Monitoring and Diagnosing Malicious Attacks with Autonomic Software*

Vítor E. Silva Souza and John Mylopoulos

Department of Information Engineering and Computer Science,
University of Trento, Italy
{vitorsouza,jm}@disi.unitn.it

Abstract. Monitoring and diagnosing (M&D) software based on requirement models is a problem that has recently received a lot of attention in field of Requirement Engineering. In this context, Wang et al. [1] propose a M&D framework that uses goal models to diagnose failures in software at different levels of granularity. In this paper we extend Wang's framework to monitor and diagnose malicious attacks. Our extensions include the addition of anti-goals to model attacker intentions, as well as context-based modeling of the domain within which our system operates. The extended framework has been implemented and evaluated through a series of experiments intended to test its scalability.

1 Introduction

Monitoring requirements for a software system during runtime and diagnosing failures is an old problem in Requirements Engineering (e.g., [2]). The problem has received considerable attention recently because of the importance that Industry and Academia are placing on adaptive/autonomic software systems. Such systems monitor their environment, diagnose problems (such as failures, sub-optimal behaviour, malicious attacks) and resolve them through some sort of a compensation mechanism. Our work addresses problems in this general area.

Wang et al. have proposed a general monitoring framework, paired with a SAT-based diagnostic reasoner adapted from Artificial Intelligence (AI) theories of action and diagnosis [1]. In this framework, software requirements are represented as goal models [3], and they determine what data to monitor for. At run-time, log data along with system requirements are coded into a propositional formula that is fed into a SAT solver. If the formula is unsatisfiable, then log data are consistent with the requirements model. If not, every possible interpretation that satisfies the formula represents a possible diagnosis of system failure(s). The proposed framework is able to diagnose failures at different levels of granularity. For instance, the diagnosis may be simply that the root-level goal failed, or it may detail which lower-level goal actually failed. Unfortunately,

* We are grateful to Yiqiao Wang for providing us with the implementation of her system and helping us understand it while designing its extensions.

A.H.F. Laender et al. (Eds.): ER 2009, LNCS 5829, pp. 84–98, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Wang's framework is limited to monitoring and diagnosing system requirements-related failures, such as system function failures. This means that the framework does not diagnose failures caused by unanticipated changes in the environment (for example, a system that was built to handle up to 10 users and fails when 20+ users log in concurrently). Nor can the system deal with malicious attacks, or failures caused by discrepancies between design models and the system's operations.

The main objective of this work is to extend Wang's framework with the purpose of monitoring and diagnosing malicious attacks. To this end, we have added support for a richer goal model that can represent not only stakeholder needs (goals), but also attacker intentions (anti-goals). Since the relationship between anti-goals and attacks (the plans by which an attacker attempts to fulfill his intentions) is notoriously context-dependent, we have also extended Wang's framework to represent and reason with contextual variability.

Anti-goals were proposed by van Lamsweerde et al. [4] to model security concerns during requirements elicitation. They are goals that belong to external malicious agents, whose purpose is to prevent the system from working by targeting one or more of its goals or tasks. By proposing this extension to Wang's framework and integrating it with the diagnostic reasoning, we cover the case in which all system components are working properly, but an external agent is preventing the system from functioning correctly.

Contextual variability in goal models was proposed by Lapouchnian [5] as a way to explicitly specify in the modeling notation how domain variability affects requirements. In this work we integrate this idea in the diagnostic framework, allowing for it to verify which goals and tasks of the model have an active context at any given time. This mechanism fits well in the architecture of systems that have a monitoring capability, and offers additional requirements for the monitoring component of such systems.

The rest of the paper is divided into the following sections: section 2 presents an overview of Wang's framework proposed in [1]; section 3 describes our extensions to Wang's framework. Section 4 details the implementation of these extensions, while section 5 presents the results of the evaluation experiments for the extended framework. Section 6 compares our proposal with related work. Finally, section 7 concludes and sketches ideas for future work.

2 The Diagnostic Framework

Wang et al. propose a framework to monitor the satisfaction of software requirements and diagnose what goes wrong in its execution in case of failure [1]. Figure 1 shows an overview of the framework's architecture.

The framework receives as input a goal model representing system requirements, a common use for goal models in the past decade [3]. Goal models represent requirements in a tree-like structure that starts at the main goal of the system and is decomposed (using AND or OR decomposition) in subgoals and tasks, which are the monitorable leaves of the tree. Functional and non-functional

