Towards a Requirements-Driven Framework for Detecting Malicious Behavior against Software Systems

Hamzeh Zawawy
University of Waterloo
Waterloo, Canada
hzawawy@engmail.uwaterloo.ca

Kostas Kontogiannis
Natl. Technical University of Athens
Athens, Greece
kkontog@softlab.ntua.gr

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

Serge Mankovskii
CA Labs
Markham, Canada
Serge.Mankovskii@ca.com

June 27, 2011

Abstract

Root cause determination for software failures that occurred due to intentional or unintentional third party activities is a difficult and challenging task. In this paper, we propose a new technique for identifying the root causes of system failures stemming from external interventions that is based first, on modeling the conditions by which a system delivers its functionality utilizing goal models, second on modeling the conditions by which system functionality can be compromised utilizing anti-goal models, third representing logged data as well as, goal and anti-goal models as rules and facts in a knowledge base and fourth, utilizing a probabilistic reasoning technique that is based on the use of Markov Logic Networks. The technique is evaluated in a medium size COTS based system and the DARPA 2000 Intrusion Detection data set.

1 Introduction

Root cause analysis refers to the identification of errors that lead to failures in a software system. By the term failure we mean the deviation of the system's observed behaviour from the expected one, while by the term error we refer to a system bug or internal misconfiguration.

Overall, the root cause analysis problem is a difficult and challenging task to solve, due to the complexity of interdependencies between the systems components, the often incomplete logged data that could fall short explaining the cause of a failure and, the lack of models that could be used to denote conditions and requirements under which a failure could be manifested. The problem becomes even more complex when the failures are not due to internal system errors, but due to external actions by third parties that intentionally or unintentionally cause the failure of a software system. The software engineering community has responded by proposing different types of techniques for addressing the root cause determination problem. One type of techniques is based on the association of event patterns with known system errors when a failure is observed. Another type of techniques is based on collections of rules that apply diagnostic expert knowledge.

A third type of techniques is based on model driven root cause diagnosis where the a model of the system is built and an analysis process attempts to identify the components or the conditions which may explain an observed failure.

In this paper, we take a complementary approach where we attempt to combine the best features of the aforementioned types of techniques. More specifically, we first utilize model driven approaches to denote not only the conditions, constraints, actions and tasks, a system is expected to enact in order to deliver its functionality but also, the conditions, constraints, actions and tasks that can be taken by an external user to invalidate or threaten a given system action or task. We refer to the first types of models as goal models and, the second types of models as anti-goal models. Second, we utilize pattern based approaches to limit the size of the logs that can be considered to satisfying or denying a specific system goal, model action, or task. Third, we utilize rule based approaches to denote goal and anti-goal models as diagnostic rules that can be enacted in a probabilistic reasoning framework provided in the form of a Markov Logic Network. More specifically, once a failure is observed, the corresponding goal model is analyzed to identify the tasks and actions that may explain the observed failure and the anti-goal model is investigated to increase the confidence of the goal model analysis. The confidence increase is manifested when a goal model task or action fails while its corresponding anti-goal succeeds. Confidence levels are computed using Markov Logic Network inference as a function of the reasoning process combined with past observations that provide a level of training for the reasoning process.

The paper is organized as follows. Section 2 covers the research baseline and related work. Section 3 presents the log data representation and filtering approaches. In section 4, we present the overall architecture and processes of the proposed framework. Section 5 describes the diagnostics process using a running example. Section 6 presents a COTS-based case study and a scalability evaluation for the framework. Section 7 concludes the paper and presents directions for further research.

2 Related Research

This section presents related work in the areas of requirements modeling, root cause diagnostic reasoning, and malicious behavior detection.

Goal Modeling

Goal Models are tree based formalisms that can be used to represent and denote not only the conditions and the constraints under which functional and non-functional requirements of a system can be delivered but also, to represent and denote positive and negative contributions, a requirement or a design decision may have on other requirements. In this respect, *functional requirements* are represented as hard goals, while *non-functional requirements* are represented as soft goals [3]. A goal model is a tree structure including AND/OR decompositions of goals into subgoals and tasks (leaf subgoals). When a goal G is AND (OR) decomposed into subgoals $G_1, \ldots, G_n$, then all (at-least-one) of G's subgoals must be satisfied for G to be satisfied. Goals and tasks can impact each other's satisfaction using contribution links: $\xrightarrow{\text{++S}}$, $\xrightarrow{\text{--S}}$, $\xrightarrow{\text{++D}}$ and $\xrightarrow{\text{--D}}$.

More specifically, given two goals G_1 and G_2, the link $G_1 \xrightarrow{\text{++S}} G_2$ (respectively $G_1 \xrightarrow{\text{--S}} G_2$) denotes that if G_1 is satisfied, then G_2 is strongly (++) satisfied (respectively denied).

The meaning of links $\xrightarrow{\text{++D}}$ and $\xrightarrow{\text{--D}}$ are dual w.r.t. to $\xrightarrow{\text{++S}}$ and $\xrightarrow{\text{--S}}$ respectively, by inverting satisfiability and deniability. The class of goal models used in our work has been formalized by Giorgini et al. [5], where appropriate algorithms have been proposed for inferring whether a set of root-level goals are satisfied or not.

Anti-Goal Modeling

Anti-goals represent the tasks and actions of an intruder with the intention to threaten or compromise specific security goals. Similarly to goal models, anti-goal models are also denoted as AND/OR trees built through systematic refinement until leaf nodes are derived. Leaf nodes represent tasks (actions) an intruder can perform to fulfill an anti-goal, and conse-

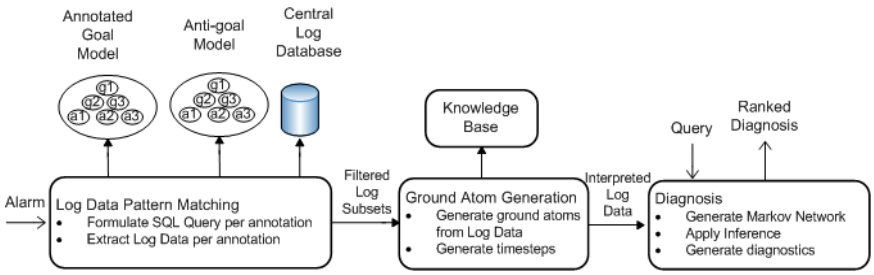

Figure 1: Logical Architecture of the Malicious Behavior Detection Framework.

quently deny a stakeholder's goal for the system. Anti-goal models were initially proposed by Lamsweerde [12] to model security concerns during requirements elicitation. Formally, anti-goals are defined as follows: let G be a goal and Dom a set of domain properties. An assertion O is said to be an *obstacle* (anti-goal) to G in Dom if and only if the following hold:

1. $\{O, Dom\} \models \neg\ G\ (O)$
2. $\{O, Dom\} \mid false_{Dom}$

Condition (1) states that the negation of a goal is a logical consequence of the obstacle specification and the set of domain properties available; condition (2) states that the obstacle may be logically consistent with the domain of the logs and the goal models.

