Noname manuscript No.
(will be inserted by the editor)

Adaptive Socio-Technical Systems: A Requirements-Based Approach

Fabiano Dalpiaz, Paolo Giorgini, John Mylopoulos

Received: date / Accepted: date

Abstract A Socio-Technical System (STS) consists of an interplay of humans, organizations and technical systems. STSs are heterogeneous, dynamic, unpredictable and weakly controllable. Their operational environment changes unexpectedly, actors join and leave the system at will, actors fail to meet their objectives and underperform, dependencies on other actors are violated. To deal with such situations, we propose an architecture for STSs that makes an STS self-reconfigurable, i.e. capable of switching autonomously from one configuration to a better one. Our architecture performs a Monitor-Diagnose-Reconcile-Compensate (MDRC) cycle: it monitors actor behaviours and context changes, diagnoses failures and under-performance by checking whether monitored behaviour is compliant with actors goals, finds a possible way to address the problem, and enacts compensation actions to reconcile actual and desired behaviour. Compensation actions take into account the *autonomy* of participants in an STS, which cannot be controlled. Our architecture is *requirements-driven*: we use extended Tropos goal models to diagnose failures as well as to identify alternative strategies to meet requirements. After presenting our conceptual architecture and the algorithms it is founded upon, we describe a prototype implementation applied to a case study concerning smart-homes. We also provide experimental results that suggest that our architecture scales well as the size of the STS grows.

Keywords Self-Adaptive Software · Socio-Technical Systems · Goal Models · Requirements Engineering

DISI, University of Trento

1 Introduction

Socio-Technical Systems (STSs) involve a rich interplay of human actors—the traditional constituents of organizations—and technical systems [20]. The engineering of STSs is not a mere extension of traditional techniques for distributed software or information systems. Specific factors need to be considered: the volatility and unpredictability of the operational environment, the heterogeneity, autonomy and uncontrollability of participating actors, and the social dependencies that hold or emerge between them. A major challenge for STSs is to guarantee (or at least facilitate) a purposeful and effective interaction between actors, to ensure that overall objectives are achieved.

By interaction we refer here to *social interaction* as in multi-agent systems [44], the establishment and evolution of social relationships on the basis of the messages exchanged between actors. To successfully design STSs, interaction with user interfaces [38] has to be taken into account too. However, we do not consider this type of interaction in this paper.

We encounter many examples of STSs in our daily lives. A smart-home that helps heart patients carry out everyday activities is an STS. It includes cameras, biomedical sensors, and other devices, human actors such as the patient itself, social workers, caregivers, doctors. The LinkedIn[1] social network is an STS consisting of the LinkedIn website, professionals looking for a job, business scouts searching for interesting profiles, companies advertising vacancies, and the LinkedIn Corporation that aims at increasing revenues by offering professional relationships management. Likewise, a logistics department in a wholesale fruit company is

[1] www.linkedin.com

an STS. A logistics management system supports purchases while warehouse personnel ensures fruit arrives to the warehouse just in time for delivery. These three examples of STSs differ in their degree of openness: a social network is extremely open, internal logistics consists of a relatively static set of actors, while a smart-home is somewhere in-between.

As these examples demonstrate, STSs are more than heterogeneous systems. Unlike these systems, the participants in an STS are autonomous, the operational environment is volatile, and actors are linked via social dependencies, rather than simply in terms of method invocations.

To effectively cope with the volatility of their operational environment, STSs design should provide STSs with self-reconfiguration capabilities. A configuration of an STS is a network of social dependencies among the participating subsystems, wherein each subsystem executes specific tasks and depends on other subsystems to achieve its own requirements. A system is self-reconfigurable if it can switch autonomously to a different configuration in response to failures and underperformance. Each subsystem in an STS has to adapt to fulfil its own requirements.