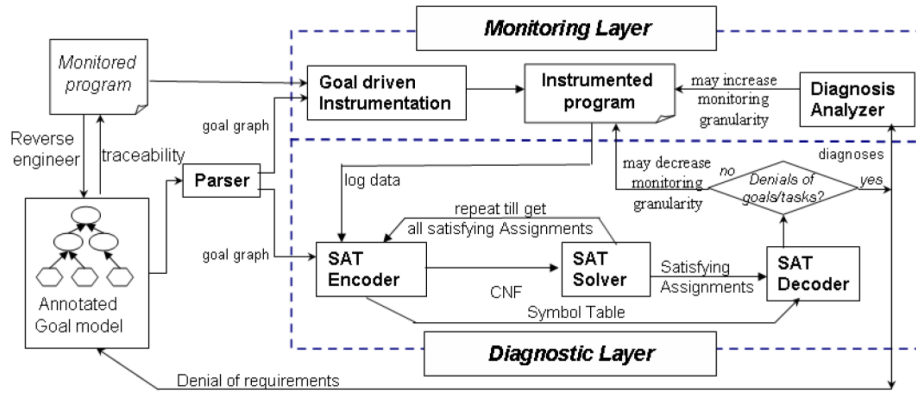

Fig. 1. Overview of the monitoring and diagnostic framework [1]

requirements are modeled as hard and soft goals respectively. Tasks and goals can also affect one another through contribution links: graph-like edges that indicate how the satisfiability or deniability of an element can affect another element.

Figure 2 shows the decomposition of one of the goals of the webmail system SquirrelMail [6]. To send an e-mail, one must fulfill all the sub-goals and tasks of goal $g1$'s AND-decomposition, namely, load the login form, process the send mail request and send the message. To process the send mail request, on the other hand, it's enough to accomplish one of the OR-decomposed children of goal *g1.2*: either you get the compose page or you report an IMAP error. The latter contributes positively to the non-functional requirement of usability. Possible contributions are helps (+), hurts (−), makes (++) and breaks (−−) [3].

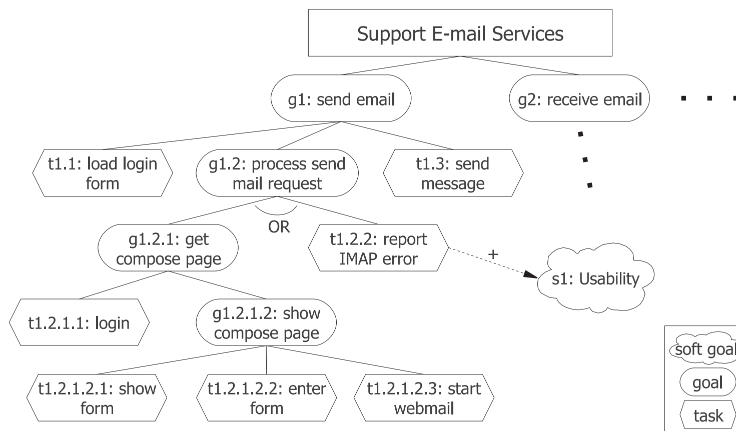

Fig. 2. Goal model of SquirrelMail [6] adapted from [7]

Each goal and task is given a precondition, an effect and a monitor status. Preconditions and effects are propositional formulas representing conditions that must be true before and after, respectively, a goal is satisfied or a task is executed [1]. The monitor status indicates if a task or goal should be monitored or not, making it possible to control the desired granularity level of diagnostics. Preconditions and effects for the SquirrelMail example can be seen in [1].

The monitoring layer instruments the source code of the program in order to provide the diagnostic layer a log, i.e., a set of truth values for an observed literal (preconditions and effects) or the occurrence of a task at a specific time-step [1]. The diagnostic layer can then produce axioms for three main purposes:

- **Deniability axioms**: if, according to the log, a task or goal occurred but either its precondition or its effect were not true before or after its occurrence, respectively, it's deemed denied, meaning there has been a problem with it;
- **Label propagation axioms**: propagate satisfiability and deniability between tasks and subgoals towards their parent goals, respecting the type of boolean decomposition (*and* or *or*) of the ancestors;
- **Contribution axioms**: calculate the effect that contribution links have on their targets based on the satisfiability or deniability of the source goal/task.

Together with the information from the log, the framework encodes all axioms in CNF and passes them to the SAT solver. The satisfying assignments are given to the SAT decoder, which translates them into diagnoses, i.e., information on task/goal satisfiability/deniability. Complete formalism on the axioms produced and algorithms used by the framework can be found in [1].

We have extended this framework in order to support contextual variability on goals and tasks and to take into account possible anti-goals that could be successfully preventing the system from working properly. These extensions and the changes in the goal meta-model that were necessary to accommodate them are presented in the following section.

3 The Proposed Extensions

In this work we propose two extensions to the framework described in section 2:

- **Anti-goals**: by supporting the inclusion of anti-goals in the requirements model, the framework can correctly diagnose the case in which none of the software components are faulty, but an external agent is preventing the system from working properly;
- **Contextual variability**: by supporting contextual variability in goal models, we allow for much richer requirement models "that will in turn lead to software systems that will deliver functionality closely matching customer expectations under many different circumstances" [5].