The technique proposed by Lamsweerde et al. (2000) to build and refine anti-goals, involves building two models interactively and concurrently that is, a) a goal model representing the functional and non-functional requirements of the system-to-be, and; b) an anti-goal model derived from the requirement goal model that specifies how assets of the application-to-be can be compromised. The process of elaborating anti-goal models can be either done informally by asking HOW/WHY questions or formally by regression through the goal model and the domain theory or by use of refinement and obstruction patterns as discussed in [13, 14].

Monitoring & Diagnosis

An interesting recent work in log analysis and reduction is the paper by Al-Mamory et al. [1] where the focus is to reduce false positives in root cause analysis results. That approach uses data mining to group similar alarms into generalized alarms and then analyze these generalized alarms and classify them into true and false alarms. Consequently, the generalized alarms can be used as rules to filter out false positives.

Wang et al. [15] proposed a framework to monitor the satisfaction of software requirements and provide a diagnosis in case of failure. Wang's framework is limited to monitoring requirement satisfaction and in particular failures due to system function or task failures.

Souza et al. extended Wang's framework [15] to detect failures caused by malicious attacks [10]. In addition to monitoring the satisfaction of goal models, Souza et al. added support for a richer goal model that can represent not only stakeholder needs (goals), but also attacker intentions (anti-goals). Security requirements and anti-goal models can be derived by analyzing past cases, utilizing libraries of attack or threat trees, or derived systematically as presented by Lamsweerde in [13].

Thus, if a goal or task a is one of the targets of anti-goal ag and we know that the attack ag has occurred between timesteps t_s and t_e and that the goal/task a has been denied during that same time interval, we can infer that ag has been satisfied. We can also infer that there is a chance that a is not faulty, but the attack ag successfully prevented it from working properly. In other words, if it weren't for the successful execution of the attack, the goal/task a would have succeeded. Since we cannot be sure the target goal/task hasn't failed by itself, both *fd(a, s)* (action is denied) and *fs(ag, s)* (anti-goal has succeeded) diagnoses are proposed.

Markov Logic Networks for Probabilistic Reasoning

Markov Logic Networks (MLNs) have been recently proposed in the research literature as a way of providing a framework that combines first order logic and probabilistic reasoning [4]. In this context, a knowledge base denoted by first-order logic formulae represents a set of hard constraints on the set of possible worlds that is, if a world violates one formula, it has zero probability of existing. Markov logic softens these constraints by making a world that violates a formula to be less probable but still, possible. The more formulas it violates, the less probable it becomes. More information and detailed discussion on Markov Logic Networks can be found in the paper by Domingos [4].

Malicious Behavior Categorization

In this study, we adopt the taxonomy of attacks based on the work by Lippmann et al. [9]. This taxonomy includes the following 4 classes of attacks:

Probe (scan attacks): active port exploitation for a known vulnerability of the corresponding service.

Denial of service (DoS): disruption of a host or network service by making a computer resource unavailable to its intended users.

Remote to Local (R2L): attempt to gain access in order to extract files and modify data in transit on the attacked machine.

User to root (U2R): elevation of privileges of a local user to ones normally reserved for the super user or the administrator.

In Figure 2, we show anti-goal models for attacks classified according to their category as per the categorization described above. We also show a goal model for a sample service (*files services*) being targeted by these attacks.

3 Log Data Representation

3.1 Storing Log Data

In a large system which is composed of many subsystems and where each subsystem is monitored by a different logging mechanism, it is possible that the logged data are represented in different formats and schemas. To address the problem of different log data formats and schemas, we consider mappings from the native schema of each logger into a common log schema. In general, most industry log data contain only a subset of this schema. Even though the identification of the mappings between the native log schema of each monitor component and the unified schema is outside of the scope of this paper, such mappings can be compiled using semi-automated techniques discussed in detail in the paper by Alexe et al. [2] or compiled manually by domain experts. For the purposes of this study, we have implemented the mappings manually, as tables. Figure 3 shows the mappings between the unified schema and the schema of two systems from the test environment. The unified schema we utilize for this work contains 10 fields (please see Table 1) classified into four categories: general, event specific, session information and environment related. This proposed schema represents a comprehensive list of data fields that we consider to be useful for diagnosis purposes.

4 Hypothesis Modeling

The framework proposed in this paper (Figure 1) aims at detecting malicious behavior against software systems. It consists of the following processes: goal and anti-goal models representation/compilation, log data storing, log filtering, ground atom generation and diagnostics.

4.1 Goal and Anti-Goal Model Annotations

The proposed framework is built on the premise that the monitored system′s requirements goal and anti-goal models are available by system analysts or can be reverse engineered from source code using techniques discussed by Yu et al. [16] and Lamsweerde [13]. In this paper, we consider that leafs in a goal model relate to operations of simple system components, that can be delivered as black boxes for the purposes of monitoring and diagnosis. Furthermore, for this work we adopt the formalizations for goal modeling as proposed by Giorgini et al. [5].

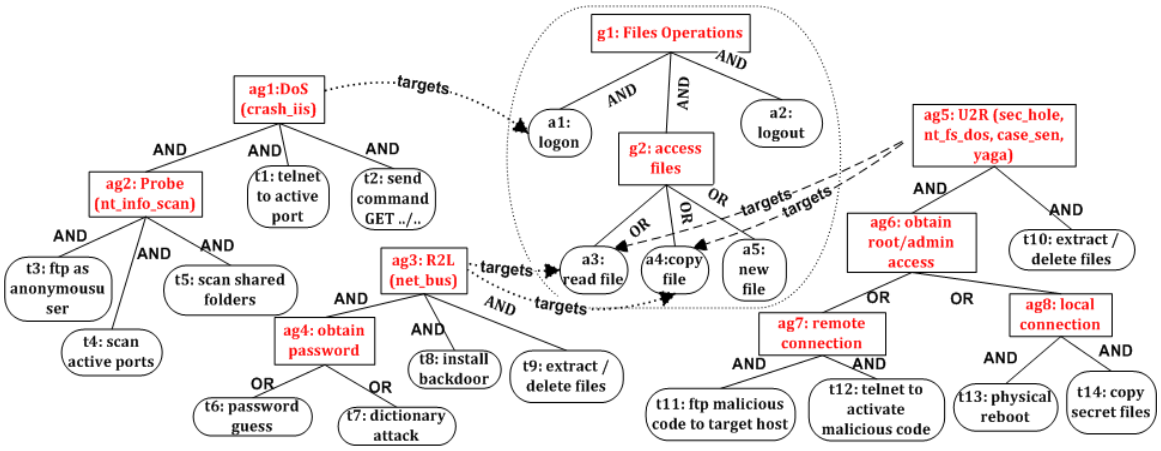

Figure 2: Anti-Goal Attacks against (g_1) the Goal Model for Files Services (center).

