2009 IEEE/ACM International Conference on Automated Software Engineering

Self-repair Through Reconfiguration: A Requirements Engineering Approach

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

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

***Abstract*—High variability software systems can deliver their functionalities in multiple ways by reconfiguring their components. High variability has become important because of current trends towards software systems that come in product families, offer high levels of personalization, and fit well within a service-oriented architecture. The purpose of our research is to propose a framework that exploits such variability to allow a software system to self-repair in cases of failure. We propose an autonomic architecture that consists of *monitoring*, *diagnosis*, *reconfiguration* and *execution* components. This architecture uses requirements models as a basis for monitoring, diagnosis, and reconfiguration. We illustrate our proposal with a medium-sized publicly available case study (an Automated Teller Machine (ATM) simulation), and evaluate its performance through a series of experiments. Our experimental results demonstrate that it is feasible to scale our approach to software systems with medium-size requirements.**

Keywords: Autonomic computing, Adaptive systems, Requirement Monitoring and Diagnosis, Self-reconfiguration

I. Introduction

High variability software systems deliver their functionalities in multiple ways by reconfiguring their components. If one of these ways fails, the system can switch to an alternative. Consider an Automated Teller Machine (ATM) system that may either print or display a receipt at the end of a transaction. If the ATM's printer malfunctions, the ATM may display a receipt instead. High variability has become an important concern for software designers. It underlies the notions of product family, software personalization, and service-oriented architecture.

This paper proposes a framework that exploits high variability in a software system to deliver self-repair capabilities through reconfiguration. The proposed framework (1) monitors a system's operations for failures; (2) diagnoses root causes when failures occur; (3) identifies the actual failure when multiple diagnoses explain monitored data; (4) generates a system reconfiguration that avoids the failed function(s) and compensates for any effects thereof.

The framework contains *monitoring*, *diagnostic*, *reconfiguration*, and *execution* components. The monitoring component monitors the satisfaction of software requirements and generates log data at a level of granularity that can be tuned depending on monitored feedback. When errors are found, the diagnostic component infers the denial of the requirements and identifies problematic components. For repair, the reconfiguration component selects a best system reconfiguration among all competing reconfigurations that are free of failures. The chosen reconfiguration contributes most positively to the systems non-functional requirements, and it minimally reconfigures the system from its current configuration. The execution component runs compensation actions to restore the system to its previous consistent state, and reconfigures the system under the chosen configuration. The monitoring and diagnostic components of our architecture are based on our earlier work [23], [24]. The idea of using high variability to support autonomicity was initially sketched in [25].

Our proposal fits within the context of on-going research on adaptive and autonomic software [12]. Much of this research achieves autonomicity by introducing special purpose code that checks predefined low level system properties and reacts if certain conditions become true [3]. Our research exploits requirements models to define monitoring, diagnostic and reconfiguration rules.

The rest of the paper is structured as follows. Section II gives the background to our work. Section III provides an overview of our self-repair architecture. Section IV presents the architecture's monitoring, diagnostic, reconfiguration planner, and execution components. Section V discusses implementation details. We illustrate and evaluate our framework in Section VI. Our related work is discussed in Section VII. Finally, we discuss limitations of our framework and our future plans in Section VIII.

II. Preliminaries

Requirements Engineering (RE) is a branch of Software Engineering that deals with the elicitation and analysis of system requirements. In recent years, RE has used goal models to model and analyze stakeholder objectives [5]. Software systems' functional requirements are represented as hard goals, while their non-functional requirements are represented as soft goals [18]. A goal model is a graph structure including AND- and OR-decompositions of goals into subgoals, as well as means-ends links that relate leaf level goals to tasks ("actions") that can be performed to fulfill them. If goal G is AND/OR-decomposed into subgoals $G_1, \ldots, G_n$, then all/at-least-one of the subgoals must be satisfied for G to be satisfied.

1527-1366/09 $29.00 © 2009 IEEE
DOI 10.1109/ASE.2009.66

Following [10], and apart from decomposition links, hard goals and tasks can be related to each other through various contribution links: $++S$, $--S$, $++D$, $--D$, $++$, $--$. Given two goals G_1 and G_2, the link $G_1 \xrightarrow{++S} G_2$ (respectively $G_1 \xrightarrow{--S} G_2$) means that if G_1 is satisfied, then G_2 is satisfied (respectively denied). But if G_1 is denied, we cannot infer denial (or respectively satisfaction) of G_2. The meanings of links $++D$ and $--D$ are dual w.r.t. to $++S$ and $--S$ respectively, by inverting satisfiability and deniability. Links $++$ and $--$ are shorthand for the $++S$, $++D$, and $--S$, $--D$ relationships respectively, and they propagate both satisfaction and denial of the source goal/task to the target goal/task. These $++$ and $--$ links represent the strong MAKE($++$) and BREAK($--$) contributions between hard goals/tasks. Hard goals and tasks can have one of two satisfaction labels: $\neg FD$ and FD, representing the full evidence that a hard goal/task is satisfied (not denied) and denied respectively.

The partial (weaker) contribution links HELP ($+$) and HURT ($-$) are not included between hard goals/tasks because we do not reason with partial evidence for hard goal and task satisfaction and denial. These weaker links may proceed from hard goals/tasks to soft goals. Soft goals can have one to four satisfaction labels: FS, FD, PS, and PD representing the full and partial evidence that a soft goal is satisfied and denied respectively. The class of goal models used in our work has been formalized in [10], where sound and complete algorithms are provided for inferring whether a set of root-level goals are satisfied.

We associate goals and tasks with preconditions and postconditions (hereafter *effects* to be consistent with AI terminology), and monitoring switches. Preconditions and effects are propositional formulae in Conjunctive Normal Form (CNF). Monitoring switches are boolean flags that can be switched on/off to indicate whether the corresponding goal/task is to be monitored. In addition, we associate soft goals with one of three priority levels: *high*, *medium*, or *low*. We use Soft goal priority value to guide the reconfiguration process in selecting a reconfiguration that contributes most positively to soft goals of higher priorities.

We use the goal model in Figure 1 as a running example throughout this paper to illustrate how our framework works. The goal model contains 9 goals and 11 tasks, shown in ovals and hexagons, respectively. The root goal $g1$ is AND-decomposed to goals $g2$ and $g7$, indicating that goal $g1$ is satisfied if and only if goals $g2$ and $g7$ are satisfied. $G2$ is OR-decomposed into goal $g3$ and task $a7$, indicating that $g2$ is satisfied if and only if either $g3$ or $a7$ is satisfied. MAKE ($++$), BREAK ($--$), HELP ($+$), and HURT ($-$) contribution links proceed from hard goals and tasks to the soft goals, $sG1$ and $sG2$, of the goal model.