Numerous research trends have explored self-reconfiguration, including self-adaptive software [35], self-managed systems [29], autonomic [26] and pervasive computing. They all recognize the importance of integrating monitoring, diagnosis, planning and reconfiguration functions into a system architecture. They enable software to evolve and adapt to open, dynamic environments so that it can continue to fulfil its purpose.

Most approaches exploit architectural models to diagnose failures and to plan for alternatives [24, 29, 13]. Some frameworks are based on higher-level models, that represent requirements. KAOS [18] was used in an architecture [21] that reconciles requirements with system behaviour, by anticipating deviations at design time and monitoring for unpredicted circumstances at runtime. Recently, goal models and code instrumentation have been used to add self-repair capabilities to legacy software systems [47]. However, neither approach is expressly suited for STSs, because they do not account for the *social context*, i.e., the multiplicity of actors and their dependencies within an STS.

We propose a conceptual architecture for self-reconfigurable STSs that is based on goal-oriented requirements models. Its purpose is to support sub-systems (actors) in achieving their requirements: whenever requirements are at risk, the architecture suggests an alternative plan. The architecture conducts a Monitor-Diagnose-Reconcile-Compensate (MDRC) cycle. An extension of Tropos [8] goal models captures actor intentionality, social dependencies between them, and the interplay between the operational context and actor goals [1], as well as domain assumptions that should not be violated. Tropos goal models are also enriched with triggering and fulfilment conditions, plan specifications, and time-outs. These extensions allow us to model monitoring and compensation rules that define the ingredients of system adaptivity.

Our architecture is expressly thought for STSs: (i) the system comprises not only of technical subsystems, but also of social actors (humans and organizations); (ii) diagnosis mechanisms can identify violated dependencies between actors; (iii) planning mechanisms include both executing plans and establishing dependencies between actors; (iv) as actors are autonomous, they are free to accept or ignore suggested alternatives.

Our diagnosis mechanism is based on actor intentions [7] and social dependencies [49, 8]. Whenever an actor's goal is triggered, the actor should adopt an intention for that goal. He can either execute plans or interact with others by establishing a dependency for that goal. On the basis of such atomic behaviour on the part of actors, the STS maintains a dependency network throughout its execution where actors depend on one another for the fulfilment of their respective goals.

Consider a smart-home STS, and suppose patient Marco's goal is to have breakfast. Either he executes a plan to prepare breakfast—get milk out of the fridge, heat up milk on stove, etc.—or he calls a catering service asking for home delivery. The second option establishes a social relation where Marco depends on the catering service for the delivery of breakfast.

Failures are diagnosed if (i) the actor adopts no intention to achieve its goal, e.g. Marco neither prepares breakfast nor calls a catering service; (ii) a plan is carried out incorrectly, e.g. Marco heats up an empty pot; (iii) a plan takes longer than expected, e.g. Marco is taking two hours to prepare breakfast; (iv) a dependency is not fulfilled, e.g. the catering service doesn't deliver breakfast in time.

Once a discrepancy between expected and actual behaviour is detected, our architecture (i) looks for an alternative configuration to fulfil actor objectives (*reconciliation*) and (ii) attempts to reconfigure the STS to ensure actors achieve their objectives (*compensation*). A configuration comprises tasks the actors should perform, social dependencies actors should establish, and compensation actions [23] to revert the effects of tasks that were in the old configuration and are not in the new one. Since components of an STS are autonomous agents, rather than conventional software components, they cannot be forced to achieve goals or execute ac-

tions. Thus, the architecture reminds actors of their goals and suggests specific plans to achieve these goals.

To offer evidence for its feasibility, we describe a prototype implementation of our architecture developed in Java and we apply it to a smart-home case study taken from the EU-sponsored Serenity project[2]. The scenario involves a patient living in a smart-home that supports him in everyday activities with the aid of external actors (doctor, social worker, catering service). We also devise a methodology that supports system engineers to apply our proposed architecture to an existing system, in order to create an adaptive STS.

