Stateful Requirements Monitoring for Self-Repairing Socio-Technical Systems

Lingxiao Fu[1], Xin Peng[1], Yijun Yu[2], John Mylopoulos[3], Wenyun Zhao[1]
[1] *School of Computer Science, Fudan University, China*
[2] *Department of Computing, The Open University, UK*
[3] *Department of Information Engineering and Computer Science, University of Trento, Italy*
{*09210240010, pengxin, wyzhao*}*@fudan.edu.cn, y.yu@open.ac.uk, jm@disi.unitn.it*

***Abstract*—Socio-technical systems consist of human, hardware and software components that work in tandem to fulfill stakeholder requirements. By their very nature, such systems operate under uncertainty as components fail, humans act in unpredictable ways, and the environment of the system changes. Self-repair refers to the ability of such systems to restore fulfillment of their requirements by relying on monitoring, reasoning, and diagnosing on the current state of individual requirements. Self-repair is complicated by the multi-agent nature of socio-technical systems, which demands that requirements monitoring and self-repair be done in a decentralized fashion. In this paper, we propose a stateful requirements monitoring approach by maintaining an instance of a state machine for each requirement, represented as a goal, with runtime monitoring and compensation capabilities. By managing the interactions between the state machines, our approach supports hierarchical goal reasoning in both upward and downward directions. We have implemented a customizable Java framework that supports experimentation by simulating a socio-technical system. Results from our experiments suggest effective and precise support for a wide range of self-repairing decisions in a socio-technical setting.**

***Keywords*-self-repair, requirements monitoring, goal models**

I. Introduction

Socio-technical systems consist of human, hardware and software components that work in tandem to fulfill stakeholder requirements. By their very nature, such systems operate under uncertainty as components fail, humans act in unpredictable ways, and the environment of the system changes. Self-repair refers to the ability of such systems to restore fulfillment of their requirements by relying on monitoring, reasoning, and diagnosing on the current state of individual requirements. Self-repair is complicated by the multi-agent nature of socio-technical systems, which demands that requirements monitoring and self-repair be done in a decentralized fashion. For example, for an online product shipping system, if a delivery man forgets to check the shipping address, a product may not be delivered before Christmas. To compensate for this omission and satisfy customer expectations, a second shipment may be generated from the nearest warehouse to the customer's home.

Existing research on self-repairing has proposed several methods for issues related to requirements monitoring [1], reasoning [2], and diagnosing [3] and self-reconfiguration [4], [5], [6]. However, little is published on how to bridge the gap between the deviations detected and the repairing actions required by specifying precise application-specific self-repairing policies. Nor are there any proposals for an integrated and reusable infrastructure for developing such self-repairing systems. Also, existing proposals generally do not support decentralized requirements monitoring and self-repairing for socio-technical systems.

To address the above problems, we propose a fine-grained and stateful requirements monitoring approach towards goal fulfillment. Rather than treating the fulfillment of a goal as a binary transition from an initial state to either an achieved or a failed state, a richer set of states is presented such that it is possible (1) to specify repairing policies as prevention, retry and compensation, upon the specification deviation of a goal at any state; (2) to reason about the interaction between fulfillment and repairing of individual goals through appropriate propagations and substitutions; and (3) to support decentralized requirements monitoring and repairing using inter-agent interactions.

Following model-based adaptation [6], [7], our approach maintains and manages a set of externalized instances for requirements goals at runtime. To support monitoring and repairing of the proposed stateful goals, we use state machines to model the runtime lifecycle by representing their fulfillment by a richer set of states and transitions. Our approach manages the runtime lifecycle of each goal by defining both external and internal event mappings between the state machines of different goals and by enforcing appropriate goal state transitions accordingly at runtime. External event mappings reflect the observed behaviors from the target system, while internal event mappings embody the interactions among different goal state machines. Based on the stateful goal monitoring, we provide a fine-grained self-repairing algorithm that combines local repairing policies such as prevention, retry and compensation, with global repairing policies including goal and agent substitutions. By the nature of isomorphic goal state machines and event-driven interactions, our approach can also support decentralized requirements monitoring and repairing.

To evaluate the effectiveness of our approach, we implemented a customizable prototype of the reasoning framework for the goal state machine and interactions. Using the prototype, we conducted an experimental study with a simu-

lated socio-technical system developed using AnyLogic [8]. The study shows that our approach can support a wide range of self-repairing decisions. Moreover, the results confirm that system failures caused by requirements deviation are reduced with an acceptable performance overhead.

The rest of the paper is structured as follows. Section II illustrates the basic ideas of our approach with a small real-life example. Section III presents the proposed approach, including an extended goal state machine and the corresponding monitoring and the self-repairing mechanisms. Section IV evaluates the proposed approach using our prototype from both qualitative and quantitative perspectives. Section V compares our work to a number of existing proposals, and section VI draws a final conclusion.

II. An Illustrating Example

To illustrate the essence of stateful requirements monitoring and the self-repairing mechanisms, we use an Order & Delivery scenario adapted from real-life commodity shopping and logistics systems.

A. Requirements Models

The Order & Delivery scenario involves two agents, i.e. **Commodity Ordering System** and **Regional Distribution Center**, and the simplified requirements of which are described by the goal model in Figure 1. There are three types of goals in the figure, i.e. **root goals** of agents, **subgoals** refined from higher-level goals by AND/OR decompositions (we assume that a goal cannot be both AND- and OR-decomposed), and **tasks** representing concrete realization of goals. Each root goal is iteratively refined by AND/OR decompositions all the way down to leaf level goals to be further refined into tasks of the same agent or into goals delegated to other agents.

The root goal of a **Commodity Ordering System**, "Order Commodity", is to offer customers what they order. It is further refined into three subgoals including "Choose items", "Pay Orders", and "Deliver Commodity". The fulfillment of commodity delivery is actually delegated to the **Regional Distribution Center** agent. The assigned distribution center will then proceed to warehouse, sort, allocate, and dispatch the shipment along with a signed receipt.