In this context, goal and anti-goal models used in this framework are extended by annotations in the form of SQL-like string pattern expressions such as the ones illustrated in Figure 6. These expressions describe the conditions under which goals/anti-goals instances can be fulfilled. In particular, tasks (leaf nodes) are associated with preconditions, occurrence and postconditions, while goals (non-leaf nodes) are associated with preconditions and postconditions only.

The annotations are expressed using string regular pattern expressions as described in the UNIX specifications [6]. An example of such an expression is *Windows.*starting.up* which is used to match strings that contain the word *Windows* followed by an unspecified number of characters, followed by the word *starting*, followed by a single unspecified character, followed by *up*. An annotation example for goal g_1 in Figure 2 is shown below:

*Description like 'Windows.*starting.up' AND Source_Component like 'hume.eyrie.af.mil'*

The matching process that aims to yield a subset of the logged data according to the annotation *i.e. query* is based either on string matching or on Latent Semantic Indexing (LSI) type of retrieval and is presented by Zawawy et al. [17]. The result of the matching (filtering) process is a list of *Matched_subsets[]* of logged events that relate to the specific annotation specification.

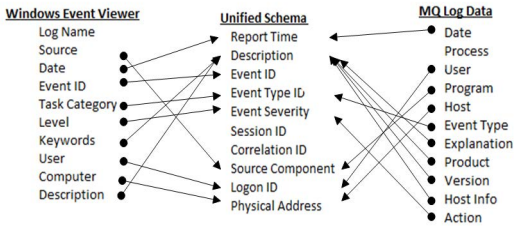

Figure 3: Mappings from Windows Event Viewer and IBM's MQ Log into Unified Schema.

4.2 Goal and Anti-Goal Models Predicates

We use first order logic to represent semantic information on goal/anti-goal models. A predicate is intensional if its truth value can only be inferred (i.e. cannot be directly observed). A predicate is extensional if its truth value can be directly observed. A predicate is strictly extensional if it can only be observed and not inferred for all its groundings [11].

We use the extensional predicates *ChildAND (parent_node, child_node)*, *ChildOR (parent_node, child_node)* to denote the AND/OR goal decomposition. Goals g_1 and g_2 shown in Figure 2 are examples of AND and OR decompositions. Similarly for anti-goals, ag_1 in Figure 2 represents an example of AND decomposed anti-goal.

We use the extensional predicates *Pre(node,*

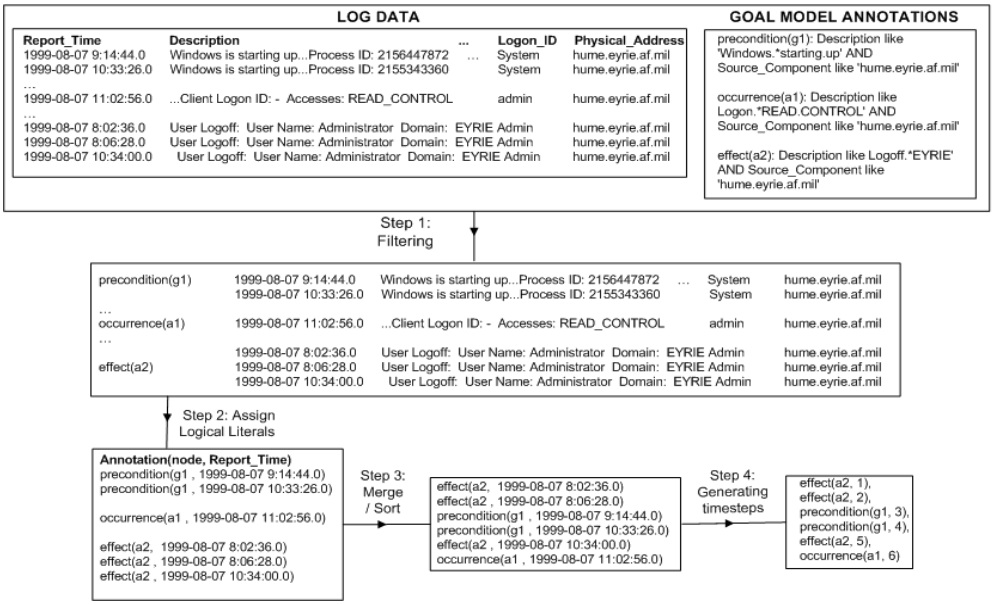

Figure 4: Transforming Filtered Log Data Subsets into Ground Atoms.

Table 1: Unified Schema for Log Data

Field Name	**Category**
Report_Time	General
Description	General
Event_ID	Event Specific
Event_Type_ID	Event Specific
Event_Severity	Event Specific
Session_ID	Session Information
Correlation_ID	Session Information
Source_Component	Environment Related
Logon_ID	Environment Related
Physical_Address	Environment Related

timestep), *Occ (node, timestep)*, and *Post (node, timestep)* to represent preconditions, tasks' occurrences and postconditions (respectively) at a certain timestep. We use the extensional predicate *Targets(anti-goal, node)* to denote the relationship between an anti-goal model and the targeted task or goal.

Finally, we use the intensional predicates *Occ(goal, timestep, timestep)* and *Satisfied(node, timestep)* to represent the goals occurrences and the goals/anti-goal/tasks satisfaction. The predicate *Satisfied* is predominantly intensional except for the top goal which satisfaction is observable (i.e. the observed system failure that triggers the root cause analysis process). If the overall service/transaction is successfully executed, then the top goal is considered to be satisfied, otherwise it is denied.

4.2.1 Ground Atoms Generation

In this section, we discuss the process to interpret and transform the (filtered) log subsets into a set of ordered ground atom literals. Figure 4 illustrates the basic steps of the transformation process with an example.

There are two inputs to this process: first, the log data stored in a common database as described in Section 3.1 and; second, the goal model for the monitored system with *precondition*, *postcondition* and *occurrence* patterns annotating each node of the model. The output is a totally timestep-ordered set of literals (ground atoms) of the form *literal(node,timestep)*.

Step 1: by applying the filtering and matching techniques outlined in section 4.1, we extract from the log database, a collection (*Matched_subsets[]*) of partially ordered log entries sets (the order is based on the timestamp in the *Report_time* field (please see Table 1)). Each subset corresponds to one annotation in

the goal model. An example of this process is shown in Figure 4 where each goal or task node annotation is associated with a set (can be empty) of log data produced by the matching process. In this respect, each annotation (*precondition*, *postcondition*, *occurrence*), is considered as a query applied to the log data base that returns a subset of the logged data that may relate to the specific annotation and goal model node. Once all the annotations for a goal model node return a non empty result, we consider that the corresponding goal model node has succeeded. An empty log data set means that no evidence can be found in the log data to indicate that this event has occurred during the observation period.

Step 2: for each *log_entry* in each *log_subset* in the set *Matched_subsets[]*, we create a literal in the form *annotation(node, log_entry.timestamp)*, where *annotation* represents *precondition*, *occurrence* or *postcondition*. An example based on the goal model in Figure 2 is to transform a log entry (*1999-08-07 1999-08-07 10:33:26.0 DARPA Security Windows is starting up ... Process ID: 2155070784*) that matches the pattern of the *precondition* of goal g_1 into an atom (*pre(g_1, 1999-08-07 10:33:26.0)*). As a result of this step, the subsets of log data are converted into subsets of literals (see step 2 in Figure 4).

Step 3: the subsets of ground atoms generated earlier are merged in one set representing all the matched log data from all the goal model nodes. A total ordering is accomplished by sorting all ground atoms based on their timestamps (see step 3 in Figure 4). To guarantee the uniqueness of each timestamp, each timestamp is appended with the corresponding process/session id.

Step 4: the timestamp in each atom is replaced by a timestep. The timestep is an integer initialized at 1, and incremented for each subsequent atom. Thus *precondition(g_1, 1999-08-07 10:33:26.0)* is transformed into *precondition(g_1,4)*.

The outcome of this process is a set of ordered ground atoms that are used in the subsequent inference phase as one source of system observation.

4.3 Goal Model Rules

Relationships in the goal model are also represented using first order logic expressions. Goals/tasks' satisfaction is expressed using the truth assignment of the *Satisfied(node)* predicate which, in turn, is inferred as follows: A task a with a precondition $\{Pre\}$ and a postcondition $\{Post\}$ is satisfied at time $t+1$ if and only if $\{Pre\}$ is true at time $t-1$ just before the task a occur at time t, and $\{Post\}$ is true at time $t+1$ (see Equation 1).

$$Pre(a, t-1) \wedge Occ(a, t) \wedge Post(a, t+1) \Rightarrow Satisfied(a, t+1) \quad (1)$$

A goal g with precondition $\{Pre\}$ and postcondition $\{Post\}$ is satisfied at time t_2 if and only if the goal occurrence finishes at time t_2, and $\{Pre\}$ is true when goal occurrence starts at t_1 where $(t_1 < t_2)$ and $\{Post\}$ is true when goal occurrence is completed at t_2 (please see Equation 2).

$$Pre(g, t_1) \wedge G\_Occ(g, t_1, t_2) \wedge Post(g, t_2)) \Rightarrow Satisfied(g, t_2) \quad (2)$$

The truth values of the predicate *Occ(goal)* (used in Equation 2) can only be inferred based on the satisfaction of all its children in the case of AND-decomposed goals (Equation 3) or at least one of its children in the case of OR-decomposed goals (Equation 4).

$$\forall a, Satisfied(a, t_1) \wedge ChildAND(g, a) \wedge (t_2 < t_1 < t_3) \Rightarrow Occ(g, t2, t3) \quad (3)$$

$$\exists a, Satisfied(a, t_1) \wedge ChildOR(g, a) \wedge (t_2 < t_1 < t_3) \Rightarrow Occ(g, t_2, t_3) \quad (4)$$

Contribution links of the form $node_1 \xrightarrow{++S} node_2$ are represented in Equation 5. (Similarly for *++D,--S,--D*).

$$Satisfied(node_1, t_1) \Rightarrow Satisfied(node_2, t_2) \quad (5)$$

As shown in Equations 3 and ORgoaloccurrence, the contribution of a child to its parent's satisfaction is a function of time. For instance, the occurrence (and consequently the satisfaction) of gaol g_1 at (timestep = 11) in Figure

2 is impacted by the satisfaction of task a_1 at (timestep = 3), goal g_2 at (timestep = 5) and task a_2 at (timestep = 7). If for example task a_1 is denied or satisfied at (timestep = 2) then this has no impact on the satisfaction of parent goal g_1 at (timestep = 7).

4.4 Anti-Goal Model Rules

Similarly to goal models, anti-goal models also consist of AND-OR decompositions, therefore anti-goal model relationships with their children nodes are represented using the same first order logic expressions as for goal models (Equations 3 and 4). The satisfaction of tasks (leaf nodes) in anti-goal models follows the same rule as in Equation 1. The satisfaction of anti-goals is expressed using the truth assignment of the *Satisfied(node)* predicates which, in turn, is inferred as follows:

$$Pre(ag_1, t_1) \wedge Occ(ag_1, t_1, t_2) \wedge Post(ag_1, t_2) \Rightarrow Satisfied(ag_1, t_2) \quad (6)$$

The satisfaction of an anti-goal ag_1 at timestep t_1 presents a negative impact on the satisfaction of a target task a_i if ag_1 occurred just before the denial of the target task a_i

$$Satisfied(ag_1, t_1) \wedge Targets(ag_1, a_i) \wedge (t_1 < t_2) \Rightarrow !Satisfied(a_i, t_2) \quad (7)$$

4.5 Uncertainty Representation

The proposed framework relies on log data as evidence for proving the occurrence of the events of interest. However, the selection of events of interest (e.g. using LSI) is not perfect and is plagued by False Positive and False Negative entries. For this reason, we allow for user-defined weights representing the confidence a domain expert has on an observation for a given case. We address uncertainty in observations using a combination of logical and probabilistic models as follows:

A) The domain knowledge representing the interdependencies between systems/services is modeled using weighted first order logic statements. The strength of each relationship is represented with a real-valued weight set based on domain knowledge and learning from a training log data set and reflects. The weight of each rule represents our confidence in this rule relative to the other rules in the knowledge base. Consequently, the probability inferred for each atom depends on the weight of the competing rules where this atom occurs. For instance, the probability of the satisfaction of task a_2 in Figure 2 (*Satisfied(a_4,t)*) is inferred based on the Equation 8 with weight w_1,

$$w_1 : \ Pre(a4, t) \wedge Occ(a4, t+1) \wedge Post(a4, t+2) \Rightarrow Satisfied(a4, t+2) \quad (8)$$

On the other hand, the contribution link based on Equation 9 with weight w_2 contributes to the denial of task a_4,

$$w2 : \ !Satisfied(g_2, t_1) \Rightarrow !Satisfied(a_4, t_2) \quad (9)$$

Consequently, the probability assignment given to $Satisfied(a_4, t)$ is determined by the rules containing it as well as the weight of these rules in MLN inferencing.

B) Applying an open world assumption to the observation where a lack of evidence does not absolutely negate an event's occurrence but rather weakens its possibility.

4.6 Weight Learning and Markov Logic Network Construction

Weight learning for rules and observations is done semi-automatically by first using discriminative learning based on a training set of past cases as discussed in the paper by Domingos [4] and then manually refined by a system expert. During automated weight learning, each formula is converted to CNF, and a weight is learned for each of its clauses from past cases. The weight of a clause is used as the mean of a Gaussian prior for the learned weight. The learned weight can be modified to reflect our confidence in the rules. For example, in Figure 2, a rule indicating that the denial of the top goal g_1 implies that at least one of its AND-decomposed children (a_1, g_2, and a_2) have been denied, should be given higher weight than a rule indicating that g_1 is satisfied based on log data showing that *pre(g_1)*, *occ(g_1)* and *post(g_1)* are true. This example relates to the case where he administrator has without any doubt

witnessed the failure of the system, even if the observed log data do not agree (due to missing or inaccurate data). Consequently, the Markov Logic Network is constructed using an exhaustive scheme of rules and predicates as discussed earlier as well as, grounding predicates with all possible values, and connecting them if they coexist in a grounded formula.

5 Diagnostics

The flowchart in Figure 5 represents the process of detection of malicious behavior using the proposed framework. This process starts when an alarm is raised indicating a system failure, and completes when a list of root causes of the observed failure is identified. In order for the diagnostic reasoning process to proceed, we consider that the annotated goal/anti-goal models for the monitored systems and their corresponding Markov networks and logical predicates have been generated as described in section 4. The diagnostic reasoning process is composed of seven steps as discussed in detail below:

Step 1: The investigation starts when an alarm is raised or when the system administrator observes the failure of the monitored service, which is the denial of a goal g.

Step 2: Based on the knowledge base KB that consists of all goal models and all observations (filtered log data and visual observation of the failure of the monitored service), the framework constructs a Markov Network that in turn is applied to generate a probability distribution for all the *Satisfied(node, timestep)* predicates for all nodes in the goal tree rooted at the denied goal g and the nodes of all other connected to it trees. This probability distribution is used to indicate the satisfaction or denial of tasks and goals at every timestep of the observation period. More specifically, if the probability of satisfaction of a task a_i at a timestep t is higher than α then we conclude that a_i is satisfied at t (i.e. $Satisfied(a_i, t) > \alpha \Rightarrow a_i$ is satisfied at t). Typically the timestep t starts at 0, and α is chosen to be 0.5. An example of the outcome of this step is the following based on Figure 2:

$$\begin{cases} Satisfied(a_1, 3) : 0.36 \\ Satisfied(a_3, 7) : 0.45 \end{cases} \Rightarrow \begin{cases} 1 : a_1 \\ 2 : a_3 \end{cases}.$$

Step 3: The outcome of the previous step is a ranked list of denied tasks/goals where the first task represents the node that is most likely to be the cause of the failure. In step 3, the framework iterates through the denied list of tasks identified in step 2, and loads the anti-goal models targeting each of the denied tasks.

Step 4: for each anti-goal model selected in Step 3, we generate the observation literals from the log data by applying the ground atoms generation process described in Section 4.2.1. Note that a task may have zero, one or multiple anti-goal models targeting it. The goal model in Figure 2 contains task a_2 which has no anti-goal models targeting it, task a_1 which has one anti-goal (ag_1) targeting it, and task a_3 which has two anti-goals (ag_3 and ag_5) targeting it.

Step 5: this step is similar to step 2 but instead of evaluating a goal model, the framework evaluates the anti-goal model identified in step 4 using the log data-based observation and determines if the top anti-goal is satisfied, i.e., has been successfully executed. This is done by constructing a Markov Network based on the anti-goal model relationships (see section 4.4) and inferring the probability distribution for the *Satisfied(anti-goal, timestep)* predicate for the nodes in the anti-goal model. In particular, we are interested in the probability assigned to $Satisfied(ag_j, t)$ where t represents the timestep at which ag_j is expected to complete. Using the example in Figure 2, ag_1 takes 11 steps to complete execution, while ag_3 takes 7 steps. In this case, we are interested in $Satisfied(ag_1, 11)$ and $Satisfied(ag_2, 7)$.

Step 6: Based on the value of $Satisfied(ag_j, t)$ identified in step 5, we distinguish the following two cases:

1. If $Satisfied(ag_j, t) >= \alpha$ (typically $\alpha = 0.5$): this indicates that ag_j is likely to have occurred and caused a_i to be denied. In this case, we add this information (anti-goal and its relationship with the denied task) to the knowledge base KB_g. This knowledge base enrichment is done at two levels: first, a new observation is added to indicate that anti-goal ag_j has occurred; second, equation 7 is added to the set of rules.
2. If $Satisfied(ag_j, t) < \alpha$: this indicates

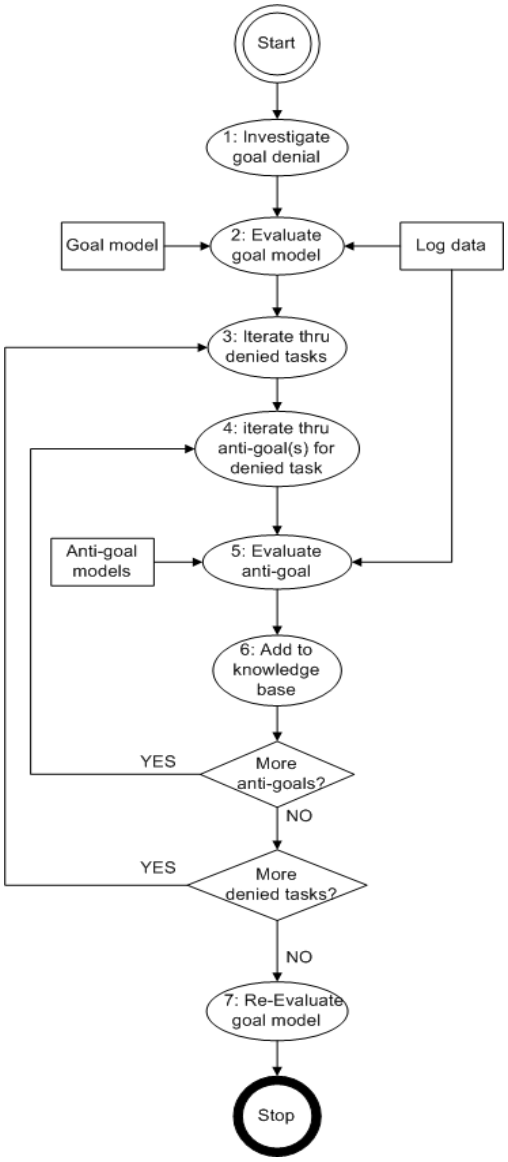

Figure 5: Malicious Behavior Detection Process

that ag_j did not occur, thus we exclude it as a potential cause of failure for a_i.