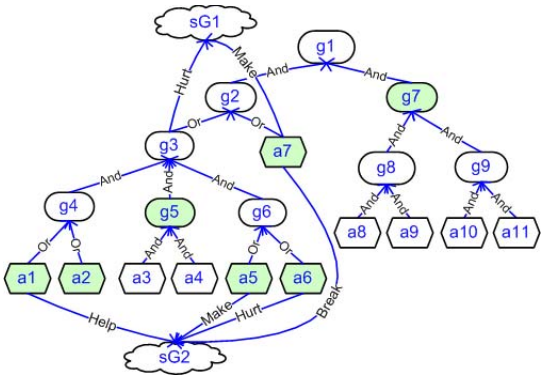

Figure 1. Example Goal Model

A. Autonomic Computing

The difficulty of maintaining today's software systems comes from their complexity. Autonomic computing [12] is an IBM initiative and a rapidly growing research area that aims at creating more self-managing software systems to shift management complexity from administrators back to software systems themselves. Autonomic systems manage themselves at lower level in accordance with policies and goals provided by administrators without their conscious recognition.

Given high level goals and policies from system administrators, An autonomic system is able to automatically reconfigure itself when new components come into or are removed from the system (self-configuration); able to continually tune its parameters for optimization (self-optimization); able to monitor, analyze, and recover from faults and failures when they occur (self-healing); and able to protect itself from malicious attacks (self-protection). All the four self management tasks require the system to consciously monitor itself and its environment, analyze monitored data to learn what it means, plan for changes to be made if any, and execute the plan. This *Monitor*, *Analyze*, *Plan*, and *Execution* loop is known as the MAPE loop for autonomic systems [12].

III. Framework Overview

We propose an autonomic architecture that takes as input a high variability legacy software system, and extends it with self-repair capabilities through reconfiguration. Four software engineering steps are necessary to prepare for the architecture's input. (1) A *reverse engineering* step where a requirements model is either (a) reverse engineered from source code using techniques we presented in [26], or (b) provided by requirements analysts. Traceability links are extracted and maintained, and they link between system source code and goals/tasks. (2) An *annotation* step where requirements analysts annotate the goals and tasks in the goal model with monitoring switches, preconditions and effects. When these switches are enabled, the architecture monitors at run

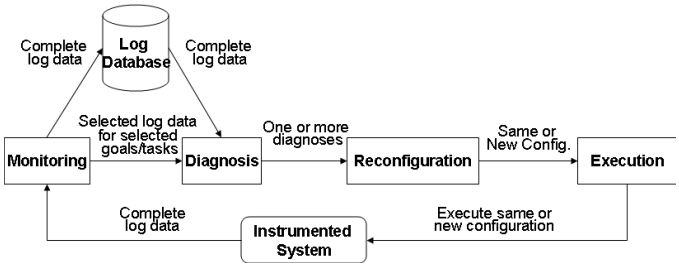

Figure 2. Architecture Overview

time the satisfaction of the corresponding goals and tasks. Goal and task satisfaction/denial is analyzed using the truth values of their associated preconditions and effects. (3) An *instrumentation* step where the managed software system is instrumented so that it produces log data at run time. The initial implementation of the instrumentation component is provided in [27]. And (4), a *compensation preparation* step where domain experts prepare compensation actions. When failures occur, these compensation actions are executed to restore the system to its previous consistent state.

Figure 2 provides an overview of our architecture. The architecture contains *Monitoring*, *Diagnosis*, *Reconfiguration*, and *Execution* components that respectively correspond to the monitor, analyze, plan, and execution (MAPE) components in an autonomic system [12]. The theoretical foundations and evaluation of the monitoring and diagnostic components are presented in [23], [24]. We propose extensions to the monitoring and diagnostic components in Sections IV-A and IV-B respectively. The reconfiguration and execution components are presented in Sections IV-C and IV-D respectively.

The instrumented system generates complete log data at run time. The monitoring component collects the complete log data and saves them to a *Log Database*. To reduce diagnostic reasoning overhead, the monitoring component selects a subset of the complete log data and passes it to the diagnostic component for analysis. The selected log data contain enough information to allow the diagnostic component to infer whether any requirements are denied.

We transform the problem of diagnosing software systems into a SAT problem by encoding goal model relations and log data into a propositional formula that is satisfied if and only if there is a diagnosis. A diagnosis specifies whether or not each goal and task is fully denied. If the diagnostic component finds denials of system requirements, it returns either a single diagnosis identifying one or more failed tasks, or multiple competing diagnoses, each pinpointing different goal/task denials.

The *reconfiguration* component computes a best reconfiguration that is free of failures. It therefore needs to "know" which goals and tasks have failed. If a single diagnosis is generated, the *reconfiguration* component needs no further information. If multiple diagnoses are returned, the framework can follow one of three approaches. The first two infer a single precise diagnosis. (1) The diagnostic component can retrieve a subset of relevant log data corresponding to failed goals/tasks from the Log Database, and infer therefrom a precise diagnosis. (2) The diagnostic component can generate a most probable diagnosis (Section IV-B1). These approaches are useful when repair makes it important to find the exact root causes of failure. At other times, it may be more preferable to find reconfigurations quickly. In these cases, the reconfiguration planner component can (3) find all valid reconfigurations without a precise diagnosis. The goal model structure and the partial log data contain enough information to make this possible.

If no requirements are denied, no reconfiguration is needed. Otherwise the reconfiguration planner component computes a best reconfiguration for the system's next execution. In our work, a configuration contains a list of tasks from the goal model, whose successful executions lead to the satisfaction of the root goal. The appropriate configuration is then passed to the *Execution* component. If the configuration passed to the *Execution* component is the same as the system's current configuration, no configuration changes are made. Otherwise the execution component executes any necessary compensation actions to bring the system to its previous consistent state. We adopt a model of long-term database transactions to deal with executions of compensation actions [8]. The execution component then reconfigures the system using the new configuration. The steps described above constitute one execution session and are repeated as necessary.

Note that the monitoring component is online. It runs with the monitored software system to collect generated log data. The diagnostic, reconfiguration, and execution components are all offline. They run after the monitored system finishes its execution. If a new configuration is generated, the monitored system runs with this new configuration in its *next* execution session.

IV. Our Framework

This section presents our monitoring, diagnostic, reconfiguration, and execution components. The underlying theories of our monitoring and diagnostic components were presented in [23], [24]. In Sections IV-A and IV-B, we review the basics of our monitoring and diagnostic components, respectively, and propose extensions necessary for self-repair. The reconfiguration and execution components are presented in Sections IV-C and IV-D.

A. Monitoring

The monitoring component can monitor requirements at different levels of granularity. There is a tradeoff between monitoring granularity and diagnostic precision. The finest level of monitoring granularity is at the functional level

where all leaf level tasks are monitored. In this case, task-level log data is generated, and a single precise diagnosis can be inferred. The disadvantage of task level monitoring is high monitoring and diagnostic overhead. Coarser levels of granularity only monitor higher-level goals in a goal model. In these cases, less complete and goal-level log data is generated, leading to multiple competing diagnoses. Both monitoring and diagnostic overheads are lower. The disadvantage is that if requirements denials are found, multiple diagnoses are returned, each pinpointing possible failures.

