UNIVERSITY
OF TRENTO

DIPARTIMENTO DI INGEGNERIA E SCIENZA DELL'INFORMAZIONE

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.disi.unitn.it

An Architecture for Requirements-driven Self-reconfiguration

Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos

February 2009

Technical Report # DISI-09-010

An Architecture for Requirements-driven Self-reconfiguration

Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos

University of Trento - DISI, 38100, Povo, Trento, Italy.
{fabiano.dalpiaz, paolo.giorgini, jm}@disi.unitn.it

Abstract. Self-reconfiguration is the capability of a system to autonomously switch from one configuration to a better one in response to failure or context change. There is growing demand for software systems able to self-reconfigure, and specifically systems that can fulfill their requirements in dynamic environments. We propose a conceptual architecture that provides systems with self-reconfiguration capabilities, enacting a model-based adaptation process based on requirements models. We describe the logical view on our architecture for self-reconfiguration, then we detail the main mechanisms to monitor for and diagnose failures. We present a case study where a self-reconfiguring system assists a patient perform daily tasks, such as getting breakfast, within her home. The challenge for the system is to fulfill its mission regardless of the context, also to compensate for failures caused by patient inaction or other omissions in the environment of the system.

1 Introduction

There is growing demand for software systems that can fulfill their requirements in very different operational environments and are able to cope with change and evolution. This calls for a novel paradigm for software design where monitoring, diagnosis and compensation functions are integral components of system architecture. These functions can be exploited at runtime to monitor for and diagnose failure or under-performance, also to compensate through re-configuration to an alternative behavior that can better cope with the situation on-hand. Self-reconfiguration then is an essential functionality for software systems of the future in that it enables them to evolve and adapt to open, dynamic environments so that they can continue to fulfill their intended purpose.

Traditionally, self-reconfiguration mechanisms are embedded in applications and their analysis and reuse are hard. An alternative approach is externalized adaptation [1], where system models are used at runtime by an external component to detect and address problems in the system. This approach – also known as model-based adaptation – consists of monitoring the running software, analyzing the gathered data against the system models, selecting and applying repair strategies in response to violations.

We analyze here the usage of a special type of system models: requirements models. Deviations of system behavior from requirements specifications have been discussed in [2], where the authors suggest an architecture (and a development process) to reconcile requirements with system behavior. Reconciliation is enacted by anticipating deviations at specification time and solving unpredicted circumstances at runtime. The underlying model is based on the goal-driven requirements engineering approach KAOS [3].

In this paper, we propose a conceptual architecture that, on the basis of requirements models, adds self-reconfiguration capabilities to a system. The architecture is structured as a set of interacting components connected through a Monitor-Diagnose-Compensate (MDC) cycle. Its main application area is systems composed of several interacting systems, such as Socio-Technical Systems [4] (STSs) and Ambient Intelligence (AmI) scenarios. We have chosen to use Tropos [5] goal models as a basis for expressing requirements, since they suit well for modeling social dependencies between stakeholders. We enrich Tropos models adding activation events to trigger goals, context-dependent goal decompositions, fine-grained modeling of tasks by means of timed activity diagrams, time limits within which the system should commit to carry out goals, and domain assumptions that need to be monitored regardless of current goals.

We adopt the BDI paradigm [6] to define how the system is expected to reason and act. The system is running correctly if its behavior is compliant with the BDI model: when a goal is activated, the system commits to it by selecting a plan to achieve it. The architecture we propose monitors system execution and looks for alternatives when detecting no progress or inconsistent behaviour.

The closest approach to our work is Wang et al. [7], which proposes a goal-oriented approach for self-reconfiguration. Our architecture differs from hers in the details of the model we use to monitor for failures and violations. These details allow us to support a wider class of failures and changes, also to compensate for them.

This paper is structured as follows: Section 2 presents the baseline of our approach, Section 3 describes our proposed architecture for self-reconfiguration, whereas Section 4 explains how to use it. Section 5 details the main monitoring and diagnosis mechanisms the architecture components use, while Section 6 shows how the architecture can be applied to a case study concerning smart-homes. Section 7 presents related work and compares our approach to it. Finally, Section 8 discusses the approach and draws conclusions.

2 Baseline: Requirements Models

A requirements-driven architecture for model-based self-reconfiguration needs a set of models to support full modeling of requirements. A well established framework in Requirements Engineering (RE) is goal-oriented modeling [3], where software requirements are modelled as goals the system should achieve (with assistance from external agents). Among existing frameworks for requirements models, we have chosen Tropos [5], for it allows to describe systems made up of several socially interacting actors depending on each other for the fulfillment of their own goals. Recently, Jureta et al. [8] have revisited the so-called "requirements problem" – what it means to successfully complete RE – showing the need for requirements modeling frameworks richer than existing ones. The core ontology they propose is based on the concepts of goal, softgoal, quality constraint, plan, and domain assumption. Direct consequence of this result is that goal models alone are insufficient to completely express system requirements, and in our framework we support some of the suggested ingredients to express requirements.

We adopt an enriched version of Tropos, which contains additional information to make it suitable for runtime usage: (i) activation events define when goals are triggered; (ii) commitment conditions express a time limit within which an agent should commit to a goal; (iii) contexts express when certain alternatives are applicable (like in Ali et al. [9]); (iv) preconditions for taks (similarly to Wang et al. [7]). In Fig. 1, two agents (*Patient* and *Supermarket*) interact by means of a dependency for goal *Provide Grocery*. The top-level goal of patient – *Have lunch* – is activated when it's 12AM, and the patient should commit to its achievement within one hour since activation. Two alternatives are available to achieve the goal, that is *Prepare lunch* and *Get lunch prepared*. In this scenario, the former option is applicable only in context $c1$, that is when patient is autonomous, whereas the latter option is applicable when the patient is not autonomous ($c2$). Goal *Prepare lunch* is and-decomposed to sub-goals *Get needed ingredients* and *Cook lunch*. The former goal is a leaf-level one, and there are two tasks that are alternative means to achieve it (means-end): *Take ingredients from cupboard* and *Order food by phone*. The latter task starts the dependency for goal *Provide grocery* on agent supermarket.

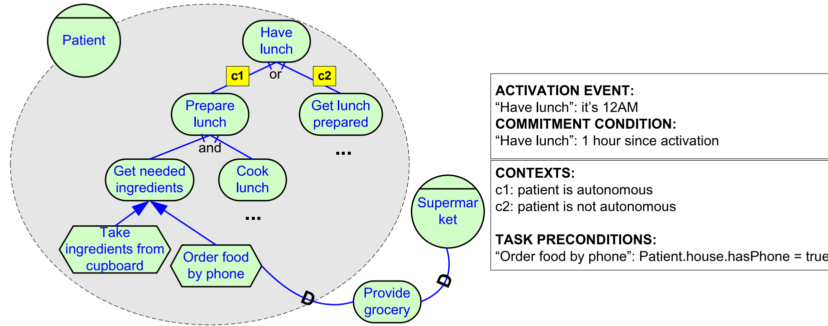

Fig. 1. Enriched Tropos goal model used by our architecture.

A shared language to express information about domain is clearly needed. This language is used to formally express contexts, preconditions, domain assumptions, and any relation between domain and requirements. We exploit aan object diagram (as in [9]), where context entities are objects, their properties are attributes, and relations between entities are association links. For instance, the precondition for task *Order food by phone* – `Patient.house.hasPhone = true` – can be expressed in an object model with classes $Patient$ and $House$, where Patient is linked to House by an aggregation called $house$, and House has a boolean attribute $hasPhone$. Domain assumptions are conditions that should always hold, and can be expressed as rules. For example, a domain assumption for our small example is that each patient has exactly one house; this assumption should hold regardless of current contexts and goals. Finally, we use a fine grained definition of tasks, in which each task is a workflow of monitorable activities to be carried out within time constraints. The completion of each activity is associated to the detection of an associated event, which is expressed over the context model. We provide further details about this formalism in Section 5.

3 System Architecture

In this section we propose our conceptual architecture for structuring systems able to self-reconfigure. We present the architecture logical view in Fig.2, exploiting an UML 2.0 component diagram to show the components and the connections among them. Component diagrams depict not only the structure of a system, but also the data flow between components (through provided and required interfaces).

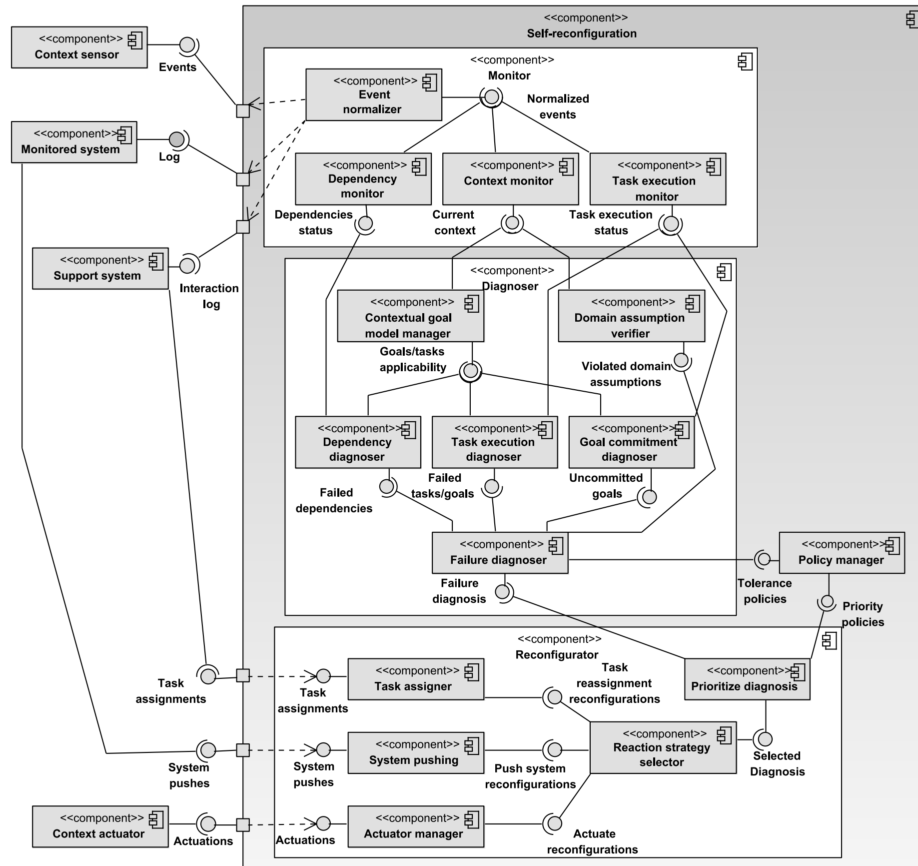

Fig. 2. Logical view on the proposed architecture for self-reconfiguration.

3.1 External components

Our architecture supports systems characterized by decentralized and non-monolithic structure, such as Socio-Technical Systems (STSs) and Ambient Intelligence (AmI) scenarios, which require quick and effective reconfiguration in response to context change

or failure. A set of external components interacts with the self-reconfiguration component, providing inputs and enacting reconfigurations.

The component *Context sensor* represents any system providing up-to-date information about the context where the system is running. In AmI settings, sensors are spread throughout the environment and collect data such as temperature, light level, noise, presence. Also desktop applications have several context sensors that provide useful values such as free memory, CPU utilization, mainboard temperature, and list of active processes. The component context sensor provides changes in the context through the interface *Events*. Possible events are changes in the light level, detection of humans in front of the door, identification of loud noise in the bathroom.

Monitored system is the system the self-reconfiguration component assists, that is the stakeholder whose requirements are monitored to diagnose and compensate failures. This system need not necessarily be software or hardware, but can be – and often is – a human or an organization. Examples of monitored systems are anti-virus software, patients living in smart-homes, firemen in crisis management settings. This component provides all available information concerning the current status of the system through the interface *Log*, and requires from the interface *System pushes* advice on what should be done (which *goals)* and how it should act (which *tasks*). A patient can be reminded to take her medicine by sending an SMS to her mobile phone (system pushes interface).