These extensions not only change the implementation of the framework, but also the format of the goal model input files, meaning they affect the goal meta-model, which describes how goal models are built. The goal meta-model for the

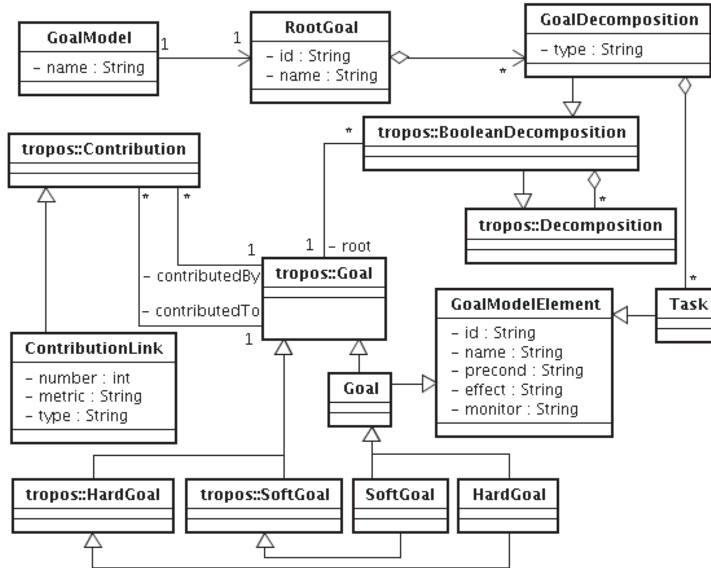

Fig. 3. The goal meta-model and its relationship with the Tropos meta-model in [8]

diagnostic framework extends the Tropos meta-model [8]. Figure 3 shows the goal meta-model and its relationship with the Tropos meta-model of [8].

Starting from the `GoalModel` class, we can see that a goal model has a root goal, which represents the objective of the system as a whole (in figure 2, "Support E-mail Services"). The root goal has a set of goal decompositions – *and* or *or*, depending on the `type` attribute –, which allows us to define complex goals in terms of sub-goals and tasks. Goals and tasks receive ID, name, precondition, effect and monitor status, which can be either *on* or *off*. Goals can contribute to other goals, specifying the metric – *helps*, *hurts*, *makes*, *breaks* – and the type: *s* (propagate satisfiability), *d* (propagate deniability) or *dual* (propagate both).

Next, we detail the changes in this meta-model and in the diagnostic framework for the inclusion of support for anti-goals and contextual variability.

3.1 Support for Anti-goals

Van Lamsweerde et al. propose a methodology for anti-goal analysis and their inclusion in requirement models in order to ensure the system satisfies critical properties such as safety, security, fault-tolerance and survivability [4]. Assuming the use of this methodology for the elicitation of anti-goals, we'd like to support them in the diagnostic framework.

The first step is the inclusion of the `AntiGoal` class and the `antiGoalTrees` association in the meta-model, as shown in figure 4 (affected classes are shaded). This allows for the inclusion of anti-goals in our goal models.

Next, we change the framework to consider the success of an anti-goal as a diagnosis. We assume the monitoring framework is capable of instrumenting the

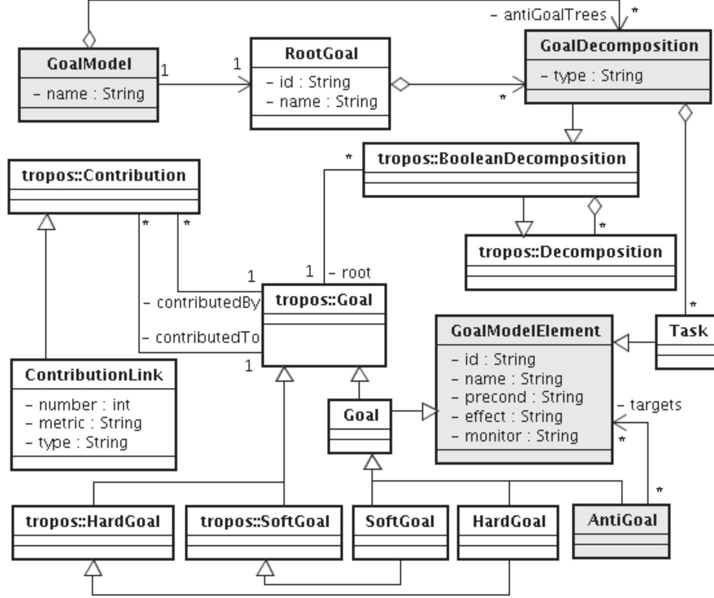

Fig. 4. New goal meta-model with support for anti-goals

source code of the system in a way it can detect when the tasks of the anti-goal tree successfully occur, as it already does with the tasks below the root goal. The current diagnostic framework is then capable of telling if an anti-goal occurred. To produce a SAT-based diagnosis we include axiom 1 in the encoded axioms.