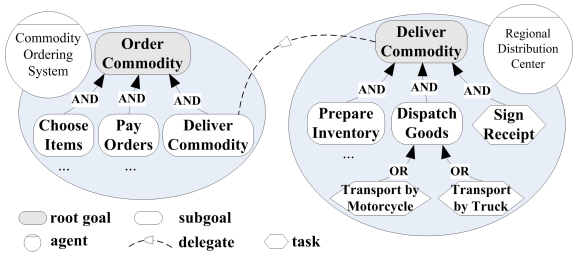

Figure 1. Partial Order & Deliver Goal Model

B. Goal State Machines

Consider the process of goal fulfillment using the example of "Deliver Commodity". Initially for a new commodity delivery task, the goal "Deliver Commodity" is not activated until a **Regional Distribution Center** accepts the assignment. The execution of commodity delivery is not started with the activation of the corresponding goal, instead, it is started when the distribution center dispatches the current handling task and begins to prepare an inventory for it. The delivery can also suspend, e.g. when interrupted by more urgent tasks, and then resume after all required resources are available. The execution process ends when all the relevant sub-tasks have finished. Note that the whole process of the goal achievement is not always linear but is possible to shift back and forth between different phases. Such stateful process of requirements fulfillment in terms of goals of all kinds can be generalized by the state machine in Figure 2.

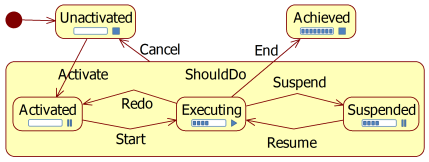

Figure 2. Elementary Goal State Machine

Rather than directly specify the behaviors of target agent (e.g., a software system or a human being), the state machine describes a general process of goal fulfillment from an external perspective. The current state of a goal indicates the satisfaction status of the requirement currently. In general, one can represent the fulfillment process of each goal (including task) with an instance of a goal state machine. The state transitions can be mapped to external events from the target systems, e.g. the *Activate* event of "Deliver Commodity" maps to the acceptance of an assigned delivery task. The state machine instances of different goals are not necessarily independent, they can interact with each other, which can be done by the mapping and propagation of the events through state transitions among different goal instances. For example, the *Start* event of "Prepare Inventory" is mapped to that of its parent goal "Deliver Commodity", making that the latter is started once the former is started. Intuitively, some semantics of goal decomposition are reflected by the interactions of goal state machines. For example, the achievement of an AND-decomposed goal implies the achievement of all its subgoals (tasks); and the achievement of an OR-decomposed goal implies the achievement of any of its subgoals (tasks).

Several constraints exist at varying stages of the goal lifecycle: *context condition* expresses when certain alternatives, i.e. subgoals of OR-decompositions, are applicable [6]; *precondition* specifies the conditions that should be satisfied when a goal is about to start; *postcondition* defines the conditions that should be satisfied when the execution of a goal is over; *commitment condition* determines a time

limit within which an activated goal committed by an agent should be achieved. For example, the context condition "goods smaller than $1M \times 0.5M \times 0.5M$" of task "Transport by Motorcycle" specifies the size of goods transportable by a motorcycle. *Context condition* and *precondition* have different implications from the perspective of system monitoring and repairing. A violation of *precondition* means some necessary conditions for task execution are not satisfied currently, but may be satisfied later. For example, "Transport by Motorcycle" has the precondition of "address attached on the package", which can be satisfied later after the workers print and attach the address label. In contrast, a violation of *context condition* means that a goal commitment is meaningless or disallowed, e.g. committed to transport goods larger than the prescribed size.

Apart from these constraints associated with goal state transition, there is also an *invariant condition* that should be kept satisfied in the whole process of goal fulfillment (i.e., maintain goals). For example, the package of the goods should be kept intact in the whole process of delivery.

C. The Repairing Process

The elementary goal state machine in Figure 2 shows a fail-free process of goal fulfillment. However, if any of the above constraints is violated within the lifecycle, a requirements deviation occurs. To avoid possible failures, proper repairing measures should be taken after deviations were detected. Figure 3 illustrates an ideal example of the self-repairing process for a requirements deviation rooted in the goal "Transport by Motorcycle". Goal "Deliver Commodity" is initially delegated to one **Regional Distribution Center** RDC-1. The labeled arrows represent the actions and their orders in the repairing process that first tries to correct the deviation locally, within the current goal (Action 1). For example, a gate guard can check the precondition "address attached on the package" when a delivery man drives a motorcycle to start a delivery task. When the guard finds the precondition not satisfied, he can prevent the delivery man from starting the task.

If all the local repairing actions fail, e.g. goods still not delivered after retrying more than 3 times, then a failure of "Transport by Motorcycle" is propagated to its parent goal "Dispatch Goods" (Action 2). As it has another alternative subgoal, a goal substitution repairing (Action 3) is then conducted to deliver the goods by truck. Similarly, in the lifecycle of "Transport by Truck", some requirements deviations and local repairing may also occur (Action 4). When it chooses to propagate the local failure (Action 5), its parent goal has to further propagate the failure to "Deliver Commodity" (Action 6), since both of the two alternative subgoals have failed. For the goal "Deliver Commodity", as it is the root goal of **Regional Distribution Center**, the failure is propagated to the delegating agent **Commodity Ordering System** (Action 7). Finally, if possible, the **Commodity Ordering System** may choose to delegate the current delivery task to another **Regional Distribution Center** RDC-2 (Action 8).

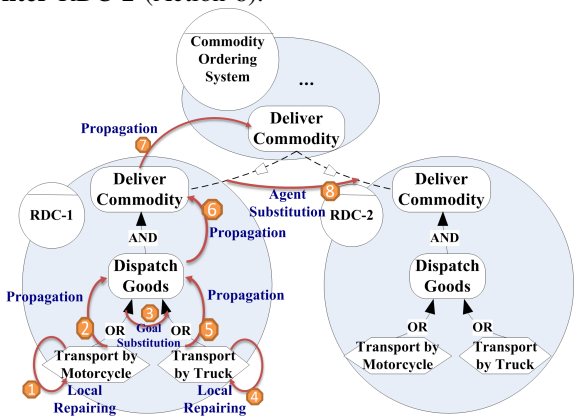

Figure 3. Hierarchical Repairing Process

By clarifying the runtime lifecycles of goals and the interactions between them, system requirements at runtime can be precisely and comprehensively monitored, thus enabling more precise and proper self-repairing actions.

III. Our Approach

Our monitoring and repairing approach is overviewed in Figure 4. Deployed to a host agent defining a goal model, our framework maintains a set of individual goal instances whose runtime lifecycle is managed by an extended goal state machine (see Section III-A). The transitions of goal instances are driven by external events captured from the corresponding host agents and internal events produced by other related goal instances through state machine interactions (see Section III-C). The external and internal events are distributed and mapped according to the general and application-specific event mapping rules (see Section III-B). In the process of goal lifecycle management, our framework checks the constraints related to different states and transitions, and takes proper repairing measures when deviations to requirements are detected (see Section III-D). The repairing measures are conducted hierarchically as shown in Figure 3. The mechanism is distributed among different agents. Cross-agent delegations and failure propagations are supported by state machine interactions, supporting multi-agent socio-technical systems.

A. Extended Goal State Machine

To facilitate the monitoring and repairing of possible requirements deviations, we extend the elementary goal state machine in Figure 2 with states, actions and transitions for monitoring and repairing, as shown in Figure 5. Besides those goal states in the elementary goal state machine, additional states are further introduced:

- **ProgressChecking** is a temporary state that infers goal's next state by state reasoning after receiving some events from the subgoals and delegate goals.

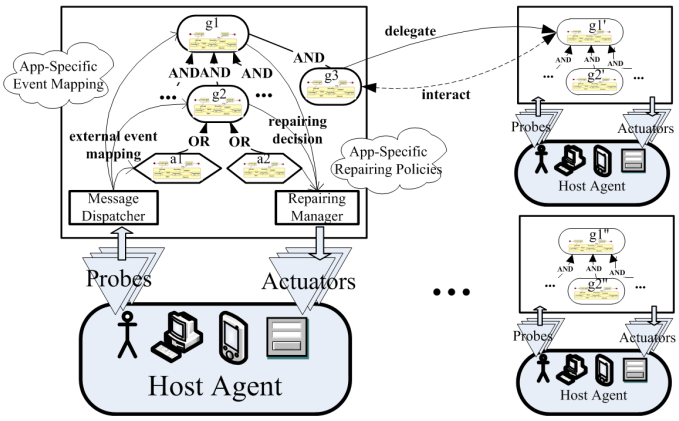

Figure 4. An overview of our approach

- **Failed** states that the current goal instance has failed and the failure is propagated to upper goals to conduct repairing at higher levels. Any goal became *Failed* should perform an application-specific *rollback* operation to eliminate the undesired effects it had brought.
- **Repairing** is also a temporary state in which requirements deviations are captured and under repairing. If repairing succeeds, the current goal instance will transit to a proper state, otherwise it will still fail.

With presence of these states, the elementary transitions are further refined and extended to support requirements monitoring and repairing. First, checking for context conditions, preconditions, and postconditions are added to the *Activate*, *Start*, and *End* transitions, respectively. If these conditions are not satisfied, requirement deviations occur and the goal state will be transferred to **Repairing**. Second, when a goal instance is in the *ShouldDo* state, its *commitment condition* and *invariants* are monitored. Once a *commitment condition* or *invariant* violation is detected, a **CCViolated** or **InvViolated** event is triggered, respectively, and transit the goal state to *Repairing*. A goal's commitment condition is usually associated with a specific time event, either a time period reminder or a particular state machine event of other goals, that is managed and checked by a *Timer* object bound to the goal instance. The timer is created when a goal instance is activated and then it constantly verifies the commitment condition (see the *setTimer* and *commitmentConditionCheck* actions at *ShouldDo* state). The invariants monitored include both constraints on system properties and temporal rules on events sequence.

Some transitions in the extended goal state machine are related to state interactions among different goal instances. These include state transition events from subgoals and delegated goals (event names prefixed with "Sub" and "Delegate", respectively). These events are triggered by the entry actions of the corresponding states of relevant subgoals and delegated goals, see those "trigger" actions.

The extended goal state machines framework is universal for monitoring and repairing requirements goals. Although the consistency rules among the states of related goals (see Table I) apply to all applications, the state machines need be customized for specific applications by defining the internal and external event mappings (see Section III-B). To enable self-repairing, application-specific actuators need to be implemented and integrated with the framework (see Section III-D).

B. Event Mapping Rules

The events that drive the goal state transitions, e.g. *Activate*, *Start*, *End*, have two different sources, by which we classify them as external and internal. External events are mapped from the monitored events of the target systems, indicating the occurrence of specific behaviors of concern. For example, the *Activate* event of "Deliver Commodity" can be mapped to an observable external event of "accept an assigned delivery task"; and the *Start* event of "Transport by Motorcycle" can be associated with the event of "a delivery man departs by motorcycle". These external events are captured and dispatched to the exact state machine instances according to additional context information such as the session ID. On the contrary, internal events are mapped and passed among different goal instances, reflecting the state interactions of them. For example, the *Start* event of "Dispatch Goods" can be mapped from that of its OR-task "Transport by Motorcycle".

Table I
CONSISTENCY RULES OF GOAL STATES

1	$ANDDecomposed(g) \wedge g.state = Achieved \leftrightarrow \forall s \in \{s \vert Goal(s) \wedge Decomposed(s,g)\} s.t. (s.state = Achieved)$
2	$ORDecomposed(g) \wedge g.state = Achieved \leftrightarrow \exists s$ s.t. $(Goal(s) \wedge Decomposed(s,g) \wedge s.state = Achieved)$
3	$ANDDecomposed(g) \wedge g.state = Failed \leftrightarrow \exists s$ s.t. $(Goal(s) \wedge Decomposed(s,g) \wedge s.state = Failed)$
4	$ORDecomposed(g) \wedge g.state = Failed \leftrightarrow \forall s \in \{s \vert Goal(s) \wedge Decomposed(s,g)\}$ s.t. $(\neg context\_cond(s) \vee s.state = Failed)$
5	$Goal(g) \wedge g.state = Unactivated \rightarrow \forall s \in \{s \vert Goal(s) \wedge Decomposed(s,g)\} s.t. (s.state = Unactivated)$
6	$Goal(g) \wedge g.state = Activated \rightarrow \forall s \in \{s \vert Goal(s) \wedge Decomposed(s,g)\}$ s.t. $(s.state = Activated \vee s.state = Unactivated)$
7	$Goal(g) \wedge g.state = ShouldDo \leftarrow \exists s$ s.t. $(Goal(s) \wedge Decomposed(s,g) \wedge s.state = ShouldDo)$
8	$Goal(g) \wedge g.state = Executing \leftrightarrow \exists s$ s.t. $(Goal(s) \wedge Decomposed(s,g) \wedge s.state = Executing)$
9	$Goal(g) \wedge g.state = Suspended \rightarrow \forall s \in \{s \vert Goal(s) \wedge Decomposed(s,g)\}$ s.t. $(\neg s.state = Executing)$

Both external and internal event mappings should conform to the semantics of goal models as well as the application-specific business rules. Table I lists the first-order logic rules that define the state consistency among goals according to the goal modeling semantics, especially those of AND/OR decompositions. The type checking function $Goal(g)$ indicates that g is a goal or a task. Functions $ANDDecomposed(g)$ and $ORDecomposed(g)$ return true when the goal g is refined by AND/OR decomposition, respectively. And function $Decomposed(s,g)$ tells that s is

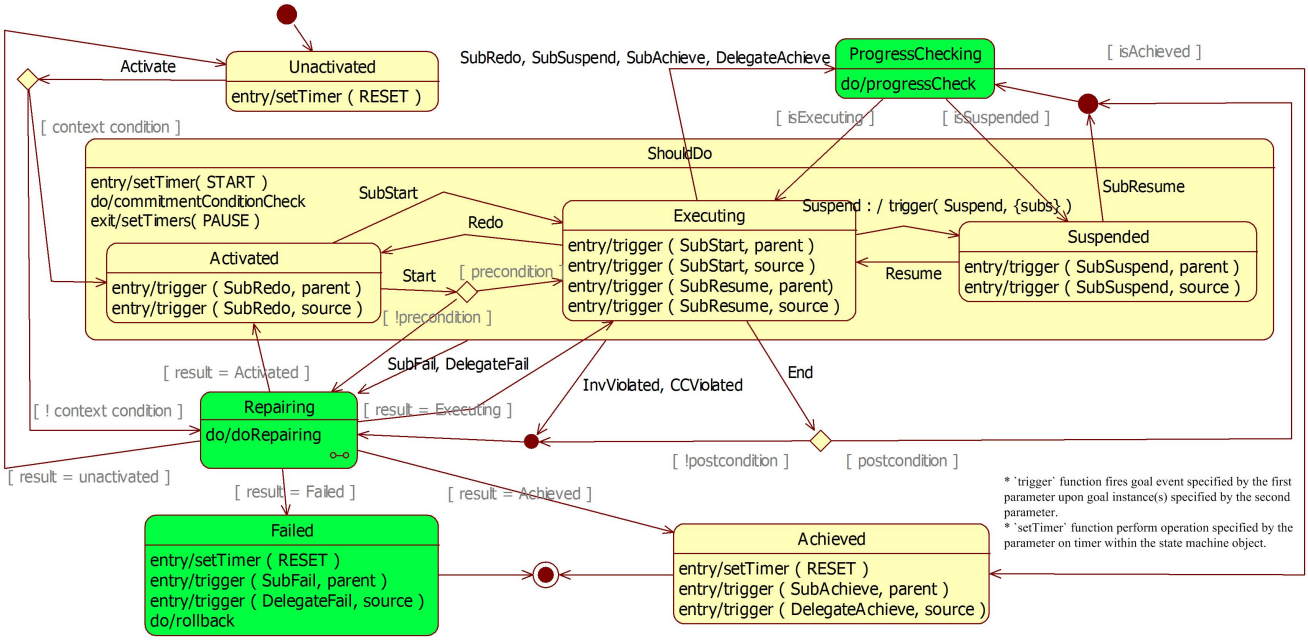

Figure 5. Extended Goal State Machine for Requirements Monitoring and Repairing

a subgoal or task of g. The state consistency rules between OR-decomposed parent and subgoals would also extend to delegate source goal and its delegate candidates, only that a valid delegate relation is established dynamically. The rules listed in Table I only apply to non-temporary states, i.e., all the goal states except for "ProgressChecking" and "Repairing".

We can then define event mapping rules describing the required mapping sources of state transition events for each kind of goals (and tasks). External event mappings are specific to the application between goal transition events and observable events from the target system. Internal event mappings are categorized into general ones and application-specific ones. The former reflect general goal state interactions defined on the rules in Table I. In contrast, the latter are internal event mappings reflecting application-specific business logic. Our event mapping rules reveal a series of rationales specific to different goal/task types. For example, root goals and OR-subgoals/tasks are activated explicitly by external events as they reflect the choices of the target system; activation events of AND-subgoals are always mapped from other goals (in most cases their parent goals or sibling goals), reflecting application-specific commitment propagation; tasks as concrete means for goal fulfillment take over most of the external events. Readers can refer to our technical report [9] for detailed description about event mapping rules.

C. Goal State Machine Interactions

Goal state machine interactions are embodied in event propagation among different goal instances and relevant state reasoning (see the **ProgressChecking** state in Figure 5). Event propagations are needed by both general and application-specific internal event mappings. Typically, some goal instances have their states changed driven by external events, then their state transition events are mapped to other goal instances. This event propagation may trigger further state transitions of other goal instances in a chain-reaction manner. An example of goal state interactions by general event mapping rules is described in Figure 6. A task t turns from "Activated" to "Executing" after receiving an external $Start$ event. Its entry to "Executing" state triggers a *SubStart* event to its parent goal g, which then turns to "Executing". This state transition may trigger further event propagation to upper goal instances.

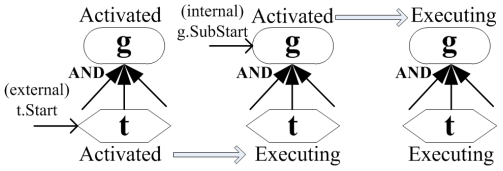

Figure 6. State Propagation Example

Rather than the straightforward $Start$ event propagation as shown in the above example, the interactions among *ShouldDo* goal instance and its subgoal instances involve goal state reasoning. Therefore, an intermediate state **ProgressChecking** and its state action **progressCheck** are introduced in the extended goal state machine. The progress checking is triggered by the *SubRedo, SubSuspend, SubResume,* and *SubAchieve* events from subgoals and *DelegateAchieve* from delegate goals. It reasons about the state of the current goal instance according to the rules listed in Table I. For example, if all the subgoals of an executing and AND-decomposed goal instance are in the **Achieved** state

when it receives a *SubAchieve* event, then its state turns to be **Achieved**. Reasoning with delegate goals works the similar way.