Before getting to next step, the framework checks whether the denied task a_i is targeted by any other anti-goal models. If so, it iterates through the list of other anti-goals ag_j that are targeting a_i by going back to Step 4. Once all anti-goals targeting a denied task a_i are evaluated, the framework checks whether there are more denied tasks and if so, it repeats step 2.

Step 7: Based on the new knowledge acquired from evaluating the anti-goal models, we re-evaluate the satisfaction of tasks based on the enriched knowledge based KB, and produce a new ranking of the denied tasks. The fact that an attack is likely to have occurred targeting a specific task that has failed increases the chances that this task is actually denied due to the intrusion leading to the overall system failure. This overall process helps improve the diagnosis but also provides interpretation for the denial of tasks.

6 Experimental Evaluations

To evaluate the proposed framework, we conducted the following set of experiments. The first case study aims examine the diagnostic effectiveness of the framework using a sample application utilizing COTS components. The objective of the second set of experiments is to evaluate the scalability and measure the performance of the framework using large data sets.

The framework's filtering and interpretation components were implemented using the Java Programming language (version 1.5). The diagnosis/inference was conducted using the Alchemy system developed at the University of Washington [7]. We also used Microsoft SQL Server 2008 to host the log database.

6.1 Credit Service Scenario

In the first case study for the proposed framework, we use the monitoring and detection of attacks on the execution of a credit evaluation business process implemented by a number of different services and COTS components.

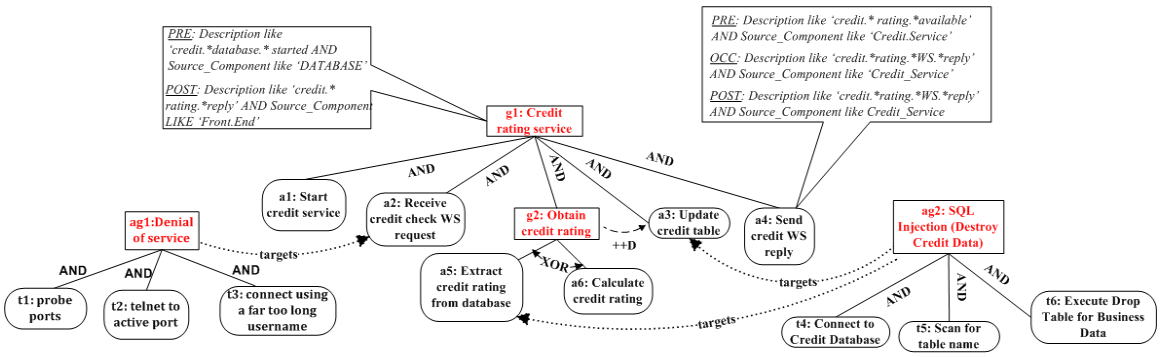

Figure 6: Goal and Anti-Goal Models for the Credit History Service

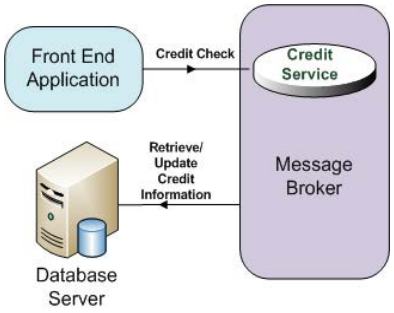

Figure 7: Layout of the Test Distributed Environment

6.1.1 Experiment Setup

The test environment (Figure 7) includes 4 systems/services: the front end application (soapUI), a message broker (IBM WebSphere Message Broker v7.0), the credit check Web Service and an SQL server (Microsoft SQL Server 2008).

The credit evaluation process starts upon receiving a Web Service request. If already available, the credit rating of the applicant is retrieved from the credit history table. For new applicants, the score is calculated and used to update the credit table. A Web Service reply containing the credit score is sent back to the front end application. In case one (or more) of the systems supporting the credit service fails, requests sent from the front end application will get no reply and will time out.

The experiment is conducted from the perspective of the system administrator where a system failure is reported and an investigation is triggered. The case study scenario consists of running an attack while executing the credit history service (see the goal model of the credit service in Figure 6) that is deployed on the distributed system shown in Figure 7 and described in section 6.1.

The anti-goal ag_1 is executed by first probing the active ports on the targeted machine, and then the attacker attempts to login but uses a very long username (16000 characters) disrupting the service authentication process, and denying legitimate users from accessing the service. Traces of this attack can be obtained in the Windows Event Viewer.

The anti-goal ag_2 models an SQL injection attack that aims at dropping the table containing the credit scores. One way to implement such an attack is to send two subsequent legitimate credit history web service requests that contain embedded malicious SQL code within a data value. In particular, we embed an SQL command to scan and list all the tables in the data value of the field *ID* as follows:

<ID>12345; select from SYSOBJECTS where TYPE = 'U' order by NAME</ID>

The system extracts the credential data value (e.g. ID) and uses it to query the database thus inadvertently executing the malicious code. The scan is followed by a second WS request that contains a command to drop the Credit_History table:

<ID>12345; Drop Table Credit_History </ID>

Traces of the credit service sessions are found in the SQL Server audit log data and the message broker (hosting the credit service). The

first anti-goal ag_1 represents the attacker's plan to deny access to the credit service to keeping busy the port used by that service. The second anti-goal ag_2 aims at injecting an SQL statement that destroys the credit history data.

6.1.2 Experiment Enactment

We run a sequence of credit service requests and in parallel we perform an SQL injection attack. The log database table contained 1690 log entries generated from all systems in our test environment.

The first step in the diagnosis is to use the filtered log data to generate the observation (ground atoms) in the form of Boolean literals representing the truth values of the node annotations in the goal (or anti-goal) model during the time interval where the log data is collected. The *Planfile1* model segment below represents the observation corresponding for one credit service execution session:

Planfile1: *Pre(g_1 , 1); Pre(a_1, 1); Occ(a_1, 2); Post(a_1, 3); Pre(a_2, 3); Occ(a_2, 4); Post(a_2, 5); Post(g_2, 5); ?Pre(a_5, 5); ?Occ(a_5, 6); ?Post(a_5, 7); Pre(a_6, 5); ?Occ(a_6, 6); ?Post(a_6, 7); ?Post(g_2, 7); ?Pre(a_3, 7); ?Occ(a_3, 8); ?Post(a_3, 9); ?Pre(a_4, 9); ?Occ(a_4, 10); ?Post(a_4, 11); ?Post(g_1, 11), !Satisfied(g_1, 11)*

In the case where we don't find evidence for the occurrence of the events corresponding to goals/anti-goals/tasks execution (precondition, postcondition, etc.), we don't treat this as a proof that these events did not occur but we consider them as rather uncertain. We represent this uncertainty by preceding the corresponding ground atoms with interrogation mark (?). In cases where there is evidence that an event did not occur, the corresponding ground atom is preceded with an exclamation mark (!).

By applying inference based on the observation in *Planfile1* (Step 2 in flowchart in Figure 5), we obtain the following probabilities for the ground atoms using the Alchemy tool:

$Satisfied(a_1, 3)$:0.99, $Satisfied(a_2, 5)$:0.99, $Satisfied(a_5, 7)$:0.30, $Satisfied(a_6, 7)$:0.32, $Satisfied(a_3, 9)$:0.34, $Satisfied(a_4, 11)$:0.03

Using step 2 in the diagnostics flowchart (Figure 5), we deduce that a_4, a_5, a_6 and a_3 are the root causes for the failure of the top goal g_1.

Using step 3 and 4 in the diagnostics flowchart, we first generate the observation for anti-goal ag_1 (*Planfile2*) and apply inference based on the anti-goal model ag_1 relationship and the generated observation.

Planfile2: *Pre(ag_1, 1), ?Pre(t_1, 1), ?Occ(t_1, 2), ?Post(t_1, 3), ?Pre(t_2, 3), ?Occ(t_2, 4), ?Post(t_2, 5), ?Pre(t_3, 5), ?Occ(t_3, 6), ?Post(t_3, 7), ?Post(ag_1, 7)*

In Step 5, the outcome of the inference indicates that ag_1 is denied (satisfaction probability is 0.0001). We iterate through step 3, and the next denied task a_5 is targeted by anti-goal ag_2. The observation generated for the anti-goal ag_2 is in *Planfile3* below:

Planfile3: *Pre(ag_2, 1), Pre(t_4, 1), Occ(t_4, 2), Post(t_4, 3), Pre(t_5, 3), Occ(t_5, 4), Post(t_5, 5), Pre(t_6, 5), Occ(t_6, 6), Post(t_6, 7), Post(ag_2, 7)*

The outcome of the inference indicates that ag_2 is satisfied (satisfaction probability is 0.59). Using step 6, we add the result to the goal model knowledge base. In step 7, we re-evaluate the goal model based knowledge base and generate a new diagnosis as follows:

$Satisfied(a_1, 3) : 0.95$, $Satisfied(a_2, 5) : 0.98$, $Satisfied(a_5, 7) : 0.23$, $Satisfied(a_6, 7) : 0.29$, $Satisfied(a_3, 9) : 0.11$, $Satisfied(a_4, 11) : 0.26$

The new diagnosis ranks a_3 as the most likely root cause for the failure of the top goal g_1. Next, a_5, a_4 and a_6 are also possible root causes but less likely. This new diagnosis is an improvement over the first one generated in step 2 since it accurately indicates a_3 as the most probable root cause. We know that this is correct since the SQL injection attack ag_2 targeted and dropped the credit table.

6.1.3 Inference Case Study

A case study of the inference was performed using Ubuntu Linux running on Intel Pentium

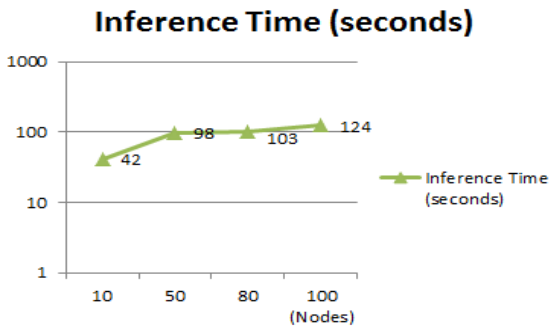

Figure 8: Impact of goal model size on inference time.

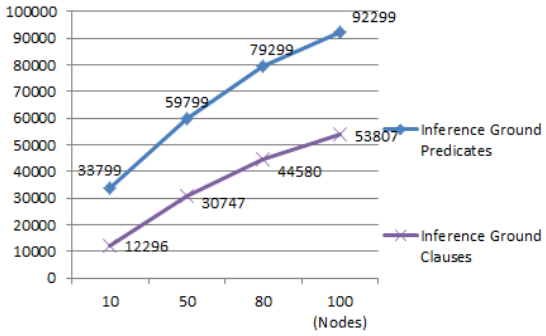

Figure 9: Impact of goal model size on ground atoms/clauses for inference.

2 Duo 2.2 GHz machine. We used 3 sets of extended goal models representing the credit service goal model. The 3 extended goal models contained 50, 80 and 100 nodes respectively. In addition, we used a set of 5 anti-goal models with a total of 40 anti-goals nodes. In particular, we are interested in measuring the impact of the size of goal and anti-goal models on the inference component. Figure 8 illustrates that the number of ground atoms/clauses, which directly impacts the size of the resulting Markov model, is linearly proportional to the goal/anti-goal models total number of nodes. Figure 8 shows that the inference time ranged from 42 seconds for a goal model of 50 nodes (10 goal nodes and 40 anti-goal nodes), up to 124 seconds for a model of 140 nodes (100 goal nodes and 40 anti-goal nodes). These results show that our approach can be applied to industrial software applications with small to medium-sized requirement models.

6.1.4 Threshold Alpha Impact

In the experiment above, we set the threshold value α as 0.5. In this experiment, we vary the value of the threshold α and analyze the impact on the diagnostic process, in particular, on steps 2 and 5 (Figure 5).

In the first case, we set α to 0.75. This has no impact on step 2, but it impacts step 5. Since both the anti-goals ag_1 and ag_2 have a probability of satisfaction below 0.75 and then they are considered to have not occurred and have no impact on the goal model evaluation. the result is that step 7 does not cause any modification to the original diagnosis.

In the second case, we set α to 0.25. The α value change impacts the decision on what tasks are denied (step 2). In this case, the diagnosis in step 2 indicates that only a_4 is denied. Similarly, a_4 is targeted by ag_1 which has no trace in the observation and thus has no impact on the overall diagnosis process. The final result is that the diagnosis in step was not modified in step 7.

As shown in the two experiments above, the threshold parameter α affects the overall diagnosis process and has to be tuned accordingly either by the user or by considering past cases.

6.2 Performance and Scalability

The second experiment evaluates the performance and scalability of the framework. We use the publicly available DARPA (Advanced Research Projects Agency) intrusion data sets, which is collected and provided by the IST Group at MIT [9]. In particular, we use the Windows NT Data Set for DARPA 2000 [8]. When stored in our common log database, the data set amounted to 153829 entries and took up to 600 MB of disk space. We use annotated anti-goal models to model the set of the attacks (see Figure 2) that are known to have traces in the DARPA 2000 Windows NT data set (see Table 2).

6.2.1 Filtering Performance

This experiment measures the performance of log filtering. To do so, we apply the detection framework on the NT event viewer log data in

Table 2: Performance Measures for Normalization and Filtering (Milliseconds)

	Normalization	LSI Filtering	Grep Filtering
Processing Time (145855 entries)	178422	708876	181299
Average Time per Log Entry	1.22	4.86	1.24

the DARPA dataset, and measure the time required (in milliseconds) to perform the computationally demanding steps mainly the normalization and filtering.

First, the normalization process parses the log data files which is in the comma separated value format (CSV) which was exported from the Windows NT event viewer (evtx) format. Next, the log data is transformed into the unified schema format and stored in the log database. Last step consists of indexing the log table based on the timestamp field.

Second, the filtering was done using two techniques: Grep and LSI. As expected, the performance of the filtering process varied depending on the technique used. The first step in the filtering is common for the two approaches and it consists of parsing the goal/anti-goal models stored as local XML files, and collecting the nodes and their associated queries. For Grep based filtering, the second step is to transform the collected queries into regular expressions and find matching log entries. Based on the processing rates shown in Table 2, the framework can handle few hundred log entries per second thus it scales up to an industrial environment.

7 Conclusions

This paper discussed a root cause analysis technique that can be used to identify the causes of failures induced by third party actions. The technique is based first, on modeling with goal models the conditions, constraints, tasks, and actions that are required for the system to deliver its functionality, second on modeling with anti-goal models the different ways an intrusive third party action can be manifested, third on a knowledge base that represents as rules and facts the aforementioned models and the logged events and, fourth a probabilistic reasoning technique that is based on Markov Logic Networks, that allows for the evaluation of possible and alternative ways to explain the failed behavior of the system. More specifically, the proposed approach upon the observation of a failure not only computes the alternative ways this failure is supported but also, computes whether these alternative ways can be manifested as results of known intrusive actions. Consequently, the proposed approach best fits to detect the class of known (as opposed to mutating or new) intrusive behavior that can be documented using anti-goal AND-OR tree model. The probabilistic reasoning framework increases the probability of an identified root cause when the corresponding root cause can be supported by an intrusive behavior. The framework has been evaluated in a medium size system that is composed of COTS components, and in the identification of known intrusions in the DARPA intrusion dataset. The results indicate that the approach can be used for root cause analysis in medium size systems and for systems that generate a few hundred events per second. Future work involves the extension of the log filtering process so that the framework will have to examine fewer events without losing accuracy, and performance enhancements required to deal with systems that generate a few thousand events per second. The work is supported by an NSERC CRD grant and the CA Labs of CA Technologies.

References

[1] Safaa O. Al-Mamory and Hongli Zhang. Intrusion detection alarms reduction using root cause analysis and clustering. *Comput. Commun.*, 32(2):419–430, 2009.

[2] Bogdan Alexe, Laura Chiticariu, Renee J. Miller, and Wang Chiew Tan. Muse: Map-

ping understanding and design by example. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancn, Mxico*, pages 10–19, 2008.

[3] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. In *Science of Computer Programming*, pages 3–50, 1993.

[4] Pedro Domingos. Real-world learning with markov logic networks. In Jean-Franois Boulicaut, Floriana Esposito, Fosca Giannotti, and Dino Pedreschi, editors, *Machine Learning: ECML 2004*, volume 3201 of *Lecture Notes in Computer Science*, pages 17–17. Springer Berlin / Heidelberg, 2004.

[5] Paolo Giorgini, John Mylopoulos, Eleonora Nicchiarelli, and Roberto Sebastiani. Reasoning with goal models. pages 167–181. Springer, 2002.

[6] The Open Group. "regular expressions". the single unix specification, version 2. http://pubs.opengroup.org/onlinepubs/007908799/xbd/re.htm, 1997.

[7] S. Kok, M. Sumner, M. Richardson, P. Singla, H. Poon, D. Lowd, and P. Domingos. The alchemy system for statistical relational ai. technical report, dept. of computer science and engineering, university of washington, seattle, wa. http://alchemy.cs.washington.edu, 2007.

[8] Jonathan Korba and Arthur C. Smith. Windows nt attacks for the evaluation of intrusion detection systems. Master's thesis, MIT, 2000.

[9] Richard Lippmann, Joshua Haines, David Fried, Jonathan Korba, and Kumar Das. Analysis and results of the 1999 darpa off-line intrusion detection evaluation. In Herv Debar, Ludovic M, and S. Wu, editors, *Recent Advances in Intrusion Detection*, volume 1907 of *Lecture Notes in Computer Science*, pages 162–182. Springer Berlin / Heidelberg, 2000.

[10] Vitor Estevao Silva Souza and John Mylopoulos. Monitoring and diagnosing malicious attacks with autonomic software. In Alberto H. F. Laender, Silvana Castano, Umeshwar Dayal, Fabio Casati, and Jos Palazzo Moreira de Oliveira, editors, *ER*, volume 5829 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2009.

[11] Son Dinh Tran and Larry S. Davis. Event modeling and recognition using markov logic networks. In *European Conference on Computer Vision*, pages 610–623, 2008.

[12] A. Van Lamsweerde, R. Darimont, and P. Massonet. Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. In *RE '95: Proceedings of the Second IEEE International Symposium on Requirements Engineering*, page 194, Washington, DC, USA, 1995. IEEE Computer Society.

[13] A. van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *Software Engineering, IEEE Trans. on*, 26(10):978–1005, 2000.

[14] Axel van Lamsweerde. Elaborating security requirements by construction of intentional anti-models. *Software Engineering, International Conference on*, 0:148–157, 2004.

[15] Yiqiao Wang, Sheila A. McIlraith, Yijun Yu, and John Mylopoulos. An automated approach to monitoring and diagnosing requirements. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 293–302, New York, NY, USA, 2007. ACM.

[16] Yijun Yu, Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Julio Leite. From goals to high-variability software design. pages 1–16, 2008.

[17] Hamzeh Zawawy, Kostas Kontogiannis, and John Mylopoulos. Log filtering and interpretation for root cause analysis. In *ICSM '10: Proceedings of the 26th IEEE International Conference on Software Maintenance*, 2010.