As shown in Figure 2, the instrumented system generates complete log data at run time. These data contain log traces for all tasks and some selected goals, allowing a single precise diagnosis to be inferred. The monitoring component selects a subset of the complete log data and passes it to the diagnostic component for analysis. Analyzing a smaller amount of data reduces diagnostic reasoning overhead. Nonetheless, the selected log data still enables the diagnosis component to infer whether any requirements are denied. In addition, the selected log data permits fast reconfiguration without a precise diagnosis. The selected subset of the log data correspond to the log instances for selected goals and tasks. Section IV-A1 explains how such goals and tasks are selected for monitoring.

Goals and tasks in a goal model are associated with monitoring switches, preconditions, and effects. Monitoring switches can be switched on/off to indicate whether the corresponding goal/task is to be monitored. Preconditions and effects are propositional formulae in Conjunctive Normal Form (CNF) that must be true before and (respectively) after a goal is satisfied or a task is successfully executed. Log data consist of truth values of observed domain literals (specified in goal/task preconditions and effects) and the occurrences of tasks, each associated with a timestep t. The logical timestep t is incremented by 1 each time a new batch of monitored data arrives and is reset to 1 when a new session starts.

1) Selecting Monitoring Granularity: Our framework allows for fast reconfiguration without a precise diagnosis, in situations where finding a precise diagnosis isn't important. We achieve this by extending the original monitoring component presented in [23], [24] with the capability of selecting an optimal monitoring granularity for the purpose of reconfiguration. With this extension, the monitoring component monitors as few goals/tasks as necessary. Yet, when failures occur, satisfactions/denials of these monitored goals/tasks give the reconfiguration component enough information to find all reconfigurations free of failures. Intuitively, goals and tasks that share the same reconfiguration are monitored collectively rather than individually. Consider the goal model given in Figure 1. Tasks $a3$, and $a4$ share the same reconfiguration, because if either fails, their respective reconfigurations are the same. It is therefore enough to know *if* either of these tasks fails, rather than *which* task fails. We introduce the concept of *AND-subtree* given in Definition 1. An AND-subtree does not contain any OR decompositions between its root and any of its tasks. The root of an AND-subtree is iteratively AND-decomposed to all its leaf level tasks through intermediate goals.

Definition 1: (**AND-subtree**) An AND-subtree is part of a goal model where all the decompositions between the root of the subtree and its leaf level tasks are AND-decompositions.

Definition 2: (**Maximal AND-subtree**) A Maximal AND-subtree is an AND-subtree where no other AND-subtrees contain it.

In the goal model given in Figure 1, goals $g5$, $g7$, $g8$, and $g9$ root AND-subtrees. Goals $g5$ and $g7$ root two Maximal AND-subtrees.

Algorithm 1 Select an Optimal Monitoring Granularity

```
select_monitoring_granularity (goal_model) {
    for each goal g {
        if (g has multiple parents)
            select g for monitoring
        else if (g is the root of a maximal AND-subtree)
            select g for monitoring }
    for each task t {
        if (t has multiple parents)
            select t for monitoring
        else if (t is OR-decomposed from its parent)
            select t for monitoring}
    }}
```

We propose Algorithm 1 for selecting an optimal monitoring granularity. Algorithm 1 selects for monitoring: (1) goals and tasks with multiple parents; (2) root goals of maximal AND-subtrees; and (3) tasks that are OR-decomposed from their immediate parents. Goals/tasks with multiple parents are monitored because they participate in multiple configurations. The Reconfiguration Planner needs to "know" whether or not they are denied before it can compute all reconfigurations. All tasks that belong to a maximal AND-subtree, and that do not have multiple parents, share the same reconfigurations. We monitor their satisfactions collectively by monitoring their root goal in the subtree. Tasks that are OR-decomposed from their immediate parents are monitored, because these tasks' satisfactions/denials determine if they participate in a reconfiguration. When monitoring is done at this optimal granularity, the reconfiguration component can compute all reconfigurations without a precise diagnosis.

In the goal model given in Figure 1, the following goals and tasks are recommended for monitoring: $a1$, $a2$, $g5$, $a5$, $a6$, and $g7$ (their respective ovals/hexagons are highlighted in Figure 1).

B. Diagnosis

The diagnostic component analyzes log data and infers satisfaction/denial labels for all the goals and tasks in a

goal model. This diagnostic reasoning process involves two steps: (1) inferring satisfaction/denial labels for monitored goals/tasks; and (2) propagating these satisfaction/denial labels to the rest of the goal model through AND-/OR-decomposition links and contribution links. We reduced the problem of searching for a diagnosis to that of the satisfiability of a propositional formula Φ, where Φ is the conjunction of the following axioms: (1) axioms for reasoning with goal/task denials (step 1); and (2) axioms for propagating inferred goal/task denials to the rest of the goal model (step 2). Interested readers can refer to [23], [24] for a complete account of our axiomatizations and algorithms.

In our work, a *diagnosis* specifies for each goal/task in the goal model whether or not it is fully denied. The predicate FD is used to express full evidence of goal and task denial at a certain timestep or during a specific session. For example, predicates $FD(g_1, 5)$ and $FD(a_1, s_1)$ indicate that goal g_1 and task a_1 are denied at timestep 5 and session s_1 respectively.

We offer two approaches to finding a single precise diagnosis when multiple diagnoses are returned. The first approach retrieves a subset of the complete log data (from the Log Database) relevant to the failed goals/tasks, and infers therefrom a precise diagnosis. The second approach computes a most probable diagnosis using information from the existing diagnoses (Section IV-B1).

1) Most Probable Diagnoses: This section discusses extensions we made to the original diagnostic component (presented in [23], [24]) which allow it to finding a most probable diagnosis. We adopt the probability model presented in [2] to calculate probabilities for goal satisfaction given the probabilities that its tasks are satisfied. The satisfaction of a root goal in a goal model depends on the satisfaction of its tasks, as well as any contribution links that may be present in the goal model. Domain experts assign satisfaction probabilities to all the tasks. These probability values are then propagated to all the goals through AND/OR decomposition links and contribution links. [2] formulates the probability that a root goal is satisfied as the conditional probability that its leaf level tasks are satisfied given the satisfaction of all the contribution links. [2] also provides tools to transform goal models into Bayesian Networks, and uses an effective Bayesian solver, JavaBayes [4], to calculate probabilities of goal satisfactions.

Probability values propagate straightforwardly from tasks to goals in a goal model without contribution links (without cycles). The probability that an AND (or OR-) decomposed goal is satisfied is the probability that all (or at least one) of its children are satisfied. Consider the running example in Figure 1. If we assign a satisfaction probability of 0.5 to all the tasks, the probabilities that goals $g4$ and $g5$ are satisfied are 0.75 and 0.25 respectively.

Calculating goal satisfaction probabilities is much more complicated when contribution links are present: contribution links may cause cycles in the goal model and therefore cause conflicts. We say a conflict exists when a goal is both satisfied and denied at the same time. The probability model given in [2] addresses these issues by formulating goal satisfactions/denials as being conditional on all the contribution links being satisfied.

If multiple diagnoses are returned, we find a most probable diagnosis using only the parts of the goal model that contain the failed goals/tasks. Satisfaction probabilities are first propagated from tasks to goals in the failed subtrees. We then use a minimal weight SAT solver [17] to find a most likely diagnosis. The minimal weight SAT solver takes as input, not only a propositional formula Φ, but also a list of integers representing the "weights" of all the goals and tasks in the maximal AND-subtree. We map goal/task satisfaction probabilities to integers, in such a way that higher satisfaction probabilities are mapped to larger integers. The minimal weight SAT solver returns either the minimal weight satisfying truth assignments (if Φ is satisfiable), or UNSAT otherwise. We decode the minimal weight assignment to a most likely diagnosis, which consists of the goals/tasks least likely to have been satisfied.

C. Reconfiguration

We define a configuration as follows:

Definition 3: (**Configuration**) A configuration consists of a set of tasks from a goal model which, when executed successfully in some order, lead to the satisfaction of the root goal (in the goal model).

This section discusses the two main algorithms (Algorithms 2 and 3) of our reconfiguration component. We show in this section that Algorithm 2 is suitable for finding a most optimal global reconfiguration when performance is not a major concern. Algorithm 3 is suitable for fast failure recovery when performance matters. Both algorithms first compute a set of system reconfigurations free of failures. If multiple reconfigurations exist, both algorithms choose one or more best reconfigurations that contribute most positively to system's soft goals, and that reconfigure the system the least from its current configuration. The difference between the two algorithms lies within the first step where the algorithms find a set of reconfigurations free of failures. Algorithm 2 finds all system configurations that are free of failures. It computes reconfigurations to all goals and tasks. Unfortunately the number of configurations increases exponentially with the size of the goal model and the number of alternatives it includes. Algorithm 3 alleviates this problem by finding only the reconfigurations to goals and tasks that had failed. The number of reconfigurations therefore does not necessarily grow with the size of the goal model. Rather it depends on the sub-goal model it searches on. We present and compare experimental results using these two algorithms in Section VI-B.

Algorithm 2 Find Best Global Reconfigurations

```
findGlobalReconfig (goal_model, failures) {
  //find all global reconfigurations
  allReconfigs = all reconfigures that are free of failures
  //apply the first selection criterion
  if (goal_model contains softgoals) {
      for each reconfig in allReconfigs
          score = calConfigScore(goal_model, config)
      bestConfigs = reconfigs with the highest score
  }
  //apply the second selection criterion
  if (multiple reconfigs tie on their scores)
      bestReconfigs = configs that minimally reconfigures
          the system from its current configuration
  return bestReconfigs
```

Algorithm 3 Find Best Local Reconfigurations

```
findLocalReconfig (goal_model, failures) {
  //find all local reconfigurations
  for each denied goal and task {
     reconfigableGoal = the closest OR-decomposed
                ancestor goal to the denied goal/task
     localReconfigs = configs contained under
                reconfigableGoal that are free of failures
  }
  allLocalReconfigs = configs in all localReconfigs
  //apply the first selection criterion
  if (goal_model contains softgoals) {
      for each reconfig in allReconfigs
          score = calConfigScore(goal_model, config)
      bestConfigs = reconfigs with the highest score
  }
  //apply the second selection criterion
  if (multiple reconfigs tie on their scores)
      bestReconfigs = configs that minimally reconfigures
       the system from its current configuration
  return bestReconfigs
```

Algorithm 2 takes as input a goal model and failed goals/tasks. It calculates *globally* all reconfigurations that do not include any failed goals/tasks. If more than one such reconfiguration exists, it applies our two selection criteria to compute the best reconfiguration(s). The first criterion selects configurations that contribute most positively to soft goals. For each reconfiguration, the algorithm calls Algorithm 4 (*calConfigScore*) to calculate a score that indicates how positively the reconfiguration contributes to soft goals. The algorithm chooses the reconfiguration(s) with the highest scores. If multiple "best reconfigurations" tie, it uses the second selection criterion to further prune the list. Criterion two selects configurations that minimally reconfigure the system by choosing configurations that reuse the greatest number of tasks from system's current configuration. Algorithm 2 searches for best global reconfigurations. It may replace goals/tasks that did not fail with alternatives that contribute more positively to soft goals. Algorithm 2 is suitable when we want to find (and/or reset the system to) an optimal initial system configuration - when performance is not a big concern. Due to its poor scalability to the goal model size (as will be shown in Section VI-B), the algorithm is not suitable to be used for fast failure recovery when performance is a major concern.

Algorithm 3 finds local reconfigurations that replace only denied goals and tasks with their alternatives, and selects therefrom the best reconfigurations. For each denied goal/task, the algorithm finds its immediate, or closest, OR-decomposed ancestor goal. Local reconfigurations free of failures are contained under this ancestor goal. Take for example the goal model in Figure 1. Goal $g2$ is the closest OR-decomposed ancestor to task $a3$. If $a3$ fails, the algorithm searches for reconfigurations under the decomposition of $g2$ that are free of failures. Task $a7$ is chosen as the only local reconfiguration to $a3$. If no such OR decomposed ancestor goals exist within the goal model, no reconfigurations exist. If more than one local reconfiguration is found, the algorithm selects the best reconfiguration(s) using the same criteria as Algorithm 2. Algorithm 3 only finds local best reconfigurations that replace *denied* goals/tasks. The number of such reconfigurations does not directly depend on goal model size. It depends rather on where in the goal model the failures have occurred, together with the structure of the goal model. In worst case scenarios, the number of local reconfigurations can be exponential to the sub-goal model the algorithm searches on. In Section VI-B, we show that Algorithm 3 can be used for fast failure recovery for systems with medium-sized requirements.

To calculate each configuration's contribution score to soft goals, we first propagate satisfaction labels from hard goals and tasks in the given configuration to soft goals in a goal model. In other words, we want to compute satisfaction labels for soft goals in a goal model assuming that all hard goals and tasks in a given configuration are satisfied. Axioms 1 to 4 describe label propagations from hard goals and tasks (hG) to soft goals (sG) of a goal model. These axioms specify that if link $++S$ (or $--S$, $+S$, $-S$) proceeds from hG to SG, and if hG is fully satisfied in execution session s, then sG is fully satisfied (or fully denied, partially satisfied, partially denied respectively) in s. The $++D$, $--D$, $+D$, $-D$ cases are dual w.r.t. $++S$, $--S$, $+S$, $-S$ respectively. Axioms 5 to 6 specify that for any soft goal sG, if it is fulled satisfied (or denied) in s, it is also partially satisfied

(or denied respectively) in s.

$$hG \xrightarrow{++S} sG : \neg FD(hG, s) \rightarrow FS(sG, s) \quad (1)$$

$$hG \xrightarrow{--S} sG : \neg FD(hG, s) \rightarrow FD(sG, s) \quad (2)$$

$$hG \xrightarrow{+S} sG : \neg FD(hG, s) \rightarrow PS(sG, s) \quad (3)$$

$$hG \xrightarrow{-S} sG : \neg FD(hG, s) \rightarrow PD(sG, s) \quad (4)$$

$$FS(sG, s) \rightarrow PS(sG, s) \quad (5)$$

$$FD(sG, s) \rightarrow PD(sG, s) \quad (6)$$

Algorithm 4 Calculate a Reconfiguration's Contribution to Soft Goals

```
calConfigScore (goal_model, config) {
  FSValue=1, PSValue=0.5, FDValue=-1, PDValue=-0.5
  score = 0
  propagate satisfaction labels following axioms 1 to 6
  for each softgoal s_i{
    priority = s_i's priority value
    for each of s_i's satisfaction label {
      if (label == FS)
        score += priority × FSValue
      else if (label == PS)
        score += priority × PSValue
      else if (label == FD)
        score += priority × FDValue
      else if (label == PD)
        score += priority × PDValue }}
  return score}
```

Algorithm 4 calculates a contribution score for a given configuration. A configuration's total contribution to all the soft goals is the sum of its contributions to each soft goal. The algorithm propagates satisfaction labels from hard goals/tasks in the given configuration to soft goals by following Axioms 1 to 6. A soft goal has one to four satisfaction labels: FS, FD, PS, and PD. There may be both full and partial evidence that a soft goal is both satisfied and denied in the same execution session s. Therefore, a soft goal can have one or more or even all of the four satisfaction labels associated to it in s. We assign numeric values: 1, 0.5, -1, and -0.5 to FS, PS, FD, and PD respectively. In addition, a soft goal is assigned one of three priority values: *high*, *medium* or *low*. Numeric values of 3, 2, and 1 are used to represent them respectively.

For each such goal s_i, the algorithm obtains its priority value, as well as all of its satisfaction labels. For each of s_i's satisfaction label, the algorithm increases the configuration's score by the product of s_i's priority and the label's numerical value. For example, consider a soft goal sG with high priority and satisfaction labels FS and PD. SG's score is calculated as follows: FSValue $\times$ priority + PDValue $\times$ priority, which is 1 $\times$ 3 + (-0.5) $\times$ 3 = 1.5. Note that soft goals with satisfaction labels of FS and FD only or PS and PD only have satisfaction scores of 0, because the evidences of their satisfaction and denial cancel each other out.

1) Reconfiguring the Running Example: We illustrate our reconfiguration algorithms by applying them on the running example given in Figure 1 of Section II. The goal model in the running example contains five possible configurations. The successful execution of all the tasks in each of the five configurations leads to the satisfaction of the root goal $g1$. These 5 configurations are:

Configuration 1: [a1, a3, a4, a5, a8, a9, a10, a11]
Configuration 2: [a1, a3, a4, a6, a8, a9, a10, a11]
Configuration 3: [a2, a3, a4, a5, a8, a9, a10, a11]
Configuration 4: [a2, a3, a4, a6, a8, a9, a10, a11]
Configuration 5: [a7, a8, a9, a10, a11]

Consider if Configuration 1 is executed, and if $a1$ fails, Algorithm 3 finds one local reconfiguration (Configuration 3) that replaces only the failed task $a1$ with its alternative task $a2$ (i.e. did not replace any task that did not fail). Since there is only one local reconfiguration, no further selection is necessary. Configuration 3 is returned as the best local reconfiguration.

Table I
FINDING A BEST GLOBAL RECONFIGURATION

Soft Goal Priority	Contribution Scores	Best Config.
sG1=medium sG2=medium	config3=1, config4=-2, config5=0	config3
sG1=high sG2=low	config3=0, config4=-1, config5=2	config5

Algorithm 2 is called to find a best global reconfiguration. The algorithm first finds three global reconfigurations that do not include $a1$ (Configurations 3, 4, and 5). It then needs to choose a best configuration among the three. We illustrate the selection process through two experiments. Their experimental results are reported in Table I. These two experiments assign different priority values to the soft goals, $sG1$ and $sG2$, in the goal model. In the first experiment (row 1 of Table I), both $sG1$ and $sG2$ have *medium* priority. In the second experiment (row 2), $sG1$'s priority is *high*, while $sG2$'s priority is *low*. The table's second column lists each global reconfiguration's contribution score to soft goals. The last column lists the chosen configuration. In the first experiment, the scores for configurations 3, 4, and 5 are 1, -2, and 0 respectively. Configuration 3 is therefore chosen. In the second experiment, the contribution scores changed to 0, -1, and 2 respectively, making configuration 5 the best.

We now show how the scores of 0 and 2 were derived for configuration 5. Configuration 5 involves executions of task $a7$ and goal $g7$. One MAKE (shorthand for $++S$ and $++D$) and one BREAK (shorthand for $--S$ and $--D$) contribution link proceed from $a7$ to soft goals $sG1$ and $sG2$, respec-

tively. If $a7$ is fully satisfied in execution session s, the labels for the soft goals are $FS(sG1, s)$ and $FD(sG2, s)$ after the label propagation process. Configuration 5's contribution score is calculated as: *FSValue* $\times$ *priority(sG1)* + *FDValue* $\times$ *priority(sG2)*. In the first experiment, the score comes to 1 $\times$ 2 + (-1) $\times$ 2 = 0, as $sG1$ and $sG2$'s priority values are 2. In the second experiment, the score comes to 1 $\times$ 3 + (-1) $\times$ 1 = 2, as $sG1$ and $sG2$'s priority values are 3, and 1 respectively.

These two experiments show us that the priority values specify which soft goals are more important. These values can guide the reconfiguration algorithms to choose a set of best configurations that contribute most positively to the most important soft goals. Note that if any task under the decomposition of a maximal AND-subtree fails, there are no reconfigurations that can repair its failure. In this example, these "non-repairable" tasks are: $a3$, $a4$, $a8$, $a9$, $a10$, and $a11$.

D. Execution

If failures occur, the *Execution* component first executes compensation actions to bring the system back to a consistent state. Compensation actions are provided by domain experts because they are application dependent. Some tasks do not need to be compensated for. Their respective compensation actions would be empty. Some tasks cannot be compensated for. In these cases, domain experts need to determine if compensation is possible, and what the appropriate compensation behaviors are.

We adopt a model of database long lived transactions (LLTs) for executions of compensation actions [8]. If we think of the root goal g in a goal model as a saga, its leaf level tasks T_1, T_2, ..., T_n are saga g's sub-transactions. Domain experts provide compensation actions C_1, C_2, ..., C_n for all tasks in a goal model $T_1, T_2, \ldots, T_n$ respectively. The execution component ensures that either the sequence T_1, T_2, ..., T_n or the sequence T_1, T_2, ..., T_j, C_j, ..., C_2, C_1 is executed for some $0 \leq$ j $<$ n [8]. Executions of these compensation actions restore a system to its previous consistent state when failures occur.

It is noteworthy that in many applications, a root goal may be decomposed into fairly independent subgoals that do not need to access the same resources. In these cases, it may make sense to compensate for only failed goals/tasks, rather than for all goals/tasks that are successfully completed. Domain experts need to decide on a case by case basis on what the best compensation behaviors for a software systems are.

After executing compensation actions, the executor component reconfigures the monitored system by tapping into the system's own configuration API. The system runs under the new configuration in its next execution session.

V. Implementation Details

The presented framework containing monitoring, diagnosis, reconfiguration, and execution components has been implemented using the Java programming language. The source code contains 14 Java classes with about 7000 LOC. We translated the problem of requirement diagnosis to a SAT problem. The diagnostic component uses SAT4J [15], an efficient SAT solver, by including its jar file in the framework's compile path.

Monitoring specifications and instrumentation code are generated semi-automatically using AspectJ [13]. Aspect Oriented Programming (AOP) is a technology that implements crosscutting concerns in modular units. AspectJ is an aspect oriented extension to Java. The instrumentation component obtains monitoring information, such as "what" to monitor (from the goal model), and "where" to insert the monitors (from the traceability links). Using this information, the instrumentation component can semi-automatically generate monitoring specifications in AspectJ terminologies. The AspectJ compiler then uses these specifications to automatically instrument the software. Interested readers can refer to [27] for a complete account of the instrumentation component.

Higher variability software systems have their own configuration mechanisms. In many cases, the configuration is done through command line arguments, configuration files, or simply by changing the states of program variables. Traceability links connect goals and tasks in a goal model to the methods (or modules) that implement them. The system's own configuration mechanism contains information on how to configure the system so that a certain module will (or will not) be included. The executor component automatically generates an aspect file that contains all the above information - information on how to reconfigure a system so that a certain goal or task will be executed. The executor component then taps into the system's configuration mechanism to effect any necessary configuration changes.

VI. Evaluation

In [23], [24], we evaluated the correctness and performance of our monitoring and diagnostic components, demonstrating their scalability with respect to goal model size. In this section, we evaluate our reconfiguration component. We illustrate our framework by applying it to an ATM (Automated Teller Machine) simulation [1] and show that it transforms the original ATM into an adaptive system with self-reconfiguration capabilities. We then evaluate the performance of our reconfiguration component by applying it to randomly generated goal models (the largest goal model contains 1000 goals and tasks). The experimental results demonstrate the scalability of our solutions. As a result, we conclude that our framework can be applied to industrial software applications with medium-sized goal graphs. All experiments reported were performed on a machine with an

Athlon X2 5000+ CPU with 2GB of RAM. The operating system used was Ubuntu Linux 8.10.

A. ATM Case Study

The ATM simulation case study is an illustration of OO design used in a software development class at Gordon College [1]. The application simulates an ATM performing customers' *withdraw*, *deposit*, *transfer* and *balance inquiry* transactions. Its original source code contains 36 Java Classes with 5000 LOC. In this section, we illustrate reconfiguration Algorithms 2 and 3. We also show that our framework transforms the original ATM software into an adaptive system with self-reconfiguration capabilities.

The implementation of the original ATM was extended to include alternative behaviors for the purpose of making reconfiguration possible. Figure 3 shows a partial goal graph of the extended ATM, with 19 goals and 26 tasks. The priority levels for all soft goals are set to *medium*. Four major alternative behaviors were added: (1) the original ATM asks the operator to manually enter ATM cash amount (represented by task $a3$); the extended ATM offers a cash sensor that automatically senses the cash amount (represented by task $a2$). (2) The original ATM asks customers to insert their bank cards ($a7$); the extended ATM allows a customer to manually enter her card number if she does not have the physical card with her ($g7$). (3) The original ATM takes customers' input from a full-keyed physical keypad ($a5$ and $a9$); the extended ATM adds a backup two-key keypad so that it can be used if the full-keyed keypad fails ($a6$ and $a10$). (4) The original ATM prints customers' receipts ($a21$); the extended ATM displays the receipt if its printer fails ($a22$).

Table II
CHOOSING A BEST GLOBAL RECONFIGURATION FOR ATM

No.	Configurations	Score
1	**a1, a2, a4, a5, a8, a9, a11, g13, a21, a23, g19**	**1**
2	a1, a2, a4, a6, a8, a9, a11, g13, a21, a23, g19	0
3	a1, a2, a4, a5, a8, a9, a11, g13, a22, a23, g19	1
4	a1, a2, a4, a6, a8, a9, a11, g13, a22, a23, g19	0
5	a1, a2, a4, a5, a8, a10, a11, g13, a21, a23, g19	0
6	a1, a2, a4, a6, a8, a10, a11, g13, a21, a23, g19	-1
7	a1, a2, a4, a5, a8, a10, a11, g13, a22, a23, g19	0
8	a1, a2, a4, a6, a8, a10, a11, g13, a22, a23, g19	-1
9	a1, a3, a4, a5, a8, a9, a11, g13, a21, a23, g19	1
10	a1, a3, a4, a6, a8, a9, a11, g13, a21, a23, g19	0
11	a1, a3, a4, a5, a8, a9, a11, g13, a22, a23, g19	1
12	a1, a3, a4, a6, a8, a9, a11, g13, a22, a23, g19	0
13	a1, a3, a4, a5, a8, a10, a11, g13, a21, a23, g19	0
14	a1, a3, a4, a6, a8, a10, a11, g13, a21, a23, g19	-1
15	a1, a3, a4, a6, a8, a10, a11, g13, a22, a23, g19	0
16	a1, a3, a4, a6, a8, a10, a11, g13, a22, a23, g19	-1

The framework first calls Algorithm 1 (inside the monitoring component) to compute the optimal monitoring granularity. Algorithm 1 selects goals $g13$ and $g19$, which are roots of maximal AND-subtrees, for monitoring. Tasks not under $g13$ and $g19$ are also monitored. The instrumentation component instruments the ATM at the appropriate places using AspectJ technologies [27]. At run time, the ATM produces log data that are analyzed by the diagnostic component to infer requirement denials. Consider a scenario where a configuration containing goals/tasks [*a1, a2, a4, a7, a8, a9, a11, g13, a21, a23, g19*] is executed. An error was injected into the execution of task *a7*. The diagnostic component returns a single diagnosis containing $FD(a7, s)$, indicating that $a7$ is fully denied in session s.

The framework calls Algorithm 2 to find all global reconfigurations. Table II reports the results. Algorithm 2returns a total of 16 global reconfigurations that do not include the failed task. The first, second, and last columns of the table list respectively the reconfiguration number, the actual reconfiguration, and its contribution score. The listed reconfigurations contain both goals and tasks (instead of all tasks) for readability. Reconfigurations 1, 3, 9, and 11 tie on their contributions to soft goals, with scores of 1. The algorithm then applies the second selection criterion to choose a best reconfiguration. This second criterion seeks the reconfiguration that minimally reconfigures the system from its current configuration. The first configuration is selected because it replaces failed task $a7$ with its alternative task $a5$ and reuses all other goals/tasks from the current configuration.

Under the same failure scenario, Algorithm 3 finds 2 local reconfigurations: reconfigurations 1 and 2 that replace $a7$ with $a5$ and $a6$ respectively. Reconfigurations 1 and 2 have a contribution score of 1 and 0 respectively. As a result, reconfiguration 1 wins out as the best alternative.

In this example, Algorithms 2 and 3 returned the same best reconfiguration because the best global and local reconfiguration happens to be the same. In its next execution, the ATM runs with a new configuration free of failed components.

B. Performance Evaluation

To evaluate the scalability of our reconfiguration component, we performed 20 experiments on 20 progressively larger goal models containing from 50 to 1000 goals and tasks. Our program generated these goal models randomly. For each node in the goal model, our program randomly decides: (1) whether it is a goal or a task, (2) whether it is AND- or OR- decomposed, and (3) which goal is its parent.

We performed these experiments using both reconfiguration Algorithms 2 and 3 to compare their efficiency on larger goal graphs. These experiments show that the reconfiguration component scales well with the size of the goal model when it searches for the best local (instead of global) reconfigurations (Algorithm 3). In [23], [24], we demonstrated the scalability of our monitoring and diagnostic components. As a result, we conclude that our

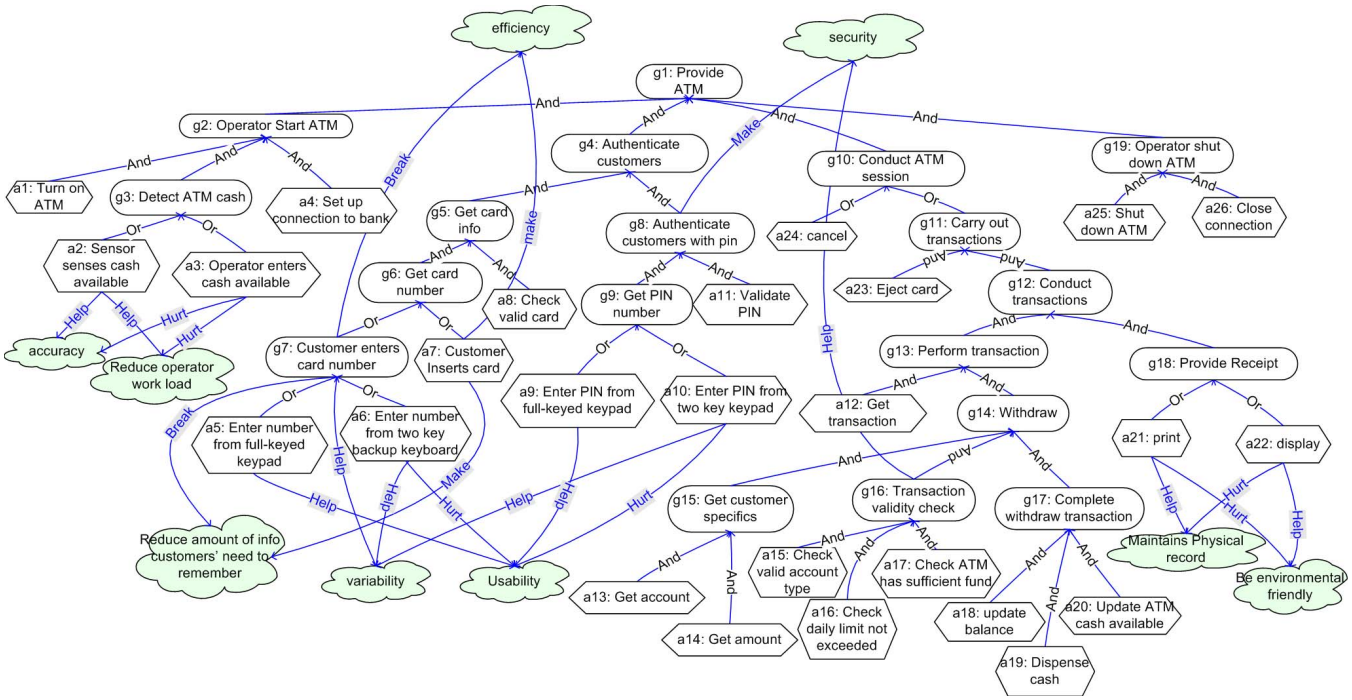

Figure 3. Extended ATM Goal Model

Table III
FINDING GLOBAL VS. LOCAL RECONFIGURATIONS

#Goals &Tasks	#Global Configs	Time (s) (Global)	#Local Configs	Time (s) (Local)
50	57	0.149	14	0.004
100	6045	2.26	16	0.004
150	30308	15.986	9	0.001
200	24523	19.874	11	0.002
250	22463	21.485	107	0.019
300	out	out	9	0.031
350			2	0.031
400			13	0.054
450			13	0.043
500			10	0.030
550			7	0.030
600			7	0.031
650			9	0.040
700			3	0.024
750			9	0.031
800			13	0.045
850			8	0.038
900			9	0.043
950			9	0.034
1000			14	0.042

framework can be applied to industrial software applications with medium-sized requirement models.

Table III and Figure 4 report experimental results. Each row in Table III corresponds to the average results of 10 experimental runs with equal goal model size. In each experiment, the framework randomly selects one task to

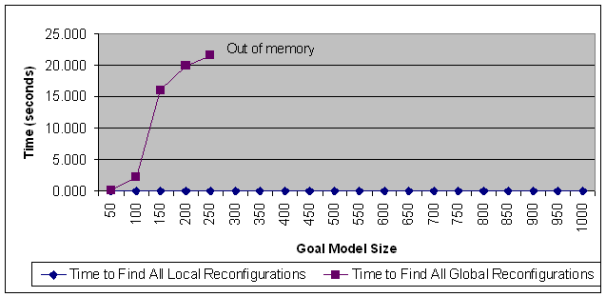

Figure 4. Scalability Comparison

fail, and the reconfiguration component calls Algorithms 2 and 3 to find best global and local reconfigurations respectively. Column 1 lists the number of goals/tasks in the goal graph. Columns 2 and 3 list the total number of global reconfigurations returned by Algorithm 2, and the total time taken (in seconds) to find them, respectively. Columns 4 and 5 list the total number of local reconfigurations returned by Algorithm 3 and the time taken (in seconds) to find them, respectively.

Figure 4 depicts the relationships between the total time taken (in seconds) for finding global and local reconfigurations (the y-axis - the values in columns 3 and 5 of Table III) and the goal model size (the x-axis - the values of column 1 of Table III). As discussed in Section IV-C, the number of global reconfigurations grows exponentially with respect to the size of the goal model; an "out of memory"error

was returned with experiments on goal models containing more than 300 goals/tasks. In contrast, the number of local reconfigurations and the time taken to find them do not necessarily increase with increasing goal model size. Rather, they depend on the structure of the sub-goal graph in which failures occur.

VII. Related Work

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

Robinson also presented a requirement monitoring framework, ReqMon [20]. If an observed event is a satisfaction (or violation) event, satisfaction (or denial) status is updated for the requirement. ReqMon requires diagnostic formulae to be generated manually using obstacle analysis [14]. Our work, on the other hand, makes assumptions about what can fail. This allows our framework to automatically infer diagnoses given a model of system requirements and log data. Moreover, the ReqMon tools do not offer reconciliation mechanisms when requirements are violated.

Salifu et. al presented a formal approach for analyzing the impact of changes in a software system's context (environment) on the satisfaction of the system's requirements [21], [22]. The impact of varying contextual properties on monitoring and switching was studied. Monitoring aims to detect changes in the system's context. If changes are detected, the software system's requirements are violated. Behavior switching specifications specify new behaviors for the system to switch to. The main difference between our work and Salifu et. al's is that their approach focuses on monitoring requirements violations caused by changes in the system's context, while our work diagnoses failures within the software system itself.

Li et. al. presented a self-reconfiguration system for service-oriented systems [16]. A software system's non-functional requirements are represented as service level agreements (SLAs). Reconfiguration is triggered if either an SLA is violated, or if a resource over- or under- consumption is detected. Our work differs from Li et. al's in that their approach monitors the satisfaction of a system's non-functional requirements. Consequently, the system reconfigures for the purpose of self-optimization. Our work monitors the satisfaction of a system's functional requirements. The system reconfigures for the purpose of failure recovery.

Many researchers favor an architectural approach to software adaptation. For instance, Garlan and Cheng et. al. presented a framework that uses an architectural model to monitor a system's running properties and performs adaptation if system constraints are violated [9]. Gomaa and Hussein [11] proposed the use of software configuration patterns to support dynamic reconfiguration in software product families. Oreizy et. al [19] provided an outline of an architectural approach that allows for adaptation and evolution management. Our work defers from these approaches in that we use a system's requirements as a basis for monitoring, diagnosis and adaptation. None of the research discussed above ([9], [11], [19]) presented framework performance evaluations or discussed scalability issues. It is therefore difficult to compare our respective approaches in terms of performance and scalability.

VIII. Conclusion

We have presented an implemented framework that enables high variability systems to self-reconfigure when failures occur. We have also evaluated our framework through a series of experiments. The results of our experiments suggest that it is feasible to scale our approach to industrial-size software with medium-size requirements.

Before concluding, we note some features and limitations of our proposal. The proposed framework depends on the availability (and correctness) of requirements goal model and traceability links. It is challenging for domain experts to provide a fine grained goal model for large software systems. Fortunately, the hierarchical and layered structure of goal models enables us to model (and analyze) a software system at different levels of granularity. A large-scale, complex software system can be modeled using few goals and tasks at a high level of granularity. In cases where a fine-grained goal model is not available, the system can be modeled with a few high level goals/tasks with relatively less effort. Traceability links are required to map requirements to the monitored system's source code. Traceability links may also be represented at different granularity levels. If low-level, detailed traceability links are not available, higher level traceability links can be used to relate high level goals to larger-scaled sub-systems of the monitored system.

The worst case performances of both global and local reconfiguration algorithms (Algorithms 2 and 3 respectively) can be exponential to the size of goal model they search on. The performance of Algorithm 2 (global) can be exponential to the size of the entire goal model. The performance of Algorithm 3 (local) can be exponential to

the partial goal model it searches on. As future work, we plan to design better reconfiguration algorithms that find best global reconfigurations with linear time performance. For evaluation, we experimented on randomly generated goal models, instead of goal models of real systems. The largest goal model we experimented on contained 1000 goals and tasks. It would have been difficult to obtain real world goal models of this scale. As further work, we plan to evaluate our approach on real life applications, especially large scale applications. Moreover, our framework monitors the satisfaction of software system functional requirements. In future, we plan to extend our framework to monitor for other kinds of failures, such as violations of software systems' non-functional requirements and domain assumptions. Finally, our repair/reconfiguration component is limited to high variability software systems that have alterative ways of fulfilling their requirements. As future work, we also plan to explore other repair methodologies that do not depend on reconfiguration. We will thereby extend our framework to design systems with higher levels of self-management capabilities such as self-optimization, self-healing, and self-protection.

REFERENCES

[1] R. Bjork. An example of object-oriented design: an atm simulation. http://www.cs.gordon.edu/courses/cs211/ ATMExample/index.html/, 2007.

[2] Borys J. Bradel. Extending goal models with a probability model and using bayesian networks. In *International Conference on Software Engineering Research and Practice*, 2009.

[3] B. Cheng, R. de Lemos, D. Garlan, H. Giese, M. Litoiu, J. Magee, H. Muller, and R. Taylor. SEAMS 2008: software engineering for adaptive and self-managing systems. In *SEAMS*, 2008.

[4] F. G. Cozman. The javabayes system. *The ISBA Bulletin*, 7(4):16–21, 2001.

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

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

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

[8] H. Garcia-Molina. Sagas. In *COMAD*, 1987.

[9] D. Garlan, S.W. Cheng, A.C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37:46–54, 2004.

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

[11] H. Gomaa and M. Hussein. Dynamic software reconfiguration in software product families. In *5th International Workshop on Software Product-Family Engineering*, 2003.

[12] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer Socicety Press*, 36:41–50, 2003.

[13] G. Kiczales, J. Kersten M. Hilsdale, E. Hugunin, J. Palm, and W. Griswold. *An Overview of AspectJ*. Springer Berlin, 2001.

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

[15] D. Le Berre. A satisfiability library for java. http://www.sat4j.org/, 2007.

[16] Y. Li, K. Sun, J. Qiu, and Y. Chen. Self-reconfiguration of service-based systems: a case study for service level agreements and resource optimization. In *ICWS*, 2005.

[17] P. Liberatore. Algorithms and experiments on finding minimal models. Technical report, University of Rome, 2000.

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

[19] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. L. Wolf. An architeture-based approach to self-adaptive software. *Intelligent Systems and Their Applications*, 14:54–62, 1999.

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

[21] M. Salifu, Y. Yu, and B. Nuseibeh. Specifying monitoring and switching problems in context. In *15th IEEE International Requirements Engineering Conference*, 2007.

[22] M. Salifu, Y. Yu, and B. Nuseibeh. Analysing monitoring and switching requirements using constraint satisfiability. Technical report, The Open University, 2008.

[23] Y. Wang, S. McIlraith, Y. Yu, and J. Mylopoulos. An automated approach to monitoring and diagnosing requirements. In *ASE*, 2007.

[24] Y. Wang, S. McIlraith, Y. Yu, and J. Mylopoulos. Monitoring and diagnosing software requirements. *Journal of Automated Software Engineering*, pages 3–35, 2009.

[25] Y. Yu, A. Lapouchnian, S. Liaskos, J. Mylopoulos, and J.C.S.P. Leite. From goals to high-variability software design. In *ISMIS*, 2008.

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

[27] X. Zhou. A goal-oriented instrumentation approach for monitoring requirements. Master's thesis, Univeristy of Toronto, 2008.