D. Repairing

In the extended goal state machine, a "Repairing" state is introduced to handle those requirements deviations captured by requirements monitoring, with the objective of minimizing possible failures. If the repairing is successful, then the current goal instance returns to elementary states; otherwise, a failure occurs and is propagated to the parent goal instance, and triggers further repairing at higher-level.

Table II
LOCAL REPAIRING POLICIES

Deviation Type	State/Transition	Repairing Policy
Context Condition Violation	Activate (T)	Prevention
Precondition Violation	Start (T)	Compensation
		Prevention
Postcondition Violation	End (T)	Compensation
		Retry
Invariant Violation	Executing (S)	Retry
Commitment Violation	ShouldDo (S)	Compensation
		Retry

1) Local Repairing Policies: When a requirements deviation is captured, local repairing is first conducted to tackle the deviation within the scope of the current goal instance. We summarize the local repairing policies as shown in Table II. The second column of the tabel gives the state or transition at which the deviation occurs. The three suggested local repairing policies are **Prevention**, **Compensation**, and **Retry**. The **Prevention** policy is to prevent a goal to activate or start if the contextual condition or the precondition is not satisfied. **Compensation** represents an active repairing action that takes some additional measures to recover the undesired system status. **Retry** means giving up the goal fulfillment progress so far and trying again to achieve the goal with new commitment.

Prevention policy has a different implication at contextual condition violation and precondition violation. Contextual condition violation means that the intended choice (usually an OR-decomposed subgoal) is not allowed for the current task session, thus the **Prevention** action is to remind the user or the system of the right choice. When a precondition is violated, the **Prevention** action prevents a goal instance from begin execution for the moment, but the goal may start after the required precondition is satisfied. Apart from **Prevention**, the more positive action **Compensation** can be planned and actuated to make the condition satisfied when it is applicable. Another action **Retry** is used for postcondition, invariants, and commitment deviations, kicks the goal back to the *Activated* state and enforce it to re-*start*.

2) Repairing Decision: Combining repairing policies as listed in Table II and state propagation mechanism in Figure 5, a hierarchical system self-repairing (see Figure 3) is achieved with the **doRepairing** action in *Repairing* state. This comprehensive repairing activity takes the requirements deviation as input (indicated by the incoming transitions to *Repairing* state), plans appropriate actions from both local and non-local (goal/agent substitution) policies, and returns the target of the outgoing transition according to the result of the repairing decision and actuation. A detailed description about the repairing decision algorithm can be found in [9].

DoRepairing starts by evaluating the type of monitored deviation, and conducted repairing for specific deviation types respectively. For context deviation monitored, **Prevention** is directed to the actuator at target system side, and *Unactivated* is returned as the target state of the immediate outgoing transition. Postcondition and commitment violation bear the same repairing planning process. **Compensation** is preferred to **Retry** due to possibly less cost. However, finding appropriate compensatory actions for a specific deviation is non-trivial and is beyond the major concern of this paper. A *retrieveSolution (cond1, cond2)* function is to retrieve possible solutions that can transform the context from an initial status $cond1$ to a targeted status $cond2$. This function can be implemented by techniques like knowledge-based component retrieval and composition [10]. If found, the selected plan is pushed to the actuator, and the current goal is thus *Achieved*; otherwise, **Retry** is in use only if its application *match*es particular condition. The current goal is *Failed* if none of above works. Local repairing for precondition and invariants violation is planned similarly using the corresponding policies specified in Table II, only that successful compensation on precondition case would return *Executing* as the following state, and for the sake of space we skip the details within.

According to Figure 5, a goal transits to *Failed* and triggers *SubFail* on its parent or *DelegateFail* on its delegation source, thus *DoRepairing* is called again immediately on the parent/source instance and looks for a non-local repairing policy. For subgoal failure, an AND-decomposed goal propagates failure to its parent again; an OR-decomposed goal would select an unactivated while currently context-compatible goal. The goal either returns to *Activated* as a new substitution is turned on or becomes *Failed* if no alternative is available. The goal being delegated reacts to its delegate failure similarly as what an OR-decomposed goal does to its subgoal failure, except that the substituted goal is searched among agents capable of fulfilling the same goal (g.delegateCandidates).

3) Customizable Repairing Policies: In addition to the common repairing decisions, we also reserve part of our method to be customizable in order to adapt in domain-specific applications. A *match* function is used in the repairing decision algorithm to determine the applicability of **Retry** policy. We argue that no **Retry** action should be performed indefinitely as this might be either useless in some cases or even hazardous for the system in terms of repeated failures. Two factors we currently defined for *match* function are **use limit** and **avoidance goal state**

pattern. A *use limit* specifies the upper bound of the policy occurrences. For example, *match* function returns false if a task-to-fail's retry time has reached a specified limit. By *avoidance goal state pattern*, **Retry** is ruled out if monitored goal state trace history contains that of specified state pattern, in terms of regular expression-like representation. For example, avoidance pattern of *(executing activated)+* is specified to "Transport by Truck", suggesting employee's unskillfulness. If "Transport by Truck" failed eventually, the *match* function in the repairing decision algorithm would return false and skip *Retry* on the task, and in futher steps an **Agent Substitution** might occur. The *match* function performs conjunction on above two factors checking.

IV. Experimental Study

To evaluate the effectiveness of our approach, we have implemented a customizable framework in Java for an experiment through the simulation of a typical socio-technical system. Implementing goal state machines and managing their interactions through repairing decisions, the framework provides a customizable interface for defining goal models and repairing policies, and integrating with application-specific monitors and actuators. The socio-technical system is simulated using AnyLogic [8], which has Java APIs to measure the effectiveness through several key performance indicators.

A. Settings for Experimental Study

Consider a food preparing system in a socio-technical environment, as shown in Figure 7, its goal model involves five agents: a software agent **Food Ordering System**, and four human agents: **Customer**, **Order Dispatcher**, **Chef**, and **Delivery Man**. A customer gets food prepared by either ordering from a restaurant or by cooking instant food with a microwave. The restaurant runs a food ordering service for customers to place orders online. If a customer orders food via the food ordering system, an order dispatcher will be delegated to handle the order by arranging a chef to cook the food and by assigning a delivery man to deliver the food after confirmation.

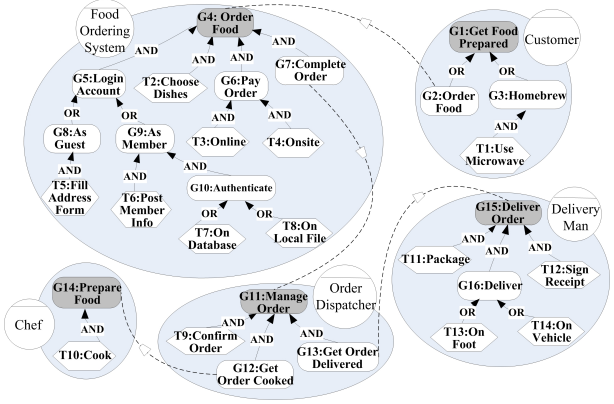

Figure 7. Food Preparing Goal Model

Some of the constraints specified for the goals/tasks in Figure 7 are listed in Table III, following expected repairing policies that are compatible in this experiment if violation occurs. Take **Customer** as example, a customer who intends to have food heated with microwave may carelessly input unmatched mode and parameters, and only to get cold food as a result. The error may even damage the device and expose its user to safety hazard. In this case, an alarm to notify the user about the erroneous configuration and guide him/her to retreat would be helpful. Other requirements denials may come from food ordering systems and restaurant employees. The dish display web page occasionally goes down and prevents a customer from proceeding with an order; An internal database exception may block a customer's login info from being authenticated, when an alternative local backup file reading would suffice; A slow delivery man may spend much time on finding his way to customer's home and break his commitment to deliver order within 30 minutes, when another delivery man can be assigned to take over the task. If a constraint violation turns out to be unrepairable during food ordering service, then the cascading failure would deny customer's goal of getting prepared food. We assume that, if an order is successfully delivered but is overdue (exceeding customer's expected time), the customer will complain but his goal of food preparation is still considered to be achieved.

Table III
Food Preparing Goal Constraints and Repairing Policies

Goal	Constraint Type	Specification	Repairing
T1	precondition	¬wellConfigured	Prevent
T5	postcondition	info.correct	Compensate
T7	invariants	¬database.exception	GoalSUB (G10)
T2	invariants	¬displayPage.corrupt	Retry
T3	commitCondition	¬payment.expired	GoalSUB (G6)
G13	commitCondition	responseTime<30m	AgentSUB
T11	invariants	¬pack.inappropriate	Retry

B. Simulation Scenario

A simulation of the food preparing system was created using AnyLogic [8], and used as the subject for evaluating our stateful monitoring and repairing approach. AnyLogic is an all-around logic modeling and simulation software that allows us to build the desired agent-based system model in combination with discrete triggering events and message communications. Using the AnyLogic modeling tool and its terminology, we created five *active object classes* as the *agents* corresponding to the five agent types in the food preparing system, and defined their connections within the *main* environment.

Figure 8 illustrates the simulation scenario in AnyLogic. A number of customers choose to order food from a restaurant or cook instant food by themselves at a certain probability. A food ordering system receives orders from customers and put them into a queue for order dispatchers to handle. An order dispatcher checks each received order and passes it to chefs. After a chef gets an order ready,

the dispatcher assigns a delivery man to package and ship the ordered food to the customer. Once built, the scenario can be simulated with different parameters, e.g., in one simulation, 150 customer instances are created with one restaurant, which has one order dispatcher, 10 chefs and 10 delivery men.

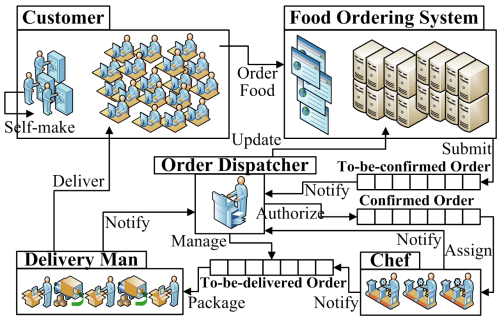

Figure 8. Overview of the Simulation Scenario

In AnyLogic modeling, each agent possesses a statechart dictating its behaviors and reactions to incoming messages at runtime. Inter-agent interaction is realized by message exchanging between the connected ones. Agents' choices on alternative state transitions are simulated by specifying a triggering rate for related transitions. For example, the transition rates of customer choosing to order food from restaurant is specified by the parameter *makingOrderRate* (see Table IV). To simulate occasional negligence or adverse conditions that may cause potential failures, we inject some faulty state transitions and stochastic delays into the behavior models of related agents. For example, a faulty state transition for a customer to improperly launch a microwave is introduced with a probability specified by the parameter *misuseRate*.

Our monitoring and repairing framework is integrated with the simulation system using AnyLogic, which exposes code injection entries and event handling interfaces for active object classes in Java. Thus interested agent behaviors can be monitored by goal state machine via external event mapping, and repairing actions are realized as AnyLogic agent messages by implementing the *RepairingPolicy* interface of our framework and are registered into the framework as callback handlers along with customized repairing policies.

C. Results and Evaluation

We ran the experiment on a computer with Intel Core2 Duo P8700 CPU and 2G memory. The configurations of the used probability parameters are listed in Table IV. The simulation scenario was executed 20 times with and without self-repairing respectively. In each execution, a customer tries to get food prepared once, and the time of customer starting preparing food is set to be linear in the time line. For evaluation, the experiment produced 3000 **Customer** sessions, that is, 20 times of execution times 150 instances in each execution, with or without self-repairing respectively.

The following measurements of key performance indicators were collected for evaluation: **a)** the number of customers who have their food successfully prepared, indicating goal satisfaction from the customer's perspective; **b)** the number of food ordering requests sent to the restaurant; **c)** the number of food orders successfully delivered, indicating goal satisfaction from the restaurant's perspective; **d)** the number of damaged microwaves, indicating occurrences of potentially critical safety hazard; and **e)** the number of customer complaints, indicating the quality of service provided by the restaurant.

Table IV
SIMULATION PARAMETERS CONFIGURATION

Agent	Parameter	Value
Customer	makingOrderRate	0.8
Customer	misuseRate	0.2
Customer	microwaveDamageRate	0.1
FoodOrderSystem	memberLoginRate	0.9
FoodOrderSystem	incorrectInfoRate	0.1
FoodOrderSystem	corruptPageRate	0.1
FoodOrderSystem	onlinePaymentRate	0.8
FoodOrderSystem	databaseExceptionRate	0.05
FoodOrderSystem	paymentExpireRate	0.05
Order Dispatcher	responseDelayRate	0.2
Delivery Man	inappropriatePackageRate	0.1
Delivery Man	damagedPackageRate	0.05

The experiment results are shown in Figure 9, where the letter under each bar corresponding to the measurements mentioned above and the results with and without asterisks indicate the measurements with and without self-repairing respectively. It can be seen that the number of customers having their food prepared (a) is improved 22.27% after introducing self-repairing; the success rate of restaurant's order delivery (c/b) rises from 77.58% to 95.51%; the hazards brought by damaged microwaves (d) are entirely eliminated; and 63% customer complaints (e) are ruled out. The complaints are not brought down as much as failed orders because some ordering sessions are still delayed and cause delivery overdue, although they were repaired by **Retry** and **Agent Substitute** policies.

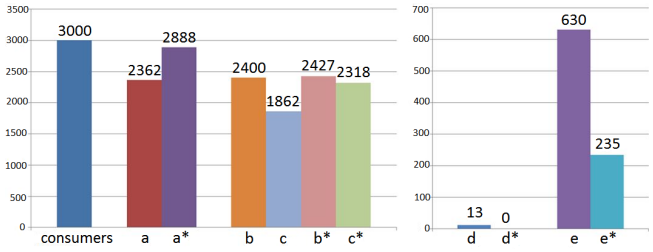

Figure 9. Improved results: $a* > a, (c*/b*) > (c/b), d* < d, e* < e$

We took a segment of reinterpreted goal state transition trace (see Table V) from our experiment to illustrate how our stateful monitoring and repairing approach works behind the simulation scenario. In Table V, the first column refers to the goals/tasks in Figure 7 and the other columns corresponding to different steps in the trace; the second row shows goal events issued in the corresponding steps; the third to eighth rows show the states of related goals in each step (G15' and T11' are goal instances for another delivery man). The

trace starts with T11's external *start* event, indicating that a delivery man begins to pack an order. The *executing* state is propagated in turn to G15, G13, G11. Then a postcondition violation is detected with T11's *end* event, making T11 failed since no local repairing policies are defined for it. This failure is then propagated to G15, triggering an internal *delegateFail* event for G13. On behalf of the order dispatcher, our repairing module decides to enforce **Agent Substitute** by assigning another delivery man from available delegate candidates list for the order, resulting in creation of a set of new goal instances (e.g. G15' and T11') on the delivery man. The new delivery task starts later on.

Table V
SEGMENT FROM RUNNING SIMULATION

Step	76722	78250	78264	78282	78291
Event	T11.start	T11.end	G13.delegateFail	G15'.acti	T11'.start
G11	exec	exec	exec	acti	exec
G13	exec	exec	rep	acti	exec
G15	exec	failed	failed	failed	failed
T11	exec	failed	failed	failed	failed
G15'	-	-	-	acti	exec
T11'	-	-	-	acti	exec

D. Discussion

In the experimental study, we conducted a series of application-specific customization and extension to the monitoring and repairing framework. Apart from defining a goal model for the 5 agents, we defined 72 external goal events and the corresponding event mappings, 14 goal constraints, and 9 actuators. These monitors and actuators were implemented by 85 Java modules with totally 1234 lines of code. Using a tool to semi-automatically generate part of the code based on the parameterized code templates, it is shown that the effort required for the customization and extension is acceptable compared with the benefits achieved.

It is important to note that any repairing action is possible to fail again, though the reasoning of repairing failure rate is out of this paper's scope. Interestingly, we did find that approximately 15% of repairing actions failed to come into effect during our experiments. Since we left all the agent scheduling work to AnyLogic and after-violation transitions within an agent behavior statechart has limited timeout trigger specified, it is likely that the repairing action pushed by goal state machine just misses the timeout window, so that the agent continues its deviated behavior. This fact compensates for the missing explicit repairing failure rate expression and makes sense because repairing actions should not compromise real-time feature in a real-time system and real-life transactions always have suspension limits.

During the application-specific repairing actions enacted in our experiment, a possibility to refine and regroup repairing policies also emerged. Taking the policy *Prevent* in Table III as an example, once a potential microwave malfunction caused by ill parameters is monitored, not only was the ongoing process blocked but also user got a warning from it. Therefore two atomic policies *Blocking* and *Warning* can be composed into one *Prevent*. Finding such atomic opportunity to forge new policies could further simplify the logic.

V. RELATED WORK

In a pioneer work, Feather et al. [11] proposed an architecture and a development process for runtime requirements monitoring and repairing based on goal-oriented requirements engineering. Combining requirements-time goal reasoning, event-based runtime monitoring, and self-adaptation tactics, their work mainly concentrates on requirements deviations caused by unsatisfied environment assumptions. A more recent approach proposed by Wang et al. [3], [4] diagnoses and repairs errors detected at runtime. The repairing is implemented by selecting a best system configuration from alternative configurations using SAT-solver based goal reasoning, and the selected configuration contributes most positively to the systems non-functional requirements. Dalpiaz et al. [6], [12] proposed a requirements-driven self-reconfiguration architecture, using goal models-based reasoning to monitor for and diagnose failures, and axiomatized a support relation in a service-oriented environment to reason whether adopting a role is compatible with given goals, application scenarios, and various commitments. Unlike the logic based failure reasoning in these methods, our work uses the state history of relevant goals to simulate for more precise repairing decisions. Instead of considering non-functional requirements in selecting repairing reconfigurations through goal substitutions, we also simulate probablistically the interactions between the modelled agents.

Unlike previous work [4], [6], our approach supports decentralized requirements monitoring and self-repairing. The lifecycle management of goals and support of self-repairing decision are for decentralized agents, thus does not require a global view of the target system. In addition, our approach supports fine-grained requirements monitoring and repairing at individual goals level, providing a hierarchical repairing process for both local and global policies. These advantages are achieved by extending state machine for individual goals, and by propagating failures among goals and agents through event-based state machine interactions.

Apart from goal-based approaches, Salifu et al [13] use problem frames to specify monitoring and adapation requirements and introduce state machines to represent the composition of adaptation switches. A simulation of such systems was explored [14] using Websphere Business Modeler. One difference here is that we offer goal-based patterns to specify state machines to embed the reasoning logic in simulation. Complement to the proposal in this work, the analysis of state machines of contextual domains in the physical world still need to be elicited based on domain expertise.

Awareness requirements approaches [15], [16] concern monitoring as meta-requirements about success/failure of functional requirements and quality-of-service requirements, in addition to uncertainty of domain assumptions at runtime.

Several attempts have been made to model the awareness requirements using fuzzy logic [17], [18], feedback loops [19], [20]. This work demonstrates the benefits of simulating such an adaptive system by modeling goal state machines.

Design-by-contract methodology for implementation [21] supports runtime checks. Traditional runtime verification methods like the MOP framework proposed by Chen et al. [22] monitor generic constraints such as preconditions, postconditions and invariants at runtime. Although MOP provides violation handlers for developing their own repairing implementations, it does not involve requirements models in monitoring and does not provide builtin self-repairing mechanisms. Model-based monitoring methods such as Schneider et al. [23] require strict behavior models specific to applications. In contrast, the state machines are used in our approach to specify and manage the runtime lifecycles of requirements goals.

VI. Conclusions and Future Work

We have proposed a distributed, stateful, goal-based monitoring and self-repairing framework for socio-technical systems. The main contributions of our proposal are as follows. First, by defining state transitions of goal instances and their interactions at runtime, the approach provides a comprehensive requirements monitoring mechanism that combines both traditional constraints checking (i.e. precondition, postcondition, and invariant) and goal-oriented requirements reasoning. Second, our monitoring framework detects deviations from state histories of relevant goal instances, and supports precise and fine-grained self-repairing decision making. Third, the isomorphic nature of goal state machines and event-driven interactions facilitate the implementation of decentralized requirements monitoring and repairing in socio-technical systems. In the future, we consider integrating this work with runtime modeling of the behaviors of physical world domains and conducting more experiments with real socio-technical systems, e.g., applications in the field of pervasive computing.

Acknowledgment

This work has been supported in part by Chinese National grants of Natural Science Foundation No. 90818009, High Technology Development 863 Program No.2012AA011202 and by both the European Research Council advanced grants 267856 and 291652.

References

[1] A. R. Dingwall-Smith, "Run-time monitoring of goal-oriented requirements specifications", Ph.D. dissertation, University of London, 2006.

[2] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani, "Reasoning with goal models", in *ER*, 2002.

[3] Y. Wang, S. A. Mcllraith, Y. Yu, and J. Mylopoulos, "An automated approach to monitoring and diagnosing requirements", in *ASE*, 2007.

[4] Y. Wang and J. Mylopoulos, "Self-repair through reconfiguration: A requirements engineering approach", in *ASE*. IEEE Computer Society, 2009, pp. 257–268.

[5] M. J. Khan, M. M. Awais, and S. Shamail, "Enabling self-configuration in autonomic systems using case-based reasoning with improved efficiency", in *ICAAS*, 2008.

[6] F. Dalpiaz, P. Giorgini, and J. Mylopoulos, "An architecture for requirements-driven self-reconfiguration", in *CAiSE*, 2009.

[7] S. W. Cheng, A. C. Huang, D. Garlan, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure", in *ICAC*, 2004.

[8] X. Technologies, "Anylogic", http://www.xjtek.com/anylogic/.

[9] L. Fu, X. Peng, Y. Yu, J. Mylopoulos, and W. Zhao, "Stateful requirements monitoring for self-repairing socio-technical systems", Software Engineering Lab, Fudan University (FDSE-TR201201), Tech. Rep., 2012.

[10] L. Chen, N. R. Shadbolt, C. Goble, F. Tao, S. J. Cox, C. Puleston, and P. R. Smart, "Towards a knowledge-based approach to semantic service composition", in *ISWC*, 2003.

[11] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, "Reconciling system requirements and runtime behavior", in *IWSSD*, 1998.

[12] A. K. Chopra, F. Dalpiaz, P. Giorgini, and J. Mylopoulos, "Modeling and reasoning about service-oriented applications via goals and commitments", in *CAiSE*, 2010.

[13] M. Salifu, Y. Yu, and B. Nuseibeh, "Specifying monitoring and switching problems in context", in *RE*, 2007.

[14] M. Salifu, Y. Yu, A. K. Bandara, and B. Nuseibeh, "Analysing the requirements for monitoring and switching: A problem-oriented approach", *J. Syst. Software*, to appear, 2012.

[15] V. E. S. Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos, "Awareness requirements for adaptive systems", in *SEAMS*, 2011.

[16] K. Welsh, P. Sawyer, and N. Bencomo, "Towards requirements aware systems: Run-time resolution of design-time assumptions", in *ASE*, 2011.

[17] J. Whittle, P. Sawyer, N. Bencomo, B. Cheng, and J.-M. Bruel, "Relax: Incorporating uncertainty into the specification of self-adaptive systems", in *RE*, 2009.

[18] L. Baresi, L. Pasquale, and P. Spoletini, "Fuzzy goals for requirements-driven adaptation", in *RE*, 2010.

[19] X. Peng, B. Chen, Y. Yu, and W. Zhao, "Self-Tuning of Software Systems through Dynamic Quality Tradeoff and Value-based Feedback Control Loop", *J. Syst. Software*, to appear, 2012.

[20] B. Chen, X. Peng, Y. Yu, and W. Zhao, "Are your sites down? requirements-driven self-tuning for the survivability of web systems", in *RE*, 2011.

[21] Y. L. Traon, B. Baudry, and J.-M. *Jézéquel*, "Design by contract to improve software vigilance", *IEEE Trans. Softw. Eng.*, vol. 32, pp. 571–586, 2006.

[22] F. Chen and G. Rosu, "Mop: an efficient and generic runtime verification framework", in *OOPSLA*, 2007.

[23] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial", *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.