The rest of the paper is structured as follows. Section 2 describes the research baseline for our architecture. Section 3 details our smart-home case study. Section 4 presents the conceptual architecture for adaptive STSs. Section 5 shows the diagnosis and reconfiguration mechanisms exploited by our architecture. Section 6 evaluates the implemented architecture for our case study and presents the results of scalability experiments. Section 7 discusses how to apply our architecture to an existing STS. Section 8 contrasts our approach to related work. Section 9 summarizes the contributions of this work and sketches future directions.

2 Research baseline

We articulate our research baseline in three areas: (i) goal models and actor dependency models express stakeholders requirements and the multi-agent plans to fulfil these requirements; (ii) the BDI paradigm guides diagnosis and plan selection for each actor; (iii) compensation mechanisms are exploited to deal with failures.

Goal models. Our architecture for self-reconfigurable STSs is based on requirements models. Goal-oriented techniques [18] constitute the state-of-the-art in Requirements Engineering (RE). Among existing approaches, we adopt Tropos [8]—which is founded on the *i** [49] modelling notation—as our baseline, for it exploits models of systems made up of socially interacting actors depending on each other for the fulfilment of their own goals and supports soft-goals as a means to choose between alternatives. Moreover, Tropos, unlike *i**, uses such models not only to represent stakeholder requirements but also system architectures.

Fig. 1 presents a sample Tropos goal model with two agents (Patient and Supermarket) that depend on one another for the fulfilment of goal Deliver food. The top-level goal of Patient is Have lunch; it is OR-decomposed to sub-goals Prepare lunch and Get lunch prepared. The

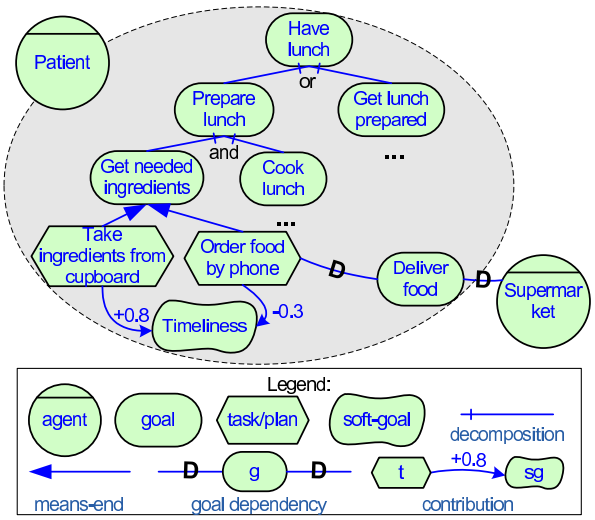


Fig. 1 A sample Tropos goal model

achievement of either sub-goal is sufficient to achieve the root goal. Goal Prepare lunch is AND-decomposed to sub-goals Get needed ingredients and Cook lunch. Goal Get needed ingredients is a leaf-level one, and is linked to two plans—Take ingredients from cupboard and Order food by phone—through a means-end decomposition. The successful execution of either plan is sufficient to achieve goal Get needed ingredients. The latter plan introduces a dependency for goal Deliver food on agent Supermarket. Supermarket will also have its own goal model for goal Deliver food. Plans contribute negatively or positively to soft-goals. Contributions are in the range $[-1, +1]$. Soft-goal Timeliness is contributed positively by task Take ingredients from cupboard $(+0.8)$ and negatively by task Order food by phone (-0.3).

Goal models capture software variability at the level of requirements [37]. They include variation points—e.g. OR-decompositions—that introduce variability, i.e. multiple alternatives to meet goals. For instance, an OR-decomposed goal is achieved by any of its sub-goals.

Recently, Jureta *et al.* [28] revisited the so-called "requirements problem"—what it means to satisfy a set of requirements—showing the need for concepts that represent preferences, domain assumptions, etc. Here, we adopt some elements from their proposal, notably goals, soft-goals, plans, and domain assumptions.

