Requirements-Driven Root Cause Analysis Using Markov Logic Networks

Hamzeh Zawawy[1], Kostas Kontogiannis[2], John Mylopoulos[3], and Serge Mankovskii[4]

[1] University of Waterloo, Ontario, Canada
hzawawy@engmail.uwaterloo.ca
[2] National Technical University of Athens, Greece
kkontog@softlab.ntua.gr
[3] University of Toronto, Ontario, Canada
jm@cs.toronto.edu
[4] CA Labs, Ontario, Canada
Serge.Mankovskii@ca.com

Abstract. Root cause analysis for software systems is a challenging diagnostic task, due to the complexity emanating from the interactions between system components and the sheer size of logged data. This diagnostic task is usually assisted by human experts who create mental models of the system-at-hand, in order to generate hypotheses and conduct the analysis. In this paper, we propose a root cause analysis framework based on requirement goal models. We consequently use these models to generate a Markov Logic Network that serves as a diagnostic knowledge repository. The network can be trained and used to provide inferences as to why and how a particular failure observation may be explained by collected logged data. The proposed framework improves over existing approaches by handling uncertainty in observations, using natively generated log data, and by providing ranked diagnoses. The framework is illustrated using a test environment based on commercial off-the-shelf software components.

Keywords: goal model, markov logic networks, root cause analysis.

1 Introduction

Software root cause analysis (RCA) is the process by which system administrators analyze symptoms in order to identify the faults that have led to system application failures. More specifically, for systems that comprise of a large number of components encompassing complex interactions, root cause analysis may require large amounts of system operation data to be logged, collected and analyzed. It is estimated that forty percent of large organizations generate more than one terabyte of log data per month, whereas eleven percent of them generate more than ten Terabytes of log data monthly [10].

In this context, and in order to maintain the required service quality levels, such IT systems must be constantly monitored and evaluated by analyzing complex logged data emanating from diverse components. However, the sheer size

J. Ralyté et al. (Eds.): CAiSE 2012, LNCS 7328, pp. 350–365, 2012.

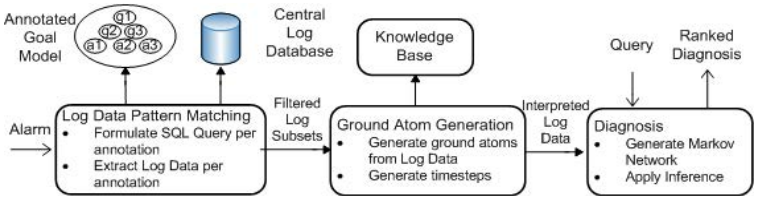

Fig. 1. Diagnostic Process

of such logged data often makes human analysis intractable and consequently, requires the use of algorithms and automated processes.

For this paper, we adopt a hybrid approach based on modeling the diagnostic knowledge as goal trees and on a probabilistic reasoning methodology based on Markov Logic Networks (MLNs). More specifically, the process is based on three main steps as illustrated in Fig. 1. In the first step, the diagnostic knowledge is denoted as a collection of goal models that represent how specific functional and non-functional system requirements can be achieved. Each goal model node is consequently annotated with *pre-condition*, *post-condition* and *occurrence* patterns. These patterns are used to filter and generate subsets of the logged data in order to increase the performance of the diagnostic process. The second step of the process is based on the use of a diagnostic rule knowledge base that is constructed by the goal models and the generation of ground atoms that are constructed by the logged data. The third and final step of the process is based on the use of the knowledge base and the ground atoms to reason on the validity of hypotheses (queries) that are generated as a result of the observed symptoms.

This paper is organized as follows. Section 2 covers the research baseline. Section 3 presents a motivating scenario. Section 4 describes the architecture and processes in the proposed framework. A case study is presented in section 5. Section 6 reviews related work. The conclusions are in section 7.

2 Research Baseline

2.1 Goal Models

Goal models have been for long proven to be effective in capturing large numbers of alternative sets of low-level tasks, operations, and configurations that can fulfill high-level stakeholder requirements [15]. Fig. 2 depicts a (simplified) goal model for a loan application. A goal model consists of goals and tasks. Goals−the squares in the diagram−are defined as states of affairs or conditions that one or more actors would like to achieve, e.g., *loan evaluation*. On the other hand, tasks−the oval shapes−describe activities that actors perform in order to fulfill their goals, e.g., *update loan database*.

Goals and tasks can impact each other's satisfaction using contribution links: *++S, --S, ++D, --D*. More specifically, given two goals G_1 and G_2, the link

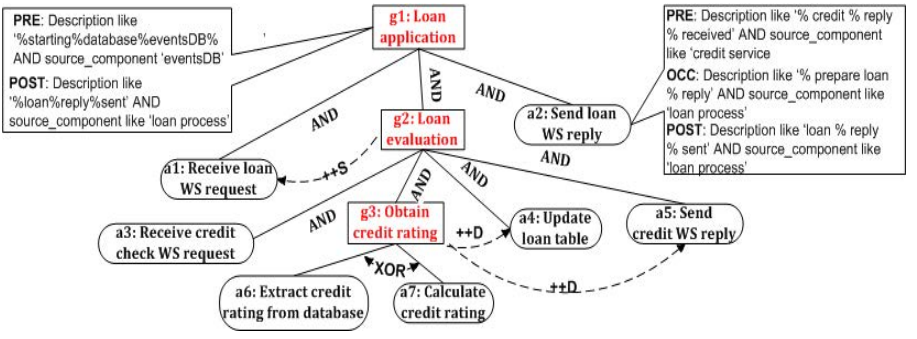

Fig. 2. Loan Application Business Process Goal Model

$G_1 \xrightarrow{++S} G_2$ (respectively $G_1 \xrightarrow{--S} G_2$) means that if G_1 is satisfied, then G_2 is satisfied (respectively denied). The meaning of links $++D$ and $--D$ are dual w.r.t. to $++S$ and $--S$ respectively, by inverting satisfiability and deniability. The class of goal models used in our work has been originally formalized in [6], where appropriate algorithms have been proposed for inferring whether a set of root-level goals are satisfied or not.

For this paper, we use the loan application goal model originally presented in [16] as a running example to better illustrate the inner workings of the proposed approach (see Fig. 2. The example goal model contains three goals (rectangles) and seven tasks (circles). The root goal g_1 (loan application) is AND-decomposed to goal g_2 (loan evaluation) and tasks a_1 (Receive loan web service request) and a_2 (Send loan web service reply), indicating that goal g_1 is satisfied if and only if goal g_2, tasks a_1 and a_2 are satisfied, and so on. Also, the contribution link $++D$ from goal g_3 to tasks a_4 and a_5 indicates that if g_3 is denied then tasks a_4 and a_5 should be denied as well. Similarly, the contribution link $++S$ from g_2 to task a_1 indicates that if g_2 is satisfied then so must be a_1.

Furthermore, as illustrated in Fig. 2, we annotate goals and tasks with *precondition*, *occurrence* and *postcondition* expressions. A task/goal is satisfied if it *has* occurred and its preconditions (postconditions) are true before (respectively after) its occurrence. Occurrence as well as, preconditions and postconditions are evaluated by a pattern matching approach that is discussed in detail in [17]. In practice, finding the expressions used in the annotations is done by users that are familiar with the monitored systems and the contents of the generated log data, or can be generated by data analysis techniques such as data mining and generalised sequence pattern algorithms. The techniques have been presented in the literature and are outside the scope of this paper. Experimentally, we followed an iterative approach in finding the expressions for the annotations of the goal model in Fig. 2. This process consists of assigning expressions, generating and evaluating the observation. When false positives are identified, the corresponding expression(s) are updated accordingly, and the process is restarted.

2.2 Markov Logic Networks

MLNs have been recently proposed as a way of providing a framework that combines first order logic and probabilistic reasoning [5]. A knowledge base consisting of first-order logic formulae represents a set of hard constraints on the set of possible worlds that it describes. In probabilistic terms, if a world violates one formula, it has zero probability of being an interpretation of the knowledge base. Markov logic softens these constraints by making a world that violates a formula to be less probable but still, possible. The more formulas a world violates, the less probable it becomes. In MLNs, each logic formula F_i is associated with a positive real-valued weight w_i. Every grounding (instantiation) of F_i is given the same weight w_i. In this context, a Markov Network is an undirected graph that is built by an exhaustive grounding of the predicates and formulas as follows:

- Each node corresponds to a ground atom x_k (an instantiation of a predicate).
- If a subset of ground atoms $x_{\{i\}} = \{x_k\}$ are related to each other by a formula F_i with weight w_i, then a clique (subset of the graph where each two vertices in the subset are connected by an edge) C_i over these variables is added to the network. C_i is associated with a weight w_i and a feature function f_i defined as follows,

$$f_i(x_{\{i\}}) = \begin{cases} 1 & F_i(x\{i\}) = True, \\ 0 & otherwise \end{cases} \tag{1}$$

First-order logic formulae serve as templates to construct the Markov Network. Each ground atom, X, represents a binary variable in a Markov network. The overall network is then used to model the joint distribution of all ground atoms. The corresponding global energy function can be calculated as follows,

$$P(X = x) = \frac{1}{Z} \exp(\sum_i w_i f_i(x_{\{i\}})) \tag{2}$$

where Z is the normalizing factor calculated as,

$$Z = \sum_{x \in X} \exp(\sum_i w_i f_i(x_{\{i\}})) \tag{3}$$

where i denotes the subset of ground atoms $x_{\{i\}}$ that are related to each other by a formula F_i. The Markov network can then be used to compute the marginal distribution of events and perform inference. Since inference in Markov networks is #P-complete, approximate inference is proposed to be performed using the Markov chain Monte Carlo (MCMC), and the Gibbs sampling. More details on the Markov Logic Networks and their use can be found in [11].

3 Motivating Scenario

We use as a running example the RCA for a failed execution of a loan evaluation business process implemented by a service oriented system. The normal

execution scenario for the system starts upon receiving a loan application in the form of a Web Service request. The loan applicant′s information is extracted and used to build another request that is sent to a credit evaluation Web Service. The credit rating of the applicant is returned as Web Service reply. Based on the credit rating of the loan applicant, a decision is made on whether to accept or reject the loan application. This decision is stored in a table before a Web Service reply is sent back to the front end application.

The requirements of the *Loan application* process are modeled in the goal tree illustrated in Fig. 2. For our running example we consider that the operator observes the failure of the top goal g_1 of the goal model. Surprisingly, even with this relatively simple scenario, the relationship between failures and their faults is not always obvious due to cascading errors and incomplete log data as well as due to intermittent connection errors, incorrect data entries that are hard to debug when large number of requests are processed during a short time.

4 Root Cause Analysis Framework

4.1 Building a Knowledge Base

In this section, we discuss the first component of the framework which consists of a process of building a diagnostic knowledge base from goal models.

Goal Model Annotations. We extend the goal models by annotating the goal model's nodes with additional information on the events pertaining to each of these nodes. In particular, tasks (leaf nodes) are associated with *pre-condition*, *occurrence* and *post-condition* patterns, while goals (non-leaf nodes) are associated with pre-condition and post-condition patterns only. These annotations are expressed using string pattern expressions of the form,

[*not*] *column_name* [*not*] *LIKE* "*match_string*"

where *column_name* represents a field name in the log database and *match_string* can contain the following pattern matching symbols:

- %: Matches strings of zero or many characters.
- Underscore (_): Matches one character.
- [...]: enclose sets or ranges, such as $[abc]$ or $[a-d]$.

These symbols are based on the SQL Server 2008 R2 specifications [9] so that goal model annotations can readily be usable in an SQL query. Moreover, we adopt all the predicates from the SQL specifications such as *LIKE* and *CONTAINS*. With respect to our running example, an annotation is the precondition for goal g_1 (Fig. 2) shown below:

Pre(g_1): Description LIKE '%starting%Database%eventsDB%' AND source_component LIKE 'eventsDB'

This annotation example is considered as a pre-condition pattern for goal g_1 and succeeds when it matches event traces generated by the *eventsDB* database system where trace logs have a description text containing the keyword *starting*, followed by space, then followed by the keyword *Database*, then space then followed by the keyword *eventsDB* (see Fig. 2 for more annotation examples).

Goal Model Predicates. We represent the states and actions of the monitored system/service as first order logic predicates. 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 [13]. We use the extensional predicates *ChildAND (parent_node, child_node)*, *ChildOR(parent_node, child_node)* to denote the AND/OR goal decomposition. For instance, *ChildAND (parent, child)* is true when *child* is an AND-child of *parent* (similarly for *ChildOR*). Examples of AND goal decomposition are goals g_1 and g_2 (see Fig. 2). An example of OR decomposition is goal g_3. In Fig. 2, goal g_1 is the *AND* parent of a_1, g_2 and a_2. Such parent-child relationships are represented by assigning truth values to the ground atoms: *ChildAND(g_1, a_1)*, *ChildAND(g_1, g_2)* and *ChildAND(g_1, a_2)*. We use the extensional predicates *Pre(node, timestep)*, *Occ(node, timestep)*, and *Post(node, timestep)* to represent tasks' preconditions, occurrences and postconditions (respectively) at a certain timestep. For our work, we assume a total ordering of events according to their logical or physical timestamps [4]. Finally, we use the intensional predicates *G_Occ(node, timestep, timestep)* and *Satisfied(node, timestep)* to represent the goals occurrences and the goals/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 RCA process). If the overall service/transaction is successfully executed, then the top goal is considered to be satisfied, otherwise it is denied.

Rules Generation. The goal model relationships are used to generate a knowledge base consisting of a set of predicates, ground atoms and a set of rules in the form of first order logic expressions.

As presented above, the predicates *ChildAND(a,b)* and *ChildOR(c,d)* represent the *AND* and *OR* decomposition where a is the *AND* parent of b, and c is the *OR* parent of d (respectively).

A task a with a precondition *Pre(a, $t-1$)* and a postcondition *Post(a,$t+1$)* is satisfied at time $t+1$ if and only if $\{Pre\}$ is true at physical or logical time $t-1$ that is before task a occurs at time t, and $\{Post\}$ is true at time $t+1$ (see Equation 4 below).

$$Pre(a, t-1) \wedge Occ(a, t) \wedge Post(a, t+1) \Rightarrow Satisfied(a, t+1) \tag{4}$$

Unlike tasks which occur on a specific moment of time, goal occurrences span over an interval [t_1, t_2] that encompasses the occurrences times of its children goals/tasks. We represent the occurrence of a goal g using the predicate *G_Occ(g,t_1, t_2)* over the time interval $[t_1, t_2]$. Thus, a goal g with precondition

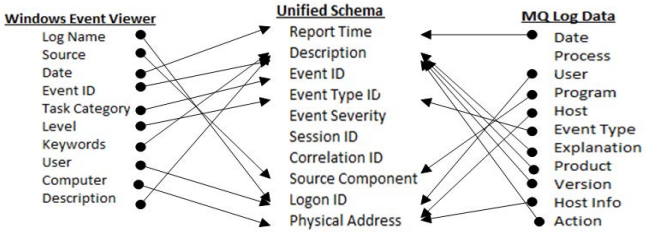

Fig. 3. Mappings from Windows Event Viewer and MQ Log into Unified Schema

Pre(g,t_1) and postcondition *Post(g, t_2)* is satisfied at time t_2 if and only if g's occurrence completes at t_2, and its precondition is true before its occurrence started at t_1 ($t_1 < t_2$) and finally, if its postcondition is true when its occurrence is completed at t_2 (see Equation 5 below).

$$Pre(g, t_1) \wedge G\_Occ(g, t_1, t_2) \wedge Post(g, t_2)) \Rightarrow Satisfied(g, t_2) \tag{5}$$

The truth values of the intensional predicate *G_Occ(goal, t_1, t_2)* (used in Equation 5) are inferred based on the satisfaction of all its children in the case of AND-decomposed goals (Equation 6) or at least one of its children in the case of OR-decomposed goals (Equation 7).

$$\forall a, Satisfied(a, t_1) \wedge ChildAND(g, a) \wedge (t_2 < t_1 < t_3) \Rightarrow G\_Occ(g, t2, t3) \tag{6}$$

$$\exists a, Satisfied(a, t_1) \wedge ChildOR(g, a) \wedge (t_2 < t_1 < t_3) \Rightarrow G\_Occ(g, t_2, t_3) \tag{7}$$

Contribution links of the form $node_1 \xrightarrow{++S} node_2$ are represented in Equation 8. (Similarly for *++D,--S,--D*).

$$Satisfied(node_1, t_1) \Rightarrow Satisfied(node_2, t_2) \tag{8}$$

4.2 Observation Generation

This section describes the second component of the framework as illustrated in Fig. 1.

Goal Model Compilation. This framework is built on the premise that the monitored system′s requirements goal model is available by system analysts or can be reverse engineered from source code using techniques discussed in [15].

Storing Log Data. In a system such as the loan application system, each component generates log data using its own native schema. We consider mappings from the native schema of each logger into a common log schema as shown in Fig. 3. We consider a unified schema for this work containing ten fields classified into four categories: *general*, *event specific*, *session information* and *environment related*. This proposed schema represents a comprehensive list of data fields considered to be useful for diagnosis purposes. In practice, many commercial monitor

environments contain only a subset of this schema. The identification of the mappings between the native log schema of each monitor component and the unified schema is outside the scope of this paper. Such mappings can be compiled using semi-automated techniques discussed in detail in [2] or compiled manually by subject matter experts. For the purposes of this study, we have implemented the mappings as tables using the Java programming language.

Log Data Interpretation. Once logged data are stored in a unified format, the pattern expressions annotating the goal model nodes are used to generate SQL queries that are applied to collect a subset of the logged data pertaining to the analysis. This matching process is discussed in detail in [17]. The $Pre(g_1)$ expression in section 4.1 can match the log data entry shown below:

Report_Time	**Description**	**Physical_Address**
2010-02-05 17:46:44.24	*Starting database eventsDB...*	*DATABASE*

In the case where $Pre(g_1)$ returns log entries when applied to the log data store, we conclude that there is evidence that the event associated with this query (goal model annotation) has occurred. If this query does not return any log entries, then we can't conclude that the event did not occur but rather that we are uncertain about its occurrence.

Ground Atoms Generation. The truth assignment for the extensional predicates is done based on the pattern matched log data. We show below a subset of the ground atoms that could be generated from the goal model depicted in Fig. 2,

pre(g_1,1), ?pre(a_1,1), !occ(a_1,2), post(a_1,3), ..., ?post(g_1,13), !satisfied(g_1,13)

The set of literals above, represents the observation of one failed loan application session. For some of the events corresponding to goals/tasks execution, there may be no evidence of their occurrence which can be interpreted as either they did not occur or they were missed from the observation set. We use 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 (!). For example, in Fig. 2 the observation of the system failure is represented by *!Satisfied(g_1,15)* which indicates top goal g_1 denial at timestep 15. Note in the above example, the precondition for task a_3 was denied and the occurrence of a_1 was not observed leading to the denial of task a_1, which led to goal g_1 not to occur and thus be denied (see Fig. 2). In turn, the denial of goal g_2 supports the observation goal g_1 did not occur and consequently satisfied.

Note that goal models are independently analyzed. For example, the ground atoms shown above are generated with respect to goal model in Fig. 2. The same log data may be analyzed by another goal model leading to a different stream of ground atoms. The two streams could have common ground atoms if the two goal models have tasks (or even annotations) in common. In the case of large

number of goal models, the value of the atoms can be cached in order to optimize the framework's performance.

The filtering of the predicate with timesteps of interest is done using Algorithm 1. Algorithm 1 consists of two steps: first, a list of literals is generated (see example above) by depth-first traversing the goal model tree and generating a list of literals from the nodes annotations (precondition, occurrences and postconditions); second, sequentially go through that list and look for evidence in the log data for the occurrence of each literal within a certain time interval. This time interval is specified as sliding window that is centered around the time when the system is reported to have failed. The size of the window depends on the monitored applications and type of transactions.

Algorithm 1. Observation Generation

Input: *goal_model*: goal model for the monitored system
log_data: log data stored in central database
Output: *literals*: set of literals of the form *[?,!] literal(node,timestep)*
Procedure [literals] **generate_literals**(goal_model) {
Set *Curr_Node* = top node in *goal_model* //start at the top of the goal tree
Set *g_counter* = 0 //set global counter to zero
[literals].addLast(*pre(Curr_Node, node.precondition_annotation, g_counter)*)
While *Curr_Node* has children,
find leftmost not-visited *child* of *Curr_Node*
recursively call *generate_literals(child)*
If all children of *Curr_Node* are visited return *[literals]*;
If *Curr_Node* has no children (task),
g_counter++
[literals].addLast(*occ(Curr_Node, node.occurrence_annotation, g_counter))*
g_counter++
[literals].addLast(*post(Curr_Node, node.postcondition_annotation, g_counter))*
return *[literals]*;}}
Procedure [literals] **assign_logical_values(log_data, [literals]) {**
for each variable *literal(node, annotation, counter)* in the set *[literals]*
filter the log data based on the pattern expression in the annotation;
if no log data found matching the pattern expression:
replace *literal(...)* by *(? literal(node, counter))* else skip
if system is reported to have failed:
literals.addLast(! satisfied(top, counter));
return [literals];}}
main(goal_model, log_data)
[literals] = generate_literals(goal_model);
assign_logical_values(log_data, [literals])
return [literals];}

The following is an example of Algorithm 1 based on Fig. 2: the log entry (*2010-02-05 17:46:44.24 Starting up database eventsDB ... DATABASE*) matches the pattern of the precondition of goal g_1 thus representing an evidence for the occurrence of this event (precondition in this case). For other logical literals that don't have evidence in the log data that they occurred, a (?) mark is assigned to these literals, such as the precondition of a_1. This process is also exemplified in Fig. 4 below where the log data is filtered and used to assign logical values to a set of ordered literals generated from the goal model.

Uncertainty Representation. This framework relies on log data as evidence for the diagnostic process. The process of selecting log data (described in the previous step) can potentially lead to false negatives and false positives which in turn lead to a decreased confidence in the observation.

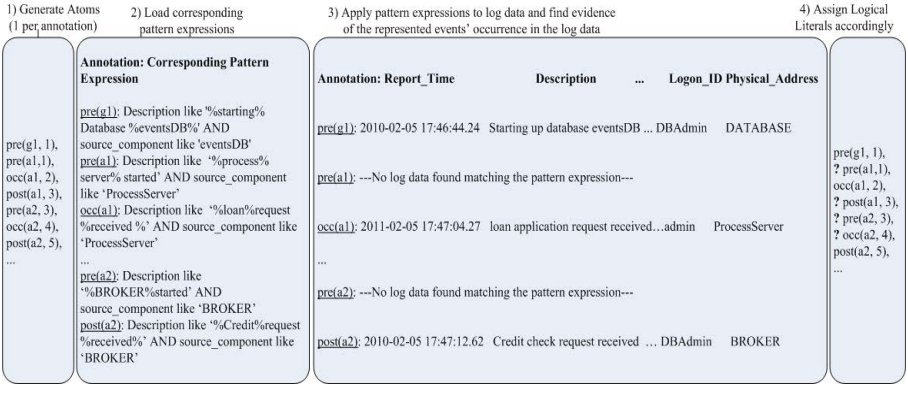

Fig. 4. Generating Observation from Log Data

We address uncertainty in observations using a combination of logic and probabilistic models: First, 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 set of real-valued weights that are generated using a learning process and a training log data set. The weight of each rule represents our confidence relative to other rules in the knowledge base (weight learning is discussed in section 4.3). Consequently, the probability inferred for each atom (consequently the diagnosis) depends on the weight of the competing rules where this atom occurs. For instance, the probability of satisfaction of task a_4 in Fig. 2 (*Satisfied*(a_4,*t*)) is impacted by the weight of Equation 9 with weight *w1* and the weight of any other equation containing a_4,

$$w1 \;\; Pre(a4,t) \wedge Occ(a4,t+1) \wedge Post(a4,t+2) \Rightarrow Satisfied(a4,t+2) \quad (9)$$

Second, uncertainty is also handled by applying an open world assumption to the observation where a lack of evidence does not necessarily negate an event's occurrence but rather weakens its probability.

4.3 Diagnosis

The third component in the framework generates a Markov Logic Network (MLN) based on the diagnostic knowledge base (Section 4.1) and then uses the generated observation (Section 4.2) to provide an inference on the root causes for the system failure.

Markov Network Construction. A Markov Logic Network is a graph constructed using an exhaustive list of rules and predicates as nodes as well as, grounding predicates with all possible values, and connecting them with a link if these predicates coexist in a grounded formula. The choice of possible values for grounding the predicates can lead to an explosion in the number of ground atoms and network connections if not carefully designed, in particular when modeling time. For this purpose, we represent time using timesteps (integers) that denote the time interval that one session of the service described by the goal model takes to execute.

Weight Learning. Learning the weight of the rules is performed using a generative learning algorithm with the use of a training set [8]. During automated weight learning, each formula is converted to CNF, and a weight is learned for each of its clauses. The weight of a clause is used as the mean of a Gaussian prior for the learned weight. On the other hand, the quality and completeness of the training set impact the set of learned weights. We measure the completeness of the training set used in our experiment with respect to the goal model representing the monitored application. In particular, the training set we considered in our experiments contains at least one pair of evidence/expected output for each node in the goal model. In addition, all the rules in the knowledge base are exercised at least once in the training set. The learnt weight can be further modified by the operator to reflect his or her confidence in the rules. For example, the rules that embed a fact such as when the system operator visually witness the system's failure (represented as the top goal being denied) should be given more weight than the rules where goal/tasks satisfaction are inferred based on "uncertain" log data.

Inference. Using the constructed Markov Network, we can infer the probability distribution for the ground atoms in the KB given the observations. Of particular interest are the ground atoms for the *Satisfied(node, timestep)* predicate which represents the satisfaction or denial of tasks and goals in the goal model at a certain timestep. Algorithm 2 below is used to produce a diagnosis for failure of a top goal at timestep T. MLN inference generates weights for all the ground atoms of *Satisfied(task,timestep)* for all tasks and at every timestep. Based on the

MLN rules listed in section 3, the contribution of a child node's satisfaction to its parent goal's occurrence depends on the timestep of when that child node was satisfied. Algorithm 2 recursively traverses the goal model starting at the top goal node. Using the MLN set of rules, the algorithm identifies the timestep t for each task's *Satisfied(n,t)* ground atom that contributes to its parent goal at a specific timestep (*t'*). The *Satisfied* ground atoms of tasks with the identified timesteps are added to a secondary list and then ordered based on their timesteps. Finally, the algorithm inspects the weight of each ground atom in the secondary list (starting from the earliest timestep), and identifies the tasks with grounds atoms that have a weight of less than 0.5 as the potential root cause for the top goal's failure. The tasks with ground atoms at earlier timesteps are identified as more likely to the source of failure. A set of diagnosis scenarios based on the goal model in Fig. 2 is shown in Table 1 and discussed in section 5.

Algorithm 2. Diagnosis Algorithm

Input: *mln*: weighted rules r for the goal model (diagnostic knowledge base)
Satisfied(n,t): ground atoms for predicate *Satisfied*
T: timestep when the top goal satisfaction is investigated
Output: Γ: ranked list of root causes
Diagnose(mln, Satisfied(n,t), T) {
initialize Θ and Γ to be empty
add *Satisfied(topgoal,T)* to Φ
for each goal g corresponding to an atom in Φ {
set t = timestep in the *Satisfied(g,t)* ground atom
for each task a child of goal g {
load rule r that shows the contribution of *Satisfied(a,t1)* to *G_Occ(g,t)*
identify the value of *t1*
add ground atoms *Satisfied(a,t1)* and its weight to Θ
}
remove *Satisfied(g1,t)* and add *Satisfied* ground atoms of all sub-goals of g to Φ
}
next, order the ground atoms in Θ based on timesteps
for each ground atom *atom* in Θ {
if weight of *Satisfied(atom,t)* ≤ 0.5) {
if a is an AND child $\Rightarrow$ add a to Γ
if a is an OR child {
find siblings of a in goal model
if no sibling of a is satisfied in $\Theta \Rightarrow$ add a to Γ
if a has at least one satisfied sibling $\Rightarrow$ CONTINUE.}}}
return Γ}

5 Case Study

The objective of this case study is to demonstrate the applicability of the proposed framework in detecting the root causes for failures in software systems that are composed of various components and services. The motivating scenario has been implemented as a proof of concept and includes 6 systems/services: a front end application (soapUI), a process server (IBM Process Server 6.1), a loan application business process, a message broker (IBM WebSphere Message Broker v7.0), a credit check Web Service and an SQL server (Microsoft SQL Server 2008). The case study consists of two scenarios. The two scenarios we present in this study include one success and one failure scenario (see Table 1). Scenario 1 represents a successful execution of the loan application process. Please note, that the denial of task a_7 does not represent a failure in the process execution since goal g_3 is exclusive OR-decomposed into a_6 *(extract credit history for existing clients)* and a_7 *(calculate credit history for new clients)*, and the successful execution of either a_6 or a_7 is enough for g_3's successful occurrence (see Fig. 2). During each loan evaluation and before the reply is sent back to the requesting application, a copy of the decision is stored in a local table (a_4 *(Update loan table)*). Scenario 2 represents a failure to update the loan table leading to failure of top goal g_1. Using Algorithm 2, we identify a_4 (Update loan table) as the root cause for failure. Note that although a_7 was denied ahead of task a_4, it is not the root cause since it is an OR child of g_3, and its sibling task a_6 was satisfied.

Table 1. Two scenarios for the Loan Application

Scenario	**Observed (& Missing) Events(s)**	**Satisfied(task, timestep)**						
		(a1,3)	**(a3,5)**	**(a6,7)**	**(a7,7)**	**(a4,9)**	**(a5,11)**	**(a2,13)**
Successful execution	Pre(g_1,1), Pre(a_1,1), Occ(a_1,2), Post(a_1,3), Pre(g_2,3), Pre(a_3,3), Occ(a_3,4), Post(a_3,5), Pre(g_2,5), Pre(a_6,5), Occ(a_6,6), Post(a_6,7), ?Pre(a_7,5), ?Occ(a_7,6), ?Post(a_7,7), Post(g_3,7), Pre(a_4,7), Occ(a_4,8), Post(a_4,9), Pre(a_5,9), Occ(a_5,10), Post(a_5,11), Pre(a_2,11), Occ(a_2,12), Post(a_2,13), Post(g_1,13), Satisfied(g_1,13)	0.88	0.87	0.93	0.01	0.66	0.63	0.82
Failed to update loan database	Pre(g_1,1), Pre(a_1,1), Occ(a_1,2), Post(a_1,3), Pre(g_2,3), Pre(a_3,3), Occ(a_3,4), Post(a_3,5), Pre(g_3,5), Pre(a_6,5), Occ(a_6,6), Post(a_6,7), ?Pre(a_7,5), ?Occ(a_7,6), ?Post(a_7,7), Post(g_3,7), Pre(a_4,7), ?Occ(a_4,8), ?Post(a_4,9), ?Pre(a_5,9), ?Occ(a_5,10), ?Post(a_5,11), ?Pre(a_2,11), ?Occ(a_2,12), ?Post(a_2,13), ?Post(g_1,13), !Satisfied(g_1,13)	0.88	0.87	0.90	0.01	0.01	0.01	0.01

The probability values (weights) of the ground atoms range 0^+ (highly denied) to 0.99 (highly satisfied). The framework was evaluated on a Ubuntu Linux running on Intel Pentium 2 Duo 2.2 GHz machine. We have used a set of extended goal models representing the loan application goal model to evaluate the performance of the framework when larger goal models are used. The 4 extended goal models contained 13, 50, 80 and 100 nodes respectively. Obtained results indicate that the matching and ground atom generation algorithms performance depends linearly on the size of the corresponding goal model and log data and thus lead us to believe that the process can easily scale for larger and more complex goal models. Our experiments have also suggested that the number of ground atoms/clauses, which directly impacts the size of the resulting Markov model, is linearly proportional to the goal model size, and that the number of ground atoms and clauses increases linearly with the size of the goal model (see Figure 5). Furthermore, results indicate that the learning and inference time ranged from 2.2 and 5 seconds for a goal model of 10 nodes, up to 34.2 and 53 seconds respectively for a model of 100 nodes (see Figure 6). As a result, the these initial case studies suggest that our approach in its current implementation can be applied to industrial software applications with small to medium-sized requirement models (i.e. 100 nodes).

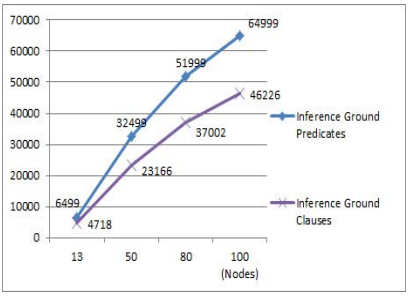

Fig. 5. Number of Ground Predicates/Clauses vs. Goal Model Size

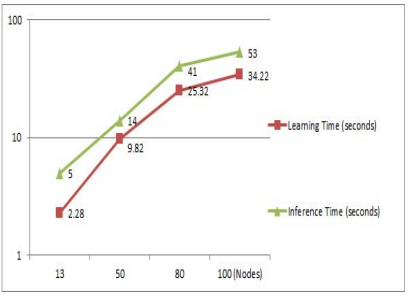

Fig. 6. Learning and Inference Time vs. Goal Model Size

6 Related Work

Our current study is based in part on earlier work by Wang [14] and our previous work [16], [17]. Wang et al. [14] proposed annotated goal models to represent monitored systems and transformed the diagnostic problem into a propositional satisfiability (SAT) problem that can be solved using SAT solvers where evidence supporting or denying the truth of individual ground predicates is collected by instrumenting software systems. Zawawy et al. [16] enhanced the framework by Wang et al. by matching natively generated log data and using it as evidence which is less intrusive and more practical when analyzing off-the-shelf commercial products. The current study extends our previous work by introducing a diagnostic component based on MLNs allowing the handling of inaccuracies in

modeling the monitored systems dependencies as well as missing and inaccurate observations. Our approach uses weighted formulas instead of hard relationships allowing for conflicts such as when a goal/task is satisfied and denied at the same time. This is an advantage over [14] which depends on accurate observation. RCA approaches in the literature can be classified as based on probabilistic approaches such as Bayesian Belief Networks [12], or based on machine learning such as decision trees and data mining [1], [3], or based on rule sets and decision matrices [7]. Steinder et al. (2004) used Bayesian networks to represent dependencies between communication systems and diagnose failures while using dynamic, missing or inaccurate information about the system structure and state [12]. Our approach is similar in that we use Markov networks (undirected graphs) instead of Bayesian networks in order to fit the evidence. The advantage of our approach over [12] is that using first order logic; we are able to model complex relationships between the monitored systems and not just simple one to one causality relationships. A more recent work pertaining to log analysis and reduction is the work by Al-Mamory et al. where RCA is used to reduce the large number of alarms by finding the root causes of false alarms [1]. [1] uses data mining to group similar alarms into generalized alarms and then analyze these generalized alarms and classify them into true and false alarm. Later, the generalized alarms can be used as rules to filter out false positives. Chen et al. (2004) use statistical learning to build a decision tree which represents all the paths that lead to failures [3]. The decision tree is built by branching for each node based on the values of a specific feature and later on pruning subsumed branches. The advantages of our approach over [1], [3] is that it is resilient to missing or inaccurate log trace, adapts dynamically for new observations, and can be used to diagnose multiple simultaneous faults.

7 Conclusions

This paper presents a framework that assists operators perform RCA in software systems. The framework takes a goal driven approach whereby software system requirements are modeled as goal trees. Once the failure of a functional or non-functional requirement is observed as a symptom, the corresponding goal tree is analyzed. The analysis takes the form of first, selecting the events to be considered based on a pattern matching process, second on a rule and predicate generation process where goal models are denoted as Horn Clauses and third, a probabilistic reasoning process that is used to confirm or deny the goal model nodes. The most probable combinations of goal model nodes that can explain the failure of the top goal (i.e. the observed failure) is considered the most probable root cause. Goal models for large systems can be organized in a hierarchical fashion allowing for higher tractability and modularity in the diagnostic process. Initial results indicate that the approach is tractable and allows for multiple diagnoses to be achieved and ranked based on their probability of occurrence. This work is conducted in collaboration with CA Labs and is funded by CA Technologies and the Natural Sciences and Engineering Research Council of Canada.

References

1. Al-Mamory, S.O., Zhang, H.: Intrusion detection alarms reduction using root cause analysis and clustering. Comput. Commun. 32(2), 419–430 (2009)
2. Alexe, B., Chiticariu, L., Miller, R.J., Tan, W.C.: Muse: Mapping understanding and design by example. In: Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, Cancun, Mexico, April 7-12, pp. 10–19 (2008)
3. Chen, M., Zheng, A.X., Lloyd, J., Jordan, M.I., Brewe, E.: Failure diagnosis using decision trees. In: Int'l. Conference on Autonomic Computing, pp. 36–43 (2004)
4. Dollimore, J., Kindberg, T., Coulouris, G.: Distributed Systems: Concepts and Design, 4th edn. Int'l Computer Science Series. Addison Wesley (May 2005)
5. Domingos, P.: Real-World Learning with Markov Logic Networks. In: Boulicaut, J.-F., Esposito, F., Giannotti, F., Pedreschi, D. (eds.) ECML 2004. LNCS (LNAI), vol. 3201, p. 17. Springer, Heidelberg (2004)
6. 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)
7. Hanemann, A.: A hybrid rule-based/case-based reasoning approach for service fault diagnosis. In: AINA 2006: Proceedings of the 20th International Conference on Advanced Information Networking and Applications, pp. 734–740. IEEE Computer Society, Washington, DC (2006)
8. Kok, S., Sumner, M., Richardson, M., Singla, P., Poon, H., Lowd, D., Domingos, P.: The alchemy system for statistical relational ai. technical report, university of washington, seattle, wa (2007), `http://alchemy.cs.washington.edu`
9. Library, M.M.: Transact-sql reference (2012), `http://msdn.microsoft.com/en-us/library/ms179859(v=sql.100).aspx#2`
10. Oltsik, J.: The invisible log data explosion (2007), `http://news.cnet.com/8301-10784_3-9798165-7.html`
11. Richardson, M., Domingos, P.: Markov logic networks. Mach. Learn. 62, 107–136 (2006)
12. Steinder, M., Sethi, A.S.: Probabilistic fault diagnosis in communication systems through incremental hypothesis updating. Comput. Netw. 45(4), 537–562 (2004)
13. Tran, S.D., Davis, L.S.: Event Modeling and Recognition Using Markov Logic Networks. In: Forsyth, D., Torr, P., Zisserman, A. (eds.) ECCV 2008, Part II. LNCS, vol. 5303, pp. 610–623. Springer, Heidelberg (2008)
14. Wang, Y., Mcilraith, S.A., Yu, Y., Mylopoulos, J.: Monitoring and diagnosing software requirements. Automated Software Engg. 16(1), 3–35 (2009)
15. Yu, Y., Lapouchnian, A., Liaskos, S., Mylopoulos, J., Leite, J.: From goals to high-variability software design, pp. 1–16 (2008)
16. Zawawy, H., Kontogiannis, K., Mylopoulos, J.: Log filtering and interpretation for root cause analysis. In: ICSM 2010: Proceedings of the 26th IEEE International Conference on Software Maintenance (2010)
17. Zawawy, H., Mylopoulos, J., Mankovski, S.: Requirements driven framework for root cause analysis in soa environments. In: MESOA 2010: Proceedings of the 4th Int'l Workshop on Maintenance and Evolution of Service-Oriented Systems (2010)