Support system represents any system connected to the monitored system by requirements level links by goal, task, or resource dependencies from the monitored system. For example, anti-virus software may depend on update sites for the resource "updated virus definition file", while patients may depend on social workers for the goal "prepare breakfast". The provided interface *Interaction log* contains information about the status of dependencies with the monitored system; the required interface *Task assignments* provides the tasks or goals for which the monitored system depends on support systems. If the patient should prepare breakfast but did not commit to it, the self-reconfiguration component can order breakfast from a catering service (the support system), enacting a dependency from the patient to the catering service for goal "prepare breakfast".

Context actuator identifies any actuator in the environment which can receive commands to act on the context. Examples of actuators in AmI scenarios are sirens, door openers, automated windows, and remote light switches. The component gets from the required interface *Actuations* the commands to enact.

3.2 Self-reconfiguration component

The self-reconfiguration capabilities of our architecture are provided by the component *self-reconfiguration*. We identified three major sub-components in the reconfiguration process, each enacting a phase in a Monitor-Diagnose-Compensate cycle. *Monitor* is in charge of collecting, filtering, and normalizing events and logs; *Diagnoser* identifies failures and discovers root causes; *Reconfigurator* selects, plans and deploys compensation actions in response to failures.

The monitoring phase starts with the component *Event normalizer*, which depends on the interfaces events, log, and interaction log through appropriate ports (graphically represented as small squares on the sides of components). This component gathers the

current status of the monitored system, of the context, and of the interaction with support systems, then it normalizes the collected data to a specific format. The normalization process requires the definition of a translation schema for each source data format. This topic is an explored one: for instance, XSLT [10] is a W3C standard to define transformations between XML schemas. The event normalizer provides the translated data through the interface *Normalized events*. This interface is required by three different components, each handling a specific type of events and combining new events with the previous status of the system. *Dependency monitor* computes the status of existing dependencies and exposes it through the provided *Dependencies status* interface. *Context sensor* is in charge of updating the interface *Current context*, processing the normalized events related to changes in the context. For instance, if the house door is closed ($door.status = closed$) and we just received an event such as $open(door, time_i)$, the status of the door will change to open ($door.status = open$). The component *Task execution monitor* handles events concerning the execution of tasks and provides the interface *Task execution status*. For example, if the patient is executing the task "Open door" and event $pressed(patient, button, time_j)$ is received, the status of task "Open door" will turn to success.

The diagnosis phase – responsibility of the component *Diagnoser* – is essentially a verification of the current status against requirements models. Models specify what should happen and hold: which goals should / can / cannot be achieved, which tasks can / cannot be executed, the domain assumptions that should not be violated. The richer the requirements model is, the more accurate the diagnosis will be. In contrast, the granularity of detected events is bounded by technological and feasibility aspects. Detecting if a patient is sitting on a sofa is reasonably realizable (e.g., using pressure sensors), while detecting if she is handling a knife the wrong way is far more complex.

Contextual goal model manager requires the interface current context, analyzes the goal model to identify goals and tasks that should / can / cannot be achieved, and provides this output through the interface *Goals / Tasks applicability*. The component *Domain assumption verifier* requires the interface current context and compares it to the list of domain assumptions. Its goal is to identify violations, which are then exposed through the provided interface *Violated domain assumptions*.

Dependency diagnoser requires the interfaces dependencies status and goals / tasks applicability, and computes failed dependencies. Dependencies fail not only if the dependee cannot achieve the goal or perform the task (e.g., the nurse cannot support the patient because she's busy with another patient), but also when changes in the context modify goal applicability and the dependency is not possible anymore (e.g., the patient exits her house and thus cannot depend on a catering service anymore). *Task execution diagnoser* requires the interfaces goals / tasks applicability and task execution status, and provides the interface *Failed tasks / goals*. The objective of this component is to verify whether the current execution status of tasks is compliant with task applicability. For example, if the patient is preparing breakfast but already had breakfast, something is going wrong and this failure should be diagnosed. *Goal commitment diagnoser* is in charge of detecting those goals that should be achieved but for whose fulfillment no action has been taken. In our framework, each top-level goal has a commitment time, a timeout within which a commitment to achieve the goal should be taken (i.e., an ad-

equate task should begin). For example, the patient should have breakfast within two hours since waking up. This component requires the interfaces goals / tasks applicability and task execution status, and provides the interface *Uncommitted goals*.

The component *Failure diagnoser* requires the interfaces containing the identified failures (failed dependencies, failed tasks / goals, uncommitted goals) and the interface *Tolerance policies* provided by component *Policy manager*. The policy manager – handling policies set by system administrators – specifies when failures do not lead to reconfiguration actions. For example, lack of commitment for washing dishes can be tolerated if the patient's vital signs are good (she may wash dishes after next meal). The provided interface *Failure diagnosis* contains the diagnoses to be compensated.

The reconfiguration phase – carried out by component *Reconfigurator* – should define compensation / reconfiguration strategies in response to any kind of failure. Its effectiveness depends on several factors: number of tasks that can be automated, available compensation strategies, extent to which the monitor system accepts suggestions and reminders. In our architecture we propose general mechanisms, but the actual success of compensation strategies is scenario-dependent and difficult to assess. Suppose a patient suddenly feels bad: if she lives in a smart-home provided with a door opener, the door can be automatically opened to the rescue team; otherwise, the rescue team should wait for somebody to bring the door keys.

The component *Prioritize diagnosis* requires the interfaces failure diagnosis and priority policies; it selects a subset of failures according to their priority level and provides them through the interface *Selected Diagnosis*. Common criteria to define priority are failure severity, urgency of taking a countermeasure, time passed since failure diagnosis. Selected diagnoses are then taken as input by the component *Reaction strategy selector*, which is in charge of choosing a reaction to compensate the failure. This component acts as a planner: given a failure, it looks for appropriate reconfigurations, and selects one of them. Three different types of reconfigurations are supported by our architecture, each manifested in a specific interface. The interface *Task reassignment reconfigurations* contains reconfigurations that involve the automated enactment dependencies on support systems. For example, if the patient didn't have breakfast and the commitment time for the goal is expired, the system could automatically call the catering service. The interface *Push system reconfigurations* includes strategies that push the monitored system to achieve its goals (reminding goals or suggesting tasks). A push strategy for the patient that hasn't had breakfast so far is sending an SMS to her mobile phone. The interface *Actuate reconfigurations* consists of compensations that will be enacted by context actuators. For instance, if the patient feels bad, the door can be automatically opened by activating the door opener.

Three components use the interfaces provided by reaction strategy selector: *Task assigner*, *System pushing*, and *Actuator manager*. Their role is to enact the reconfigurations that have been selected, and each component provides a specific interface.

4 Creating the architecture for an existing system

In this section we describe how the architecture can be used in practice to add self-reconfiguration capabilities to a distributed socio-technical system. The required input

is a set of interacting sub-systems – sensors and effectors – that compose the distributed system. The following steps should be carried out: (i) define context model (ii) define requirements models; (iii) establish traceability links for monitoring; (iv) select tolerance policies for diagnosis; and (v) choose reconfiguration and compensation mechanisms.

Steps (i) and (ii) output the models we presented in Section 2, that is the context model, the Tropos goal model, timed activity diagrams for tasks, and domain assumptions. Step (iii) defines what to monitor for at runtime, by connecting requirements to code. Traceability is ensured by associating events – produced by sensors – to *activities* that are part of a task, to *task preconditions*, to *contexts*, and to *activation conditions* for top-level goals. Events should also be normalized according to the context model defined in step (i).

Step (iv) is carried out to specify tolerance policies for failures. Indeed, some failures have to be addressed through reconfiguration, whereas some others can be tolerated. In step (v) the reaction mechanisms enacting self-reconfiguration are defined. Two sub-steps should be carried out: (i) definition of a *compensation plan* to revert the effects of the failed strategies, and (ii) identification of a *reconfiguration strategy* to retry goal achievement. Both steps exploit the actuation capabilities of the distributed system, i.e. reconfigurations consist of giving commands to effectors (execute a task, enact a dependency, issue a reminder).

5 Monitoring and diagnosis mechanisms

We detail now monitoring and diagnosis mechanisms included in our architecture. Efficient and sound algorithms need to be defined for successfully diagnosing problems in the running system. Failures are identified by comparing *monitored behavior* of the system to the *expected and allowed behaviors*. Failures occur when (a) the monitored behavior is not allowed or (b) an expected behavior has not occurred.

Table 1 defines expected and allowed goals and tasks. We use first-order logic rules for clarity, but our prototype implementation is based on disjunctive Datalog [11]. We suppose that each goal instance differs from other instances of the same goal class for actual parameters; for example, a patient's goal *Have breakfast* can be repeated every day, but with different values for the parameter *day*.

Rule (i) defines when a top-level goal should be achieved. This happens if G is a goal with parameter set P, the goal instance has not been achieved so far, the activation event has occurred before the current time, and G is a top-level goal (there is no other goal G_p and/or-decomposed into G). Rule (ii) is a general axiom saying that whenever a goal instance should be achieved, it is also allowed. Rules (iii) and (iv) define when tasks and decomposed goals are allowed, respectively. A goal instance G with parameter set P can be achieved if it has not been done so far and exists an achievable goal G_p with parameters P_p that is decomposed into G, the context condition on the decomposition is true, and the actual parameters of G are compatible with the actual parameters of G_p. A similar condition holds for means-end tasks, with two main differences: tasks are also characterized by a precondition – which should hold to make the task executable – and are connected to goals through means-end (rather than by and/or decomposition).

Expected and allowed goals and tasks identified by rules (i-iv) is used by Algorithm 1 to diagnose goals and tasks failures, comparing the monitored behavior to expected and allowed behaviors. The parameters of COMPUTEFAILURES are the monitored system, the examined goal instance and the set of failures (initially empty). The algorithm should be invoked for each top-level goal of the monitored system, and explores the goal tree recursively. In the following algorithms we represent goals as objects instead of using predicates as in Table 1; all parameters are passed by reference.

$$\frac{\begin{array}{l} goal(G) \wedge goal_parameters(G,P) \wedge \neg done(G,P) \wedge activation_evt(G,P,T) \\ \wedge\, T \leq current_time \wedge \nexists\, G_p \; s.t. \\ \quad goal(G_p) \wedge decomposed(G_p, G) \end{array}}{\textbf{should_do(G,P)}} \quad \text{(i)}$$

$$\frac{should_do(G,P)}{\textbf{can_do(G,P)}} \quad \text{(ii)}$$

$$\frac{\begin{array}{l} goal(G) \wedge goal_parameters(G,P) \wedge \neg done(G,P) \\ \wedge\, \exists\, G_p \; s.t. \\ \quad goal(G_p) \wedge goal_parameters(G_p, P_p) \wedge decomposed(G_p, G, Dec) \\ \quad \wedge\, can_do(G_p, P_p) \wedge context_cond(Dec) \\ \quad \wedge\, \forall\, p \in P \; s.t. \; (\exists\, p_p \in P_p \; s.t. \; name(p,n) \wedge name(p_p,n)), \\ \qquad value(p,v) \wedge value(p_p,v) \end{array}}{\textbf{can_do(G,P)}} \quad \text{(iii)}$$

$$\frac{\begin{array}{l} task(T) \wedge task_parameters(T,P) \wedge pre_cond(T,P) \wedge \neg done(T,P) \\ \wedge\, \exists\, G \; s.t. \\ \quad goal(G) \wedge goal_parameters(G_p, P_p) \wedge means_end(G,T,Dec) \\ \quad \wedge\, context_cond(Dec) \wedge can_do(G, P_p) \\ \quad \wedge\, \forall\, p \in P \; s.t. \; (\exists\, p_p \in P_p \; s.t. \; name(p,n) \wedge name(p_p,n)), \\ \qquad value(p,v) \wedge value(p_p,v) \end{array}}{\textbf{can_do(T,P)}} \quad \text{(iv)}$$

Table 1. First-order logic rules to define expected and allowed goals and tasks.