BDI Paradigm. The Belief-Desire-Intention (BDI) paradigm [39, 40] conceives an agent as consisting of beliefs about the world, desires (or goals) it aims at attaining, and intentions, that is plans it has decided to execute in order to fulfil its goals. The BDI paradigm originates from the practical reasoning theory [7], which focuses on the role of intentions to characterize agents. We rely on BDI principles for:

[2] Serenity: System Engineering for Security and Dependability, http://www.serenity-project.org

- *diagnosis*: to achieve his goals, an agent has to adopt an intention and to carry out a suitable plan. Diagnosis is a bottom-up process: the status of current plans is used to determine the status of goals. If the patient has not taken any ingredient from the cupboard and has not ordered food by phone, the status of goal Get needed ingredients might be set to failure.
- *reconciliation*: as a failure is identified, an alternative plan should be devised. Given the current top-level goals, goal trees are explored top-down to identify the most adequate set of tasks to achieve the goals, as well as the dependencies to be established.

Compensation. Compensation is the capability of a system to nullify the effect of started plans when a failure occurs and the system needs to switch to another configuration that does not include those plans. We take inspiration from Databases where the concept of Saga [23] has been explored as a means to compensate for a failed long-running transaction. A Saga is a set of atomic transactions treated as a unique entity, and any transaction is provided with a compensation transaction semantically equivalent to an undo action. If one step of the Saga fails, compensation is executed for all completed or in-execution transactions of the Saga.

In the spirit of Sagas, we attach compensation actions to plans, and execute compensation actions to erase or mitigate the effects of failed plans. After reverting these effects, our compensation mechanism enacts a reconfiguration to ensure the STS fulfils its mission.

3 Case study: smart-homes

We describe now a promising application for our architecture, which relates to smart-homes for supporting elderly or handicapped people. This scenario is a variant of the "smart items" case study [11] of the Serenity project: a patient lives in a smart-home and is part of a socio-technical system supporting the patient in everyday activities (e.g. eating, sleeping, taking medicine, being entertained, consulting with doctor). Both the smart-home and the patient are equipped with Ambient Intelligence devices that collect data—such as patient's vital signs and the temperature in the bedroom—and enact response actions—such as opening the door or alerting the medical centre if the patient feels giddy. Our prototype smart-home exploits many sensors: a pulse oxymeter to gather patient's heart rate and saturation level, a set of cameras capable of motion detection and object tracking, a wireless sensor network to authenticate doctors and social workers, magnetic fields to detect open doors and windows. The equipment in the house (lights, doors, windows, heating, and so on) is connected by the KNX communication bus[3]. In addition, the house includes a number of actuators for the entrance door, the windows and the blinds to enable automatic opening and closing. Moreover, lights and temperature can be adjusted automatically; some pieces of equipment can be raised or lowered, etc.

This information about the smart-home STS is useful to understand how the patient can achieve his goals with the aid of the smart-home system. We detail here a part of the case study where the system helps the patient conduct his pre-breakfast morning routine. Fig. 2 presents the goals of a patient via an extended Tropos goal model (details are given in Section 4.2.1). We provide just some details to ease the reading. The top-level goal g_1: Pre-breakfast morning routine is AND-decomposed to sub-goals g_2: Get out of bed, g_5: Check health, g_{10}: Take medicine, and g_{13}: Have a wash. The decomposition link from g_1 to g_{10} is labelled c_2; this means that the achievement of g_1 requires the achievement of g_{10} only if the context c_2 (patient suffers from chronic disease) holds. g_2 is OR-decomposed to sub-goals g_3: Get up autonomously and g_4: Get support to get up, which are valid alternatives if the patient is autonomous (c_3 holds) and not autonomous (c_4), respectively. To achieve goal g_4, possible means are t_4: use transfer sling and t_5: lift patient. Each of these plans originates a dependency on actor Patient assistant, for goals Get patient up with transfer sling and Get patient up by lifting, respectively. Three soft-goals are considered: efficacy, reliability, and low failure cost. Contributions from plans to these soft-goals are not shown here to keep the diagram readable.

4 An architecture for self-reconfigurable STSs

Fig. 3 presents our conceptual architecture for self-reconfigurable STSs exploiting an UML 2.0 component diagram to show architectural components and the connections among them.

Our architecture is founded on the MDRC cycle. *Monitoring* collects data about the state of the environment and the STS from a variety of sources; *Diagnosis* interprets these data with respect to requirements models to determine if all is well, if not, diagnose the problem-at-hand; *Reconcile* searches for an alternative configuration that deals with the problem-at-hand; *Compensation* takes necessary steps to ensure that the new plan can be executed. Our architecture is designed for STSs, which are inherently decentralized, distributed and heterogeneous systems. Therefore,

[3] a standard bus that allows devices in smart-homes to communicate: http://www.knx.org/

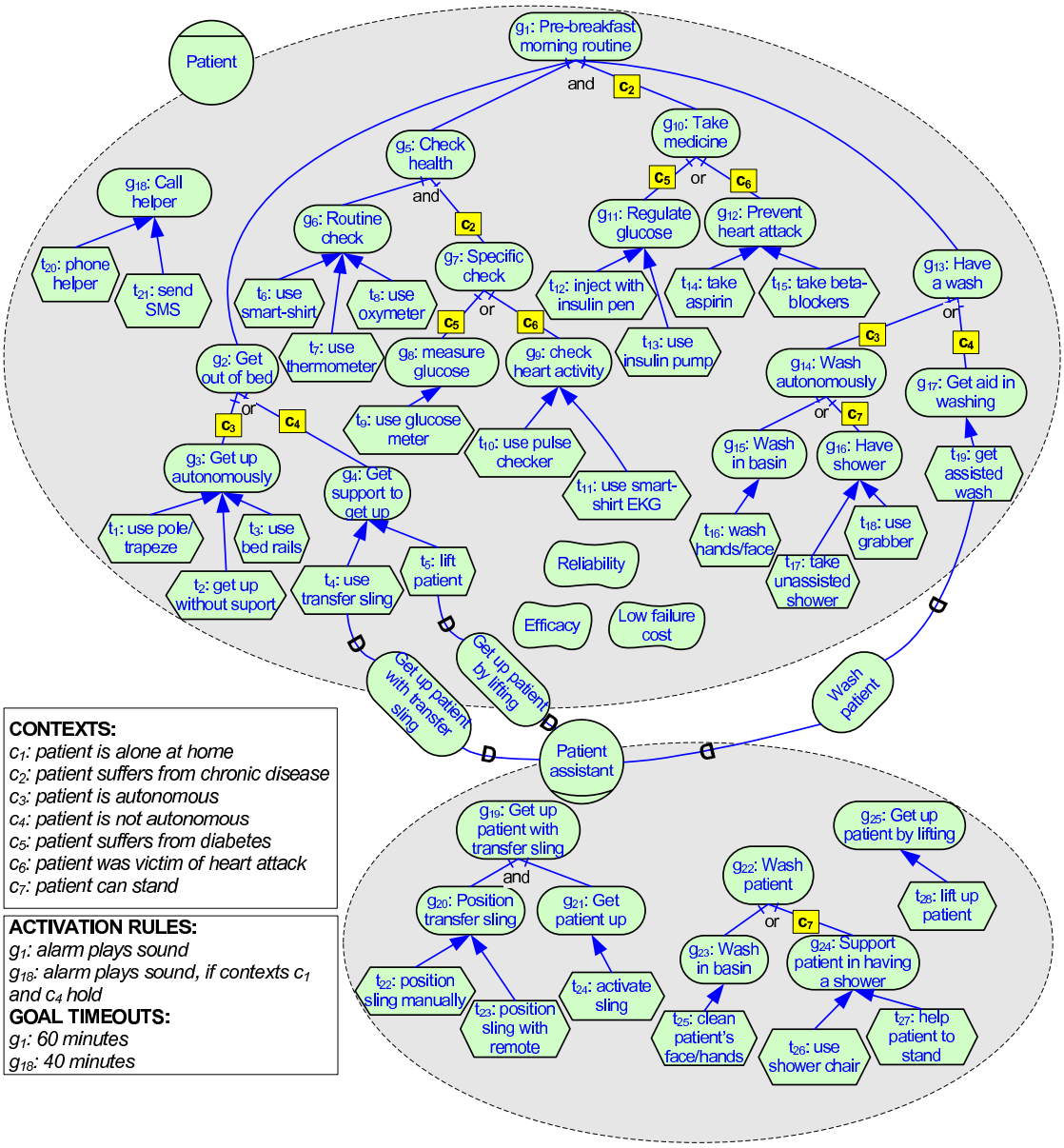


Fig. 2 Contextual goal model describing the patient health care scenario

it has to support the interaction between participating actors and functional components as well as the supervisory MDRC cycle. These interactions are supported through additional elements of the architecture:

- *Context sensors* are computational entities that provide raw data about the operational environment. In our smart-home setting, context sensors provide data about the current location of the patient, temperature and humidity levels in specific rooms, open and closed doors and windows, incoming/outgoing phone calls, the location of other actors (caregivers, doctors) within the apartment or elsewhere.
- *Agents* include all STS actors who need to be monitored to ensure that they deliver on their obligations to the system. These may be patients living in smart-homes, firemen and medical doctors in crisis management settings, air traffic controllers in charge of managing the air space around the airport they work in. Due to theri autonomy, agents cannot be controlled. They are sent directions, advice and reminders through interface *System pushes*. For instance, a patient may be reminded to take her medicine by sending an SMS to her mobile phone. Also, the system can try to assign specific tasks to other agents—establish dependencies—through interface *Task assignments*. For instance, a catering service might be asked for delivering food. A de-

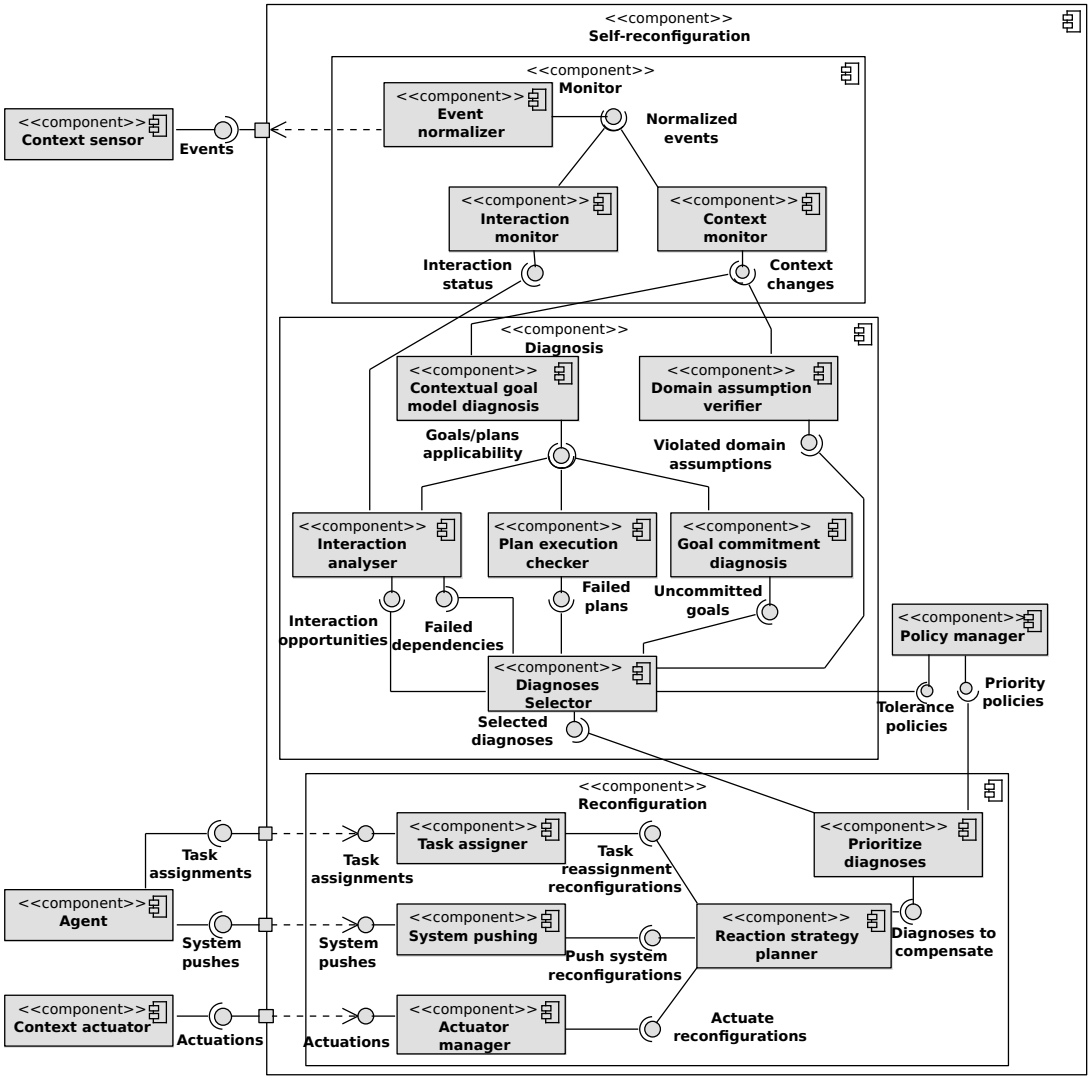


Fig. 3 Logical view on the proposed architecture for self-reconfigurable STSs

pendency is established only if the catering service accepts such request.

– *Context actuators* represent effectors in the environment that can receive commands and act. Examples of actuators are sirens, door openers, automated windows, remote light switches, automatic 911 callers. The component receives the commands to enact through the required interface *Actuations*.

The *Self-Reconfiguration* component provides the self-reconfiguration capabilities of our architecture. It consists of three sub-components. The *Monitor* component collects, filters, and normalizes received events; the *Diagnosis* component identifies failures and discovers root causes; the *Reconfiguration* component performs reconciliation and compensation in response to failures. The next subsections detail these components.

4.1 Monitor

The *Event normalizer* component initiates the monitoring function, which takes its input from the interface *Events* through appropriate ports (in Fig. 3 ports are the small squares on component borders). The collected events are normalized to a common format that expresses them over the context model (see Section 4.2.1). Normalization requires the definition of a translation schema for each raw data format. Defining these schema is an application-specific activity. If event sources provide data in standard formats (e.g., XML), transformation schemata can be defined using a transformation language (e.g., XSLT [14]). The provided interface *Normalized events* is required by the components that deal with agent interaction and contextual events. The *Interaction monitor* computes the status of existing interactions and exposes it through interface *Interaction sta-*

tus. An example is a social worker expected to visit, who calls the patient to notify she will not come. The *Context monitor* processes events related to context and exposes the interface *Context changes*. For instance, if the house entrance door is closed and an event such as $open(door, time_i)$ happens, the status of the entrance door will switch to open.

4.2 Diagnosis

Diagnosis means checking current information about the STS against requirements models. These models specify which goals and plans agents are expected to achieve and execute, as well as the domain assumptions that shall not be violated. The richer the requirements models are, the more accurate diagnosis will be. By contrast, the granularity of events that can be observed depends on technological and pragmatic aspects. Detecting if a patient is sitting on a sofa is readily feasