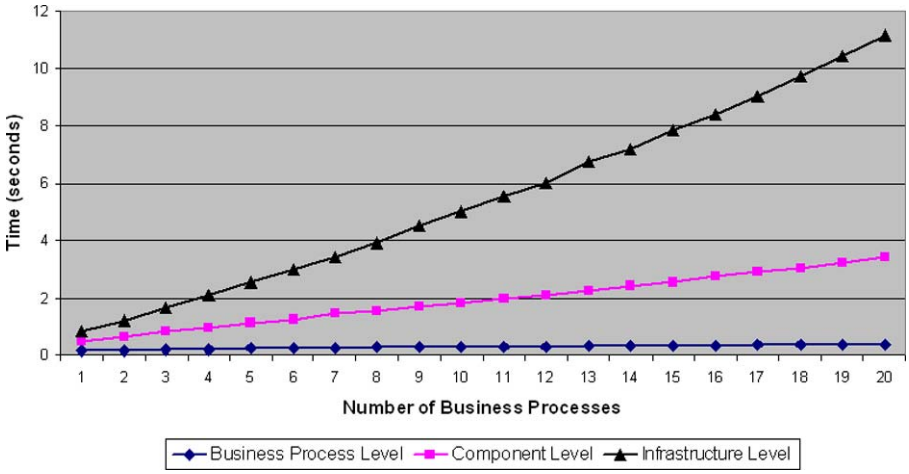

Fig. 8 ATM global goal model at the 3 layers of SOA

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.

9 Related work

9.1 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 (Feather et al. 1998; Fickas and Feather 1995) 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. The main difference between our work and Fickas' and Feather's 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 their approach, because their framework predefines requirement/assumption/remedy tuples.

Robinson has also presented a requirement monitoring framework, ReqMon in Robinson (2005). If an observed event is a satisfaction (or violation) event, satisfaction (or denial) status is updated for the requirement. The main difference between our approach and Robinson's is that ReqMon requires diagnostic formulae to be generated manually using obstacle analysis (Lamsweerde and Letier 2000). Our work, on the other hand, makes assumptions about what can fail. This allows our framework to automatically infer diagnoses given a model of system requirements and log data.

More recently, Winbladh et al. (2006) presented a goal-driven specification-based testing prototype that aims to find mismatches between actual and expected system behaviors. Their monitoring component accomplishes this by monitoring 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. Winbladh's proposal may therefore not readily scale to industrial sized applications.

None of the research discussed above (Feather et al. 1998; Fickas and Feather 1995; Winbladh et al. 2006) presents framework performance evaluations or discussions of scalability. It is therefore difficult to compare our respective approaches in terms of performance and scalability.

9.2 AI theories of diagnosis

Our work relies upon theories of diagnosis from AI (Reiter 1987; De Kleer et al. 1992; McIlraith 1998; Iwan 2002). Early AI research on diagnosis focused on static systems, and determined which components of the system were behaving normally and which were behaving abnormally. Two widely accepted AI definitions of diagnosis are consistency-based diagnosis (Reiter 1987; De Kleer et al. 1992) and abductive explanation (De Kleer et al. 1992).

Diagnosing dynamic systems has recently received more attention. McIlraith (1998) added a theory of action to traditional AI model-based diagnosis (Reiter 1987; De Kleer et al. 1992) and proposed a new type of diagnosis, *explanatory diagnosis*. Explanatory diagnosis conjectures a sequence of actions responsible for the system's aberrant behavior. McIlraith showed that conjecturing an explanatory diagnosis is analogous to AI planning. Iwan (2002) 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 and 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.

9.3 SAT-based goal analysis

In Sebastiani et al. (2004) a SAT based qualitative framework is proposed for finding satisfaction labels for a set of input goals, satisfying desired satisfaction labels for a set of target goals, using two SAT solvers. We adopted the goal model formalism in this work 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 Sebastiani et al. (2004) is

on satisfaction/denial label propagation only. In so far as satisfiability and deniability are two sides of the same coin, our work is in line with Sebastiani et al. (2004) with respect to label propagation.

10 Conclusions

We have presented a framework for monitoring and diagnosis of software systems' requirements, founded on AI theories of diagnosis. The framework has been implemented and evaluated through a series of experiments on two public domain software systems. The results of our experiments suggest that the framework is scalable and can be used with industrial-size software.

Before concluding, we note some features and limitations of our proposal. Firstly, The framework only monitors system task failures and would need extensions to handle failures caused by erroneous domain assumptions or malicious attacks. In addition, monitoring is limited to functional requirements and needs to be extended to handle non-functional requirements.

Along a different dimension, the proposed framework depends on the availability of requirements goal model and traceability links, in addition to application's source code. The hierarchical and layered structure of goal models enables us to model (and analyze) a software system at different levels of granularity. A large-scale, complex software system can be modeled using few goals and tasks at a high level of granularity. Take for example the two cases studies discussed in this paper: the SquirrelMail case study (69711 LOC) is larger than the ATM simulation (5000 LOC). Yet SquirrelMail's goal graph (11 goals/tasks) is smaller than that of the ATM simulation (88 goals/tasks) because it is modeled at a coarser level of granularity. In cases where a fine-grained goal model is not available, the system can be modeled with a few high level goals/tasks with relatively less effort. Ideally, a requirements model along with traceability links to code will already exist for software systems of the future, developed in accordance with recommended (e.g., model-driven) practices. However, even if these models and traceability links do exist, some effort may be required to instrument the code so that truth values of preconditions and effects (specified for goals/tasks) can be monitored. This instrumentation is automatic if the literals used to define goal/task preconditions and effects correspond directly to code-level artifacts (e.g., variables). If such a correspondence does not exist, the instrumentation procedure proposed in Zhou (2008) only generates instrumentation templates that need to be filled in manually.

Traceability links are required to map requirement denials to monitored system's source code to pinpoint possible failing components. Traceability links may also be represented at different granularity levels. Higher level traceability links map high level goals to larger-scaled software components, such as sub-systems and servers. Lower level traceability links map lower level goals and tasks to smaller-scaled components, such as one or several methods. If low-level, detailed traceability links are not available, higher level traceability links can be use to relate high level goals to larger-scaled sub-systems of the monitored system.

To evaluate the performance of our framework, we experimented with 20 progressively larger goal models containing from 50 to 1000 goals/tasks. We started with

an initial goal model with 50 goals/tasks. The larger goal graphs are obtained by cloning/copying the structure of this initial goal model multiple times. For example, the goal graph with 1000 goals/tasks contains 20 clones of the initial goal model with 50 goals/tasks. All the copies of the goal graphs are connected, with each copy becomes part of a bigger goal model. As can be seen from Fig. 6, the numbers of generated literals and clauses in the propositional formula Φ increased proportionally to the size of the goal model. This is because each goal/task is identified by a unique ID, and consequently unique literals are generated to present the denial and occurrence of each goal/task. Similarly for preconditions and effects, new literals are generated for them if their true values are observed at new timesteps in the log. Therefore, the cloning of goal models did not reduce the size of Φ that is used by the SAT solver.

In future work, we plan to investigate the impact of different, and randomly generated, goal model structures on the performance of the SAT solver. In addition, we plan to extend our framework by introducing failure probabilities. This will enable the diagnostic component to focus on finding the most probable diagnoses (those that pass a certain probability threshold), instead of all possible diagnoses. This extension is intended to further enhance the scalability of our framework. We also plan to complete the autonomic MAPE (Monitor, Analyze, Plan, and Execution) loop, (Kephart and Chess 2003) by designing, and providing tool support for failure repair.

Acknowledgements We are grateful to Borys Bradel for his expert advice on optimizations of the framework implementation. We thank anonymous reviewers for their useful comments.

References

Bjork, R.: An example of object-oriented design: an ATM simulation. http://www.cs.gordon.edu/courses/cs211/ATMExample/index.html/ (2007)

Castello, R.: Squirrel mail. http://www.squirrelmail.org/ (2007)

Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Sci. Comput. Program. **20**(1–2), 3–50 (1993)

Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. J. ACM **5**, 394–397 (1962)

De Kleer, J., Mackworth, A.K., Reiter, R.: Characterizing diagnoses and systems. Artif. Intell. **56**(2–3), 197–222 (1992)

Feather, M.S., Fickas, S., Van Lamsweerde, A., Ponsard, C.: Reconciling system requirements and runtime behavior. In: 9th International Workshop on Software Specification and Design (1998)

Fickas, S., Feather, M.: Requirements monitoring in dynamic environments. In: Second IEEE International Symposium on Requirements Engineering (1995)

Giorgini, P., Mylopoulos, J., Nicchiarelli, E., Sebastiani, R.: Reasoning with goal models. In: Conceptual Modeling, pp. 167–181. Springer, Berlin (2002)

Goldberg, E., Novikov, Y.: Berkmin: A fast and robust SAT-solver. In: Design, Automation, and Test in Europe, pp. 142–149 (2002)

Iwan, G.: History-based diagnosis templates in the framework of the situation calculus. AI Commun. **15**, 31–45 (2002)

Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Comput. Soc. **36**, 41–50 (2003)

Kiczales, G., Hilsdale, E., Hugunin, J.K.M., Palm, J., Griswold, W.: An Overview of AspectJ. Springer, Berlin (2001)

Lamsweerde, A.V., Letier, E.: Handling obstacles in goal-oriented requirements engineering. IEEE Trans. Softw. Eng. **26**, 978–1005 (2000)

Le Berre, D.: A satisfiability library for Java. http://www.sat4j.org/ (2007)

McIlraith, S.: Explanatory diagnosis: Conjecturing actions to explain observations. In: Principles of Knowledge Representation and Reasoning, pp. 167–179 (1998)

McIlraith, S., Fadel, R.: Planning with complex actions. In: International Workshop on Non-Monotonic Reasoning, pp. 356–364 (2002)
Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient sat solver. In: Design Automation, pp. 530–535. Assoc. Comput. Mach., New York (2001)
Mylopoulos, J., Chung, L., Nixon, B.: Representing and using nonfunctional requirements: a process-oriented approach. IEEE Trans. Softw. Eng. **18**(6), 483–497 (1992)
Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. **32**(1), 57–95 (1987)
Reiter, R.: The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In: Artificial Intelligence and Mathematical Theory of Computation, pp. 359–380 (1991)
Robinson, W.N.: Implementing rule-based monitors within a framework for continuous requirements monitoring. In: 38th Annual Hawaii International Conference on System Sciences (2005)
Ryan, L.: Efficient algorithms for clause-learning SAT solvers. Master's Thesis, Simon Fraser University (2004)
Sebastiani, R., Giorgini, P., Mylopoulos, J.: Simple and minimum-cost satisfiability for goal models. In: The 16th International Conference on Advanced Information Systems Engineering, vol. 4, pp. 20–33. Springer, Berlin (2004)
W3C: Web services (2002)
Wang, Y., McIlraith, S., Yu, Y., Mylopoulos, J.: An automated approach to monitoring and diagnosing requirements. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (2007)
Winbladh, K., Alspaugh, T.A., Ziv, H., Richardson, D.J.: An automated approach for goal-driven, specification-based testing. In: 21st IEEE/ACM International Conference on Automated Software Engineering (2006)
Yu, Y., Wang, Y., Mylopoulos, J., Liaskos, S., Lapouchnian, A., do Prado Leite, J.C.S.: Reverse engineering goal models from legacy code. In: Second IEEE International Symposium on Requirements Engineering, pp. 363–372 (2005)
Zhou, X.: A goal-oriented instrumentation approach for monitoring requirements. Master's Thesis, University of Toronto (2008)