Algorithm 1 starts with an initialization phase (lines 1-2): the status of goal g is set to `uncommitted`, since no information is initially available, and the variable *means_end* is set to false. Lines 3-10 define the recursive structure of the algorithm. If the goal is and/or decomposed (line 3), the set G contains all the sub-goals of g (line 4), and the function COMPUTEFAILURES is recursively called for each sub-goal (lines 5-6). If all the status of all the sub-goals is `success` the status of g is also set to `success` (lines 7-8). If the goal is means-end decomposed (lines 9-10), G contains the set of tasks that are means to achieve the end g, and the variable *means_end* is set to true.

If g is still uncommitted (line 11) each sub-goal (or means-end decomposed task) is examined (lines 12-39). If g is and-decomposed (lines 13-23), two sub-cases are possible: (a) if the sub-goal g_i is not allowed but its status is different from `uncommitted`, and the status of g is still `uncommitted`, the status of g is set to *fail* and the cycle is broken, for the worst case – failure – has been detected (lines 14-17); (b) if the status of

g_i is `fail` (lines 18-23), the status of g is set to `fail` in turn, and the cycle is broken (lines 19-21); if g_i is in progress, the status of g is set to `in_progress`.

Algorithm 1 Identification of goal and task failures.

COMPUTEFAILURES(s : System, g : Goal, F : Failure [ ])

1 *g.status* $\leftarrow$ `uncommitted`
2 *means_end* $\leftarrow$ **false**
3 **if** $\exists g_1 \in s.goals\ s.t.\ decomposed(g, g_1, dec)$
4   **then** $G \leftarrow \{g_i \in s.goals\ s.t.\ decomposed(g, g_i, dec)\}$
5   **for each** g_i **in** G
6   **do** COMPUTEFAILURES (s, g_i, F)
7   **if** $\forall g_i$ **in** G, $g_i.status =$ `success`
8   **then** *g.status* $\leftarrow$ `success`
9   **else** $G \leftarrow \{t \in s.tasks\ s.t.\ means_end(g, t, dec)\}$
10     $meand_end \leftarrow$ **true**
11 **if** *g.status* $=$ `uncommitted`
12   **then for each** g_i **in** G
13   **do if** $and_decomposed(g, g_i, dec)$
14   **then if** $g_i.can_do =$ **false and** $g_i.status \neq$ `uncommitted`
15   **and** *g.status* $=$ `uncommitted`
16   **then** *g.status* $\leftarrow$ `fail`
17   **break**
18   **else switch**
19   **case** $g_i.status =$ `fail` :
20   *g.status* $\leftarrow$ `fail`
21   **break**
22   **case** $g_i.status =$ `in_progress` :
23   *g.status* $\leftarrow$ `in_progress`
24   **else if** $means_end =$ **true**
25   **then** $g_i.status \leftarrow$ MONITORSTATUS (g_i, g)
26   **if** $g_i.can_do =$ **false and** $g_i.status \neq$ `uncommitted`
27   **then** $F \leftarrow F \cup g_i$
28   **if** *g.status* $=$ `uncommitted`
29   **then** *g.status* $\leftarrow$ `fail`
30   **else switch**
31   **case** $g_i.status =$ `success` :
32   *g.status* $\leftarrow$ `success`
33   **break**
34   **case** $g_i.status =$ `in_progress` :
35   *g.status* $\leftarrow$ `in_progress`
36   **case** $g_i.status =$ `fail` :
37             $F \leftarrow F \cup g_i$
38   **if** *g.status* $=$ `uncommitted`
39   **then** *g.status* $\leftarrow$ `fail`
40 **if** $g.should_do =$ **true and** *g.status* $=$ `uncommitted` **and** $g.comm_cond =$ **true**
41   **then** *g.status* $\leftarrow$ `fail`
42 **if** *g.status* $=$ `fail`
43   **then** $F \leftarrow F \cup g$

If g is or-decomposed or means-end (lines 24-39), it succeeds if at least one sub-goal (or task) succeeds. If g is means-end decomposed, the algorithm calls the function MONITORSTATUS, which diagnoses the execution status of a task (lines 24-25). If g_i is not allowed and its status is different from `uncommitted`, g_i is added to the set of failures (line 27), and if g is not committed its status is set to `fail` (lines 28-29). If g_i is allowed or its status is `uncommitted` (line 30), three sub-cases are possible: (a) if the status of g_i is `success`, the status of g is set to `success` and the cycle is terminated (lines 31-33); (b) if g_i is in progress, the status of g is set to `in_progress` and the loop is continued (lines 34-35); (c) if the status of g_i is `fail`, g_i is added to the set of failures, and if g is still uncommitted its status is set to `fail`.

If g is a top-level goal that should be achieved, its status is `uncommitted`, and the commitment condition is true, then the status of g is set to `fail` because no commitment has been taken (lines 40-41). If the status of g is `fail`, it is added to the list of failures (lines 42-43).

Algorithm 2 Checking for task commitment.

MONITORSTATUS(t : Task, g : Goal)
1 *start_time* $\leftarrow$ GETACTIVATIONTIME(g)
2 $\langle$*activity, time_limit*$\rangle \leftarrow t$.GETCOMMITMENTCONDITION()
3 **if** $\exists$*tm* : Time $s.t.$ $happened$(*activity, tm*)
4 **then if** *tm* $>$ *start_time* $+$ *time_limit*
5 **then return** `fail`
6 **else** **return** `uncommitted`
7 $\langle$*next_node, time_limit*$\rangle \leftarrow$ *act*.GETNEXT()
8 **return** CHECKNODE(*next_node, time_limit, tm*)

Algorithms 2 and 3 describe how to diagnose task failures. The general idea is to describe a task as a workflow of activities each occurring within well-defined time limits (we refer to this formalism with the term "timed activity diagram"). Successful completion of an activity a is associated to the happening of an event e (happens(e)$\rightarrow$ success(a)). Activities can be connected sequentially, in parallel (by fork and join nodes), and conditionally (through branch and merge nodes). A graphical example of this formalism is given in Fig.4. The allowed branch of a decision point is defined by a branch condition. At any instant, a well-formed model has exactly one allowed branch.