AXIOM 1. **(Anti-goal satisfiability axioms)** *Given an anti-goal a, with starting and ending time-steps t_s and t_e and a set of target elements (goals or tasks) $\{e_1, e_2, \ldots, e_n\}$, the following axiom is produced*:

$$\forall e \in \{e_1, e_2, \ldots, e_n\} : occ(a,\, t_s,\, t_e) \wedge fd(e,\, s) \rightarrow fs(a,\, s) \qquad (1)$$

Intuitively, if a goal or task e is one of the targets of anti-goal a and we know that a has been attempted and that e has been denied, we can propose as a diagnose that a has been satisfied, meaning that there is a probability that e is not faulty[1], but a successfully prevented it from working properly. In other words, if it weren't for the anti-goal's success, e would also have been successful. As we cannot be sure the target goal/task hasn't failed by itself, both *fd(e, s)* and *fs(a, s)* diagnoses are proposed.

Figure 5 shows an example of an anti-goal for the SquirrelMail example of figure 2. The anti-goal *ag1* targets the goal *g1* and task *t1.3*.

The example below shows the log for an execution of the system under a Denial of Service (DoS) attack. Preconditions and effects for the anti-goal and its tasks can be inferred from the log:

[1] We use fault in the sense proposed by ISO/CD 10303-226: an abnormal condition or defect at the component, equipment, or sub-system level which may lead to a failure.

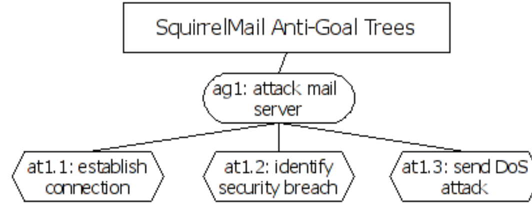

Fig. 5. Example anti-goal for the SquirrelMail goal model

connection_available(1); occ(at1.1, 2); connection_established(3); occ(at1.2, 4); breach_found(5); occ(at1.3, 6); dos_attack_performed(7); url_entered(8); occ(t1.1, 9); correct_form(10); ∼ wrong_imap(11); occ(t1.2.1.1, 12); correct_key(13); occ(t1.2.1.2.1, 14); occ(t1.2.1.2.2, 15); occ(t1.2.1.2.3, 16); webmail_started(17); occ(t1.3, 18); ∼ email_sent(19);

The proposed diagnoses for the example are *fd(t1.3, s); fs(ag1, s)*, i.e., either task *t1.3* is faulty or the anti-goal *ag1* prevented it from working.

3.2 Support for Contextual Variability

Lapouchnian believes that taking domain variability into consideration during requirements modeling will lead to software systems that match more closely customer expectations under many different circumstances. High-variability goal models attempt to capture many different ways goals can be met in order to facilitate in designing flexible, adaptive or customizable software [5].

Take, for instance, the example of figure 6. In this example, we extended the SquirrelMail example of figure 2 to capture the possibility of serving Web Services requests and performing auto-login in case the user has been authenticated before. New elements added to the goal model are shaded and, for reasons of space, only the subtree of goal *g1.2* is shown.

This causes a problem in our diagnostic framework: for goal *g1.2.1* to occur, since it's AND-decomposed, both *login* and *auto-login* tasks must occur, which is redundant. With support for contexts, all we have to say is that these tasks occur in different contexts. Furthermore, contexts can help decide which route to take to fulfill a goal in case of an OR-decomposition, such as goal *g1.2*: when a Web Services request is detected, follow goal *g1.2.3*, otherwise try goal *g1.2.1*.

To define contexts and annotate goal model elements with them, changes in the goal meta-model are necessary. Figure 7 shows the new classes added to the meta-model (shaded) and their relationship with the existing ones.

Goal models can now define context dimensions and organize them in hierarchies: a context dimension is either defined by sub-dimensions or by a formula in propositional logic. Then, goals, tasks and links can be annotated with context to indicate it only makes sense for them to occur if the context is active.

Figure 8 presents the context hierarchies for the new SquirrelMail example shown in figure 6. The formulas that define each leaf-level context dimension are

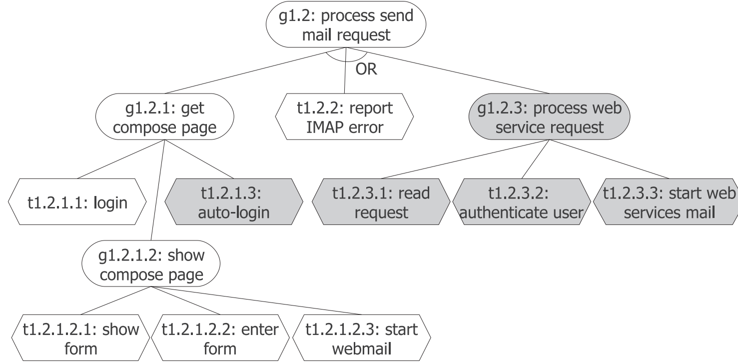

Fig. 6. Example of a contextual goal-model based on the SquirrelMail example

shown in the diagram. Tasks *t1.2.1.1* and *t1.2.1.3* are annotated with dimensions *User Not Authenticated* and *User Authenticated*, respectively, while goals *g1.2.1* and *g1.2.3* are annotated with *Web Client* and *Web Services Client* respectively.

Contexts are deemed inactive at time-step 0 and considered active in a given time-step if any of their sub-contexts are active or, in the case of leaf-level dimensions, if the formula is true at that time-step, considering the latest information on the log. This means that the program code instrumented by the monitoring framework must be capable of logging information related to these formulas.

Moreover, we'd also like to know when a goal or task has occurred outside its context. This could mean the instrumented program code isn't able to detect context change or that the software is not following the specifications. For this purpose, we also encode axioms so the result is provided as a diagnosis:

Axioms 2 and 3. **(Invalid occurrence axioms)** *Given a goal g, with starting and ending time-steps t_s and t_e or a task a, with occurring time-step t_{occ}. Suppose the function $context_formula(e, t)$ that calculates the truth value of the conjunction of all the context formulas of the annotations of element e in a given time-step t. The following axioms are produced*:

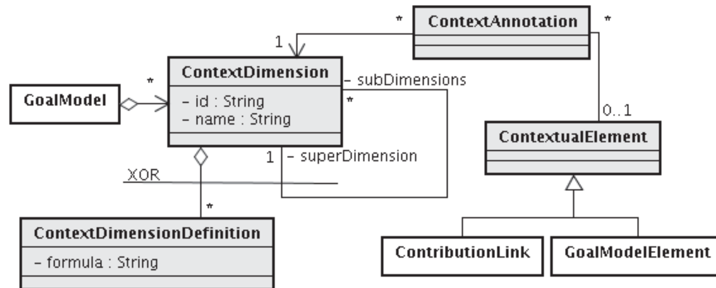

Fig. 7. Additions to the goal meta-model to deal with contextual variability

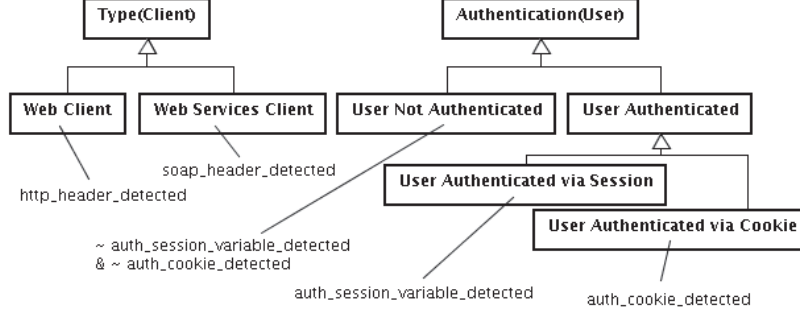

Fig. 8. Contexts for the new SquirrelMail example of figure 6

$$occ(g, t_s, t_e) \wedge \neg context_formula(g, t_s) \rightarrow iocc(g, s) \quad (2)$$

$$occ(a, t_{occ}) \wedge \neg context_formula(a, t_{occ}) \rightarrow iocc(a, s) \quad (3)$$

Intuitively, a goal or task has an active context at a given time-step t if all of its annotated context dimensions are active at that moment. A dimension is active if its context formula is true. If non-leaf, its context formula is the disjunction of the context formulas of its sub-dimensions (a non-leaf dimension is active if any of its sub-dimensions is). Thus, axioms 2 and 3 state that if any of the contexts annotated in the goal or task isn't active but the goal or task occurred anyhow, an invalid occurrence (*iocc()*) diagnosis should be produced.

The example below shows an execution log for the case where the *auto-login* task has occurred because a cookie was detected in the user's computer. No diagnoses are produced, as no errors occurred.

*url_entered(1); http_header_detected(2); auth_cookie_detected(3);
∼ wrong_imap(4); occ(t1.2.1.3, 5); correct_key(6); occ(t1.2.1.2.1, 7);
occ(t1.2.1.2.2, 8); occ(t1.2.1.2.3, 9); webmail_started(10); occ(t1.3, 11);
email_sent(12);*

4 Implementation

Wang et al. [1] describe the main algorithms used by the diagnostic framework. In this section, we present the new algorithms that were included in order to produce new axioms that allow the SAT solver to diagnose malicious attacks and consider contextual information.

The *encode_anti_goal_axioms* algorithm analyzes all anti-goals in the goal model that occurred according to the log. For every target of the occurring anti-goals, it encodes an anti-goal success axiom in the form $occ(ag, t_s, t_e) \wedge fd(e, s) \rightarrow fs(ag, s)$.

$encode_anti_goal_axioms(goal\,model,\ log)$ {
  $for\ each\ occurring\ anti\ goal\ ag$

```
        if (precond(ag) ≠ null) ∨ (effect(ag) ≠ null)
            for each element e in targets(ag)
                Φ ← Φ ∧ encodeAntiGoalSuccessAxiom(ag, e)
    return Φ
}
```

To produce invalid occurrence axioms such as $occ(g, t_s, t_e) \wedge \neg context_formula(g, t_s) \rightarrow iocc(g, s)$ (for goals) and $occ(a, t_{occ}) \wedge \neg context_formula(a, t_{occ}) \rightarrow iocc(a, s)$ (for tasks), the algorithm *encode_invalid_occurrence_axioms* was implemented. This algorithm analyzes every goal and task that has occurred and is annotated with contextual information. For each context dimension annotated in the element, it builds the contexts formula and encodes the invalid occurrence axiom.

```
encode_invalid_occurrence_axioms(goal model, log) {
    for each occurring goal and task e
        if (context_annotations(e) ≠ null)
            for each context dimension c in annotations(e)
                Δ ← Δ ∧ build_context_formula(c, log)
                Φ ← Φ ∧ encodeInvalidOccurrenceAxiom(e, Δ)
    return Φ
}
```

Each context dimension's formula is built with algorithm *build_context_formula*, which recursively navigates the context hierarchy depth-first, joining the leaf-dimensions' formulas in a disjunction.

```
build_context_formula(c, log) {
    if (hasSubDimension(c))
        for each context sub dimension sc of c
            δ ← δ ∨ build_context_formula(sc, log)
        return δ
    else
        return formula(c)
}
```

Last, but not least, changes on how the framework decides if a goal has or hasn't occurred were made due to the new contexts support. After defining any goal with a descendant occurring task as having occurred, *confirm_goal_occurrence* navigates each goal sub-tree from bottom-up, canceling the goal occurrence if any non-occurring sub-goal or task is found with an active context.

```
confirm_goal_occurrence(goal model, log, g) {
    for each sub goal sg
        confirm_goal_occurrence(goal model, log, sg)
    if (decompositionType(g) = AND)
        for each sub goal and task e of g
```

$$if\ (hasNotOccurred(e) \wedge isContextActive(e))$$
$$return\ false$$
$$return\ true$$
}

A prototype of the diagnosing framework was developed in Java.

5 Evaluation of the Proposed Extensions

As done previously in [1], we used the SquirrelMail example to illustrate the characteristics of the framework and evaluated its scalability using the Automated Teller Machine (ATM) simulation example [9]. The experiments were run in a computer with an Intel Core 2 Duo P8400 2.26GHz with 3Mb L2 1066MHz cache and 2GB DDRII 800MHz RAM.

5.1 The SquirrelMail Example

The SquirrelMail example used in [1] has been adapted to demonstrate throughout the paper the new features of the framework.

The log data in section 3.1 shows an error in task *t1.3* (and, consequently, on goal *g1*), since the task has occurred but its effect (*email_sent*) wasn't true in the subsequent time-step. This would usually mean task *t1.3* is faulty. However, with new support for malicious attacks diagnosis, the system also monitors for the successful occurrence of tasks *at1.1*, *at1.2* and *at1.3*, shown in figure 5, meaning anti-goal *ag1* might have been successful in stopping task *t1.3* from working. Therefore, $fs(ag1, s)$ is included as diagnosis alongside $fd(t1.3, s)$.

Another log is shown in section 3.2, referring to the extended SquirrelMail example of figure 8. The log shows the case in which task *t1.2.1.3* occurs instead of *t1.2.1.1*, as the former has an active context (cookie detected on time-step 3) and the latter doesn't. The result is no diagnosis produced and the goal *g1.2.1* occurs normally even though it's AND-decomposed and *t1.2.1.1* doesn't occur, as that child has an inactive context. The exact same log without *auth_cookie_detected*(3) produces *iocc*($t1.2.1.3$, s) as diagnosis, as a task (or goal) should not occur with an inactive context.

5.2 Performance Evaluation with the ATM Example

Tests with the ATM case study were based in the goal model obtained in [1] by reverse-engineering its OO design [9] and were also adapted to include malicious attacks and contextual information.

The base test set is composed of 20 goal models and their respective logs. The first model contains 50 goal model elements extracted from the ATM simulation requirements. The other models repeat these elements to produce goal models of sizes varying from 100 to 1000. Two new test sets were generated, one with anti-goals and another with contextual information.

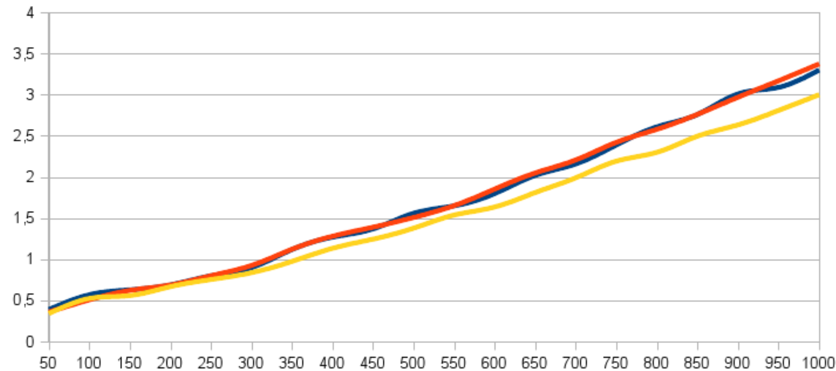

Fig. 9. Performance evaluation of the ATM Simulation case study

Figure 9 shows the time in seconds (y-axis) taken to execute the diagnose in each test set (x-axis). The lines are very close together, which shows the inclusion of anti-goals and contextual information hasn't changed the performance of the diagnosing framework. The base test set starts with 0,39s in the 50-elements goal model and goes up to 3,30s in the 1000-elements model. The test cases for anti-goals and contextual information have times that vary from 0,36s to 3,37s and from 0,34s to 3,00s, respectively. When contextual information is taken into account, processing is faster because only parts of the goal model are considered for each active context.

6 Related Work

As related work to her proposal, Wang et al. [1] cite the ReqMon framework [10] and the works by Fickas & Feather [2] and Winbladh et al. [11]. However, none of these deal specifically with malicious attacks or contextual information.

There are many proposals for security requirement engineering. Haley et al. [12] define security requirements as constraints to functions of the system and propose a framework that explicitly includes context and determines satisfaction of the security requirements. Elahi & Yu [13] incorporate security trade-off analysis into requirements engineering and develop an i*-based, goal-oriented framework for modeling and analyzing them, accompanied by a knowledge base of security trade-offs. Sindre & Opdahl propose ReqSec [14], a methodology that builds on misuse cases to integrate elicitation, specification and analysis of security requirements with the development of functional requirements of the system. Rodriguez et al. [15] propose M-BPSec, a UML 2.0 profile over the Activity Diagram which allows for the capturing of security requirements and creating of secure business processes. Mellado et al. [16] have extended the Security Requirements Engineering Process for Software Product Lines (SREPPLine) for the management of security requirements variability. These proposals focus on security requirements from analysis to validation, but not on runtime. Our work

focuses on monitoring software at runtime, for purposes of diagnosing attacks and the system components they might affect.

Some proposals include a monitoring component, but without an associated diagnostic engine. Giorgini et al. [17] extend the i*/Tropos modeling framework to define Secure Tropos, which includes the concepts of trust, ownership and delegation of permission. Within this framework, they model certain types of security requirements (for example, access control policies) and can apply formal reasoning techniques to determine whether a system specification violates any security requirements. This proposal does use monitoring (by actors, who can be system, human, or organizational) to legitimize the delegation of services to untrusted actors. Graves & Zulkernine [18] have modified an existing Intrusion Detection System (Snort) in order to use rules with context information translated from attack scenarios written in a software specification language (AsmL). Snort monitors the runtime operation of a system and alerts when a security requirement has been violated.

On the context variability side, many works on context-aware systems focus on the requirements phase. For instance, Hong et al. [19] focus on context-awareness for product families and use problem frames for representing variability in the problem space, rather than the solution space. For ubiquitous computing, Salifu et al. [20] extends the notion of context as basis of the proposed methodology for requirement elicitation. Semmak et al. [21] extends the Kaos meta-model with variability concepts (along similar lines to our own work) in order to specify a requirements family model, which then derives different specifications depending on stakeholders needs. The key difference in our approach is purpose: we model contextual variability to be able to monitor applications that have richer goal models, such as for autonomic systems.

Ali et al. [22] propose an extension to the Tropos framework for developing location-based software. Our proposal shares a lot of similarity with theirs (namely, context/location-based or-decomposition, and-decomposition and contribution to softgoals), but focuses on monitoring and diagnosing instead of modeling and analysis. Both works can be considered complimentary, as our framework could be used to monitor and diagnose location-based software developed with location-based Tropos.

7 Conclusion

By supporting anti-goals and contextual variability in the monitoring & diagnosis framework, we have extended the domain of applicability of Wang's M&D framework, notably to support monitoring and diagnosis for failures provoked by malicious attacks. The extensions have been evaluated for feasibility and scalability up to medium-sized goal models.

Future work includes the study of possible compensation mechanisms. Once our system has determined that an attack is in progress, it needs to select a compensation that will hopefully prevent the attack from succeeding. In addition, our diagnostic reasoner needs to be complemented with probabilistic reasoning

techniques that looks for probable attacks, their chances of success and the chances of particular compensation mechanisms thwarting such attacks.

References

1. Wang, Y., McIlraith, S.A., Yu, Y., Mylopoulos, J.: Monitoring and diagnosing software requirements. Automated Software Engineering 16, 3–35 (2009)
2. Fickas, S., Feather, M.: Requirements monitoring in dynamic environments. In: Proceedings of the Second IEEE International Symposium on Requirements Engineering, vol. 1995, pp. 140–147 (1995)
3. Giorgini, P., Mylopoulos, J., Nicchiarelli, E., Sebastiani, R.: Reasoning with goal models. In: Spaccapietra, S., March, S.T., Kambayashi, Y. (eds.) ER 2002. LNCS, vol. 2503, pp. 167–181. Springer, Heidelberg (2002)
4. van Lamsweerde, A., Brohez, S., De Landtsheer, R., Janssens, D.: From system goals to intruder anti-goals: Attack generation and resolution for security requirements engineering. In: Workshop on Requirements for High Assurance Systems (RHAS 2003), pre-workshop of the 11th International IEEE Conference on Requirements Engineering, Software Engineering Institute Report, September 2003, pp. 49–56 (2003)
5. Lapouchnian, A., Mylopoulos, J.: Modeling domain variability in requirements engineering with contexts. In: ER 2009: Proceedings of the 28th International Conference on Conceptual Modeling, Springer, Heidelberg (2009)
6. Castello, R.: Squirrelmail (2009), http://www.squirrelmail.org
7. Yu, Y., Wang, Y., Mylopoulos, J., Liaskos, S., Lapouchnian, A., do Prado Leite, J.: Reverse engineering goal models from legacy code. In: Proceedings of the 13th IEEE International Conference on Requirements Engineering, 2005, August -2 September 2005, pp. 363–372 (2005)
8. Susi, A., Perini, A., Mylopoulos, J., Giorgini, P.: The tropos metamodel and its use. Informatica 29, 401–408 (2005)
9. Bjork, R.C.: Atm simulation (2009), http://www.cs.gordon.edu/courses/cs211/ATMExample/
10. Robinson, W.N.: Implementing rule-based monitors within a framework for continuous requirements monitoring. In: HICSS 2005: Proceedings of the 38th Annual Hawaii International Conference on System Sciences - Track 7, p. 188a. IEEE Computer Society, Los Alamitos (2005)
11. Winbladh, K., Alspaugh, T.A., Ziv, H., Richardson, D.J.: An automated approach for goal-driven, specification-based testing. In: ASE 2006: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, Washington, DC, USA, pp. 289–292. IEEE Computer Society, Los Alamitos (2006)
12. Haley, C.B., Moffett, J.D., Laney, R., Nuseibeh, B.: A framework for security requirements engineering. In: SESS 2006: Proceedings of the 2006 international workshop on Software engineering for secure systems, pp. 35–42. ACM, New York (2006)
13. Elahi, G., Yu, E.: A goal oriented approach for modeling and analyzing security trade-offs. In: Parent, C., Schewe, K.-D., Storey, V.C., Thalheim, B. (eds.) ER 2007. LNCS, vol. 4801, pp. 375–390. Springer, Heidelberg (2007)
14. Sindre, G., Opdahl, A.L.: Reqsec - requirements for secure information systems, project proposal for fritek (2007), http://www.idi.ntnu.no/~guttors/reqsec/plan.pdf

15. RodrÃguez, A., FernÃndezMedina, E., Piattini, M.: M-bpsec: A method for security requirement elicitation from a uml 2.0 business process specification. In: Parent, C., Schewe, K.-D., Storey, V.C., Thalheim, B. (eds.) ER 2007. LNCS, vol. 4801, pp. 106–115. Springer, Heidelberg (2007)
16. Mellado, D., Fernandez-Medina, E., Piattini, M.: Security requirements variability for software product lines, pp. 1413–1420 (March 2008)
17. Giorgini, P., Massacci, F., Mylopoulos, J., Zannone, N.: Modeling security requirements through ownership, permission and delegation. In: RE 2005: Proceedings of the 13th IEEE International Conference on Requirements Engineering, Washington, DC, USA, pp. 167–176. IEEE Computer Society, Los Alamitos (2005)
18. Graves, M., Zulkernine, M.: Bridging the gap: software specification meets intrusion detector. In: PST 2006: Proceedings of the 2006 International Conference on Privacy, Security and Trust, pp. 1–8. ACM, New York (2006)
19. Hong, D., Chiu, D.K.W., Shen, V.Y.: Requirements elicitation for the design of context-aware applications in a ubiquitous environment. In: ICEC 2005: Proceedings of the 7th international conference on Electronic commerce, pp. 590–596. ACM, New York (2005)
20. Salifu, M., Nuseibeh, B., Rapanotti, L., Tun, T.T.: Using problem descriptions to represent variability for context-aware applications. In: First International Workshop on Variability Modelling of Software-intensive Systems (2007)
21. Semmak, F., Gnaho, C., Laleau, R.: Extended kaos to support variability for goal oriented requirements reuse. In: Proceedings of the International Workshop on Model Driven Information Systems Engineering: Enterprise, User and System Models (MoDISE-EUS 2008, in conjunction with CAiSE), pp. 22–33 (2008)
22. Ali, R., Dalpiaz, F., Giorgini, P.: Location-based software modeling and analysis: Tropos-based approach. In: Li, Q., Spaccapietra, S., Yu, E., Olivé, A. (eds.) ER 2008. LNCS, vol. 5231, pp. 169–182. Springer, Heidelberg (2008)