Algorithm 2 is invoked by Algorithm 1 to check the status of a particular task; its parameters are a task t and a goal g linked through means-end. Line 1 sets variable *start_time* to the time when g was activated; line 2 sets the *activity* that should be executed within a certain *time_limit* since the activation of g (the commitment time). If *activity* happened at time tm but beyond the commitment time, the algorithm returns task failure (lines 3-5). If this activity did not happen, the task is uncommitted (line 6). If no failure or uncommittment has been identified, the algorithm retrieves the next node (and its time limit) from task specification (line 7), and the recursive function CHECKNODE is called to check if *next_node* occurred within *tm*+*time_limit* (line 8). Supported node types are activity, branch and merge, fork and join, end.

The behavior of Algorithm 3 depends on node type. If the node is an activity (lines 2-10), it checks if the event for that activity happened within the time limit.

Algorithm 3 Diagnosis task execution.

```
CHECKNODE(node : Node, time_limit : Time, start_time : Time)
 1  switch
 2    case node.type = activity :
 3      if ∃t : Time s.t. happened(node, t)
 4        then if t > start_time + time_limit
 5              then return fail
 6            ⟨next_node, time_limit⟩ ← node.GETNEXT()
 7            return CHECKNODE(next_node, time_limit, t)
 8        else if start_time + time_limit < GETCURRENTTIME()
 9              then return fail
10              else return in_progress
11    case node.type = fork :
12      for each ⟨in_node_i, in_limit_i⟩ ∈ node.GETFORKS()
13      do InState_i ← CHECKNODE(in_node_i, in_limit_i, start_time)
14      if ∃state ∈ InState s.t. state = fail
15        then return fail
16      if ∀state ∈ InState s.t. state = joined
17        then ⟨out_node,out_limit⟩ ← node.GETOUTNODE()
18            t ← node.GETMAXENDTIME()
19            return CHECKNODE(out_node, out_limit, t)
20        else return in_progress
21    case node.type = join : return joined
22    case node.type = end : return success
23    case node.type = branch :
24      for each ⟨opt_node_i, opt_limit_i, cond_i⟩ ∈ node.GETOPTIONS()
25      do if EVAL(cond_i) = true
26          then OptState_i ← CHECKNODE(opt_node_i, opt_limit_i, start_time)
27                if OptState_i = merged
28                  then idx ← i
29          else if ∃t : Time s.t. happened(opt_node_i, t)
30                  then OptState_i ← fail
31                  else OptState_i ← uncommitted
32      if ∃state ∈ OptState s.t. state = fail
33        then return fail
34      if idx ≠ NIL
35        then t ← node.GETBRANCHENDTIME(idx)
36            ⟨out_node,out_limit⟩ ← node.GETOUTNODE()
37            return CHECKNODE(out_node, out_limit, t)
38        else return in_progress
39    case node.type = merge : return merged
```

If it happened after time limit, it returns failure (line 5). If the event happened within time limit, the algorithm recursively checks the next node (lines 6-7). If the event for that activity hasn't happened so far: (a) if time limit has expired failure is returned (line 9); (b) if the activity is still within its time limit `in_progress` is returned (line 10). When examining a fork, CHECKNODE is recursively called for all the forks (lines

12-13). If any fork failed a `fail` value is returned (line 15). If all the forks joined (line 16) a recursive check is performed on the node that follows the join (lines 17-19), with time limit starting from the last joined fork. Otherwise, the algorithm returns `in_progress` (line 20). When a join is met, the special state `joined` is returned (line 21). The end of the model returns `success` (line 22). If a branch is found (line 23), all its branches should be checked. If the condition for a specific branch evaluates to true (line 25) that branch is recursively checked (line 26). If the branch merged, idx is set to the branch index that merged (lines 27-28). If the branch condition is false, the first node of the branch is checked: if it happened, the algorithm returns `fail` (line 30), otherwise `uncommitted` (line 31). In lines 32-33, if any failure in the branches has been detected, failure is returned. If idx is not null (line 34) a recursive check on the node following the merge is performed (lines 35-37); if idx is null the algorithm returns `in_progress` (line 38). When a merge node is met, the special value `merged` is returned.

6 Case study: smart homes

We show now a promising application for our architecture, emphasizing how requirements models are used to define and check allowed and expected behaviors, and how the architecture performs the MDC cycle. Our case study concerns smart homes: a patient lives in a smart home, a socio-technical system supporting the patient in everyday activities (such as eating, sleeping, taking medicine, being entertained, visiting doctor). Both smart home and patient are equipped with AmI devices that gather data (e.g., patient's health status, temperature in the house) and enact compensations (e.g., open the door). The partial goal model in Fig.3 represents the requirements of the patient; due to space limitations, we present here only the top-level goal "Have breakfast".

Goal $g1$ is activated when the patient wakes up (activation event); a commitment to achieve $g1$ should be taken (either by the patient or by other agents) within two hours since goal activation. Four different contexts characterize the scenario: in $c1$ the patient is autonomous, in $c2$ the patient is not autonomous, in $c3$ the patient is at home, in $c4$ the patient is not at home. If the patient is autonomous ($c1$ holds) $g1$ is decomposed into the subtree of goal "Eat alone" ($g2$); if $c2$ holds $g1$ is decomposed into the subtree of goal "Get eating assistance" ($g22$). In the former case, $c3$ enables the subtree of goal "Eat at home" ($g3$), whereas $c4$ enables the subtree of goal "Eat outside" ($g7$). When eating at home, the patient has to prepare food ($g4$), eat breakfast ($g5$), and clean up ($g6$). Goal $g4$ is means-end to two alternative tasks: "Prepare autonomously" ($p1$) and "Order catering food" ($p2$). The latter task requires interaction with the external actor "Catering service", which should fulfill goal "Provide food" in order to execute successfully $p2$. The other subtrees of Fig.3 are structured in a similar way, thus we don't detail them here.

Our requirements models characterize each task with a precondition (possibly empty) that, if false, inhibits the execution of the task. If a task is executed but its precondition is false, a failure occurs (see rule (iv) in Table 1). For instance, a possible precondition for task "Prepare autonomously" is that in the house there are both bread and milk; a

precondition for task “Order catering food” is that the house is provided with a landline phone or the patient has a mobile phone.

Fig.4 is a timed activity diagram for task $p1$. The activity diagram starts with the activation of the goal “Prepare food”. When the patient enters the kitchen ($a1$), there is evidence that she is going to prepare food. If this doesn’t happen within 45 minutes after the goal activation the task fails. After $a1$, a fork node creates two parallel execution processes. In the first fork, the patient should open the fridge ($a2$) and put the milk on stove ($a4$); in the second fork, the bread cupboard should be opened ($a3$) and bread has to be put on the table ($a5$). The forks are then merged, and the next activity is to turn on the stove ($a6$) within a minute since the last completed activity. Sequentially, the task requires the stove to be turned off within 5 minutes ($a7$) and the milk to be poured into the cup ($a8$).

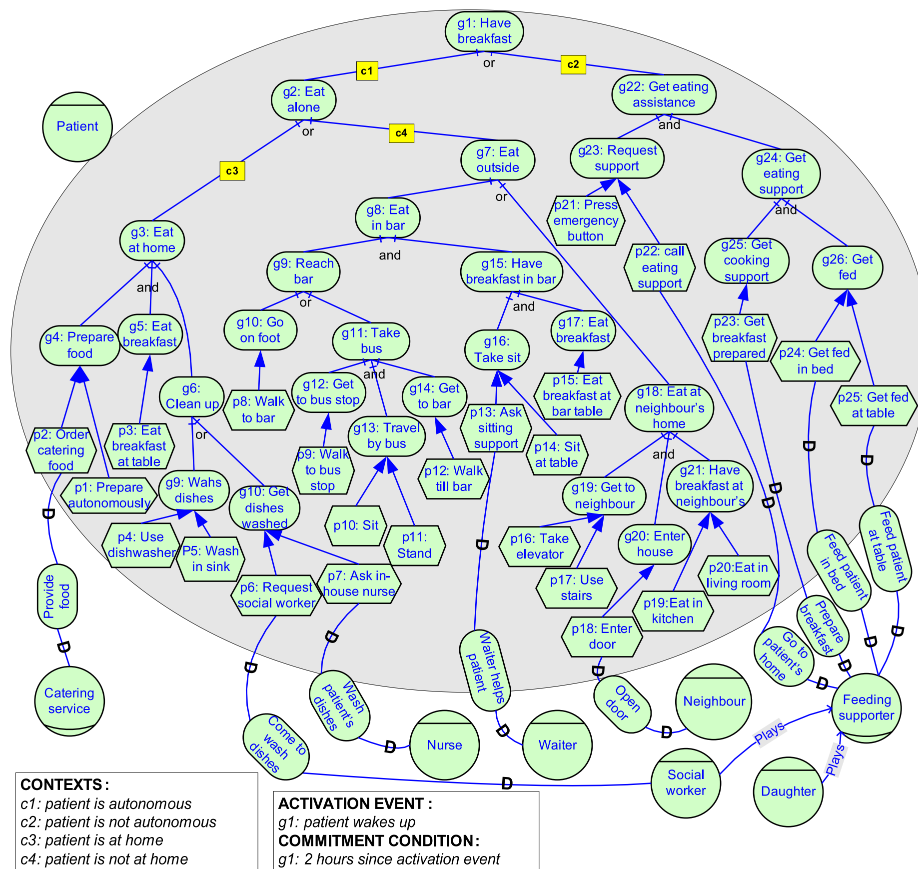

Fig. 3. Contextual goal model describing the patient health care scenario.

We conclude this section with a description of a possible reconfiguration process. Let’s suppose that patient Mike wakes up at 8.00 am. Mike is autonomous ($c1$) and at home ($c3$); the goal $g1$ is expected, and the subtree of $g3$ is the only allowed one (see

rules (i) and (ii) in Table 1). At 8.20 am Mike enters the kitchen: checking the activity diagram for $p1$ against this event (see Algorithm 2) changes the status of the goal $g4$ to `in_progress`. In turn, this status is propagated bottom-up till $g1$ (see Algorithm 1). At 8.25 Mike hasn't neither opened the fridge nor opened the bread cupboard. This violates the specification of $p1$ (Fig. 4). The reconfiguration strategy selector component selects to push the system, and the system pushing component sends a notification to the patient through an SMS message (and the time limit within which executing $a2$ and $a3$ should be reset). This changes the mind of Mike, which opens the fridge ($a2$), opens the bread cupboard ($a3$), and puts bread on table ($a5$). These events are compliant with the task specification of Fig. 4, thus the task is evaluated as `in_progress` by Algorithm 3. Anyhow, Mike does not put milk on stove ($a4$) within one minute since $a2$, therefore a new failure is diagnosed. The compensation to address this failure is to automate $p2$, and the task assigner component assigns it to a catering service. An alternative scenario evolution is that Mike exits house (the context $c4$ is true, $c3$ is not valid anymore). This would change the tasks that can happen: the subtree of $g7$ becomes the only possible one, and this influences all its sub-goals (rule (iii) in Table 1) and the tasks linked to leaf-level goals (rule (iv) in Table 1).

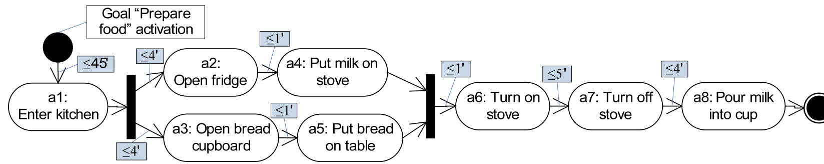

Fig. 4. Timed activity diagram for monitoring the task "Prepare autonomously".

7 Related work

Self-adaptive software has been introduced by Oreizy et al. [12] as an architectural approach to support systems that modify their behaviour in response to changes in the operating environment. This class of systems performs self-reconfiguration according to the criteria specified at development time, such as under what conditions reconfiguring, open/closed adaptation, degree of autonomy. The building units for self-adaptive software should be components and connectors. Compared to our work, the solution proposed in [12] is generic and flexible to many reconfiguration criteria, whereas we suggest particular types of models, that is requirements models.

Rainbow [1] is an architecture-based framework that enables self-adaptation on the basis of (i) an externalized approach and (ii) software architecture models. The authors of Rainbow consider architecture models as the most suitable abstraction level to abstract away unneccessary details of the system. Moreover, the usage of architectural models both at design- and at run-time promotes the reuse of adaptation mechanisms. Our proposal shares many features with Rainbow, but differs because we use higher level models to support the ultimate goal of any software system, that is to meet its

requirements. The main drawback of our choice is that establishing traceability links between requirements and code is more complex.

Sykes et al. [13] propose a three-layer architecture for self-managed software [14] that combines the notion of goal with software components. This approach is based on a sense-plan-act architecture made up of three layers: *goal management* layer defines system goals, *change management* layer executes plans and assembles a configuration of software components, *component* layer handles reactive control concerns of the components. Our proposal exploits a more elaborate goal representation framework, follows different planning (predifined plans instead of plan composition), and enacts different reconfiguration processes.

Wang's architecture for self-repairing software [7] uses one goal model as a software requirements model, and exploits SAT solvers to check the current execution log against the model to diagnose task failures. We propose a broader approach, adopting part of the complete Requirements Engineering framework proposed by Jureta et al. [8]. We use more expressive goal models, provide an accurate specification of tasks based on timed activity diagrams, allow for specifying multiple contexts that modify expected behavior, and support dependencies on other actors / systems.

Feather et al. [2] propose an approach addressing system behaviour deviations from requirements specifications; they introduce an architecture (and a development process) to reconcile requirements with behaviour. This reconciliation process is enacted by jointly anticipating deviations at specification time and solving unpredicted situations at runtime, and examine the latter option using the requirements monitoring framework FLEA [15]. FLEA is used in conjunction with the goal-driven specification methodology KAOS [3]. Our architecture differs in the usage of different requirements models (Tropos rather than KAOS), support to a wider set of failures ([2] is focused on obstacle analysis), and applicability to scenarios composed of multiple interacting actors (such as Ambient Intelligence ones).

Robinson's ReqMon [16] is a requirements monitoring framework for specific usage in enterprise systems. ReqMon integrates techniques from requirements analysis (KAOS) and software execution monitoring, and provides tools to support the development of requirements monitors. Although ReqMon's architecture covers all the reconfiguration process, accurate exploration is provided only for the monitoring and analysis phases. Our approach has broader applicability, can diagnose a larger set of failure types, and supports more reconfigurations mechanisms; on the contrary, ReqMon is particularly suitable for enterprise systems.

8 Discussion and conclusion

We have proposed a novel architecture for self-configuring systems founded on principles adopted from Goal-Oriented Requirements Engineering, externalized adaptation, and BDI paradigm. Our approach adds self-reconfiguration capabilities to a wide variety of system, among which Ambient Intelligence scenarios and Socio-Technical Systems. The architecture is a model-based one, with requirements models used to specify what can, should, and should not happen. We have detailed the main mechanisms for monitoring and diagnosis, which describe how requirements models are checked against

monitored information. We also introduced a case study – smart homes – to show how a realization of the architecture works in practice.

Several aspects will be addressed in future work. Firstly, we need a complete implementation of our architecture, as well as further experimentation on the smart-home case study. We also want to extend our framework so that it deals with a broader class of monitored phenomena, including attacks and failures caused by false doamin assumptions. Finally, we propose to introduce mechanisms through which a system can extend its variability space through collaboration with external agents.

References

1. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: architecture-based self-adaptation with reusable infrastructure. Computer **37**(10) (Oct. 2004) 46–54
2. Feather, M., Fickas, S., Van Lamsweerde, A., Ponsard, C.: Reconciling system requirements and runtime behavior. IWSSD '98 (1998) 50–59
3. Dardenne, A., Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Science of computer Programming (1993) 3–50
4. Emery, F.: Characteristics of socio-technical systems. London: Tavistock (1959)
5. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An agent-oriented software development methodology. JAAMAS **8**(3) (2004) 203–236
6. Rao, A., Georgeff, M.: An abstract architecture for rational agents. Proceedings of Knowledge Representation and Reasoning (KR&R-92) (1992) 439–449
7. Wang, Y., McIlraith, S., Yu, Y., Mylopoulos, J.: An automated approach to monitoring and diagnosing requirements. ASE '07 (2007) 293–302
8. Jureta, I.J., Mylopoulos, J., Faulkner, S.: Revisiting the core ontology and problem in requirements engineering. In: RE 08, Barcelona (2008)
9. Ali, R., Dalpiaz, F., Giorgini, P.: Location-based software modeling and analysis: Tropos-based approach. ER 2008 (2008) 169–182
10. Clark, J., et al.: Xsl transformations (xslt) version 1.0. W3C Recommendation **16**(11) (1999)
11. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM Transactions on Database Systems (TODS) **22**(3) (1997) 364–418
12. Oreizy, P., Medvidovic, N., Taylor, R.: Architecture-based runtime software evolution. ICSE 1998 (1998) 177–186
13. Sykes, D., Heaven, W., Magee, J., Kramer, J.: From goals to components: a combined approach to self-management. SEAMS 2008 (2008) 1–8
14. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. ICSE 2007 (2007) 259–268
15. Fickas, S., Feather, M.: Requirements monitoring in dynamic environments. RE '95 (1995) 140
16. Robinson, W.: A requirements monitoring framework for enterprise systems. Requirements Engineering **11**(1) (2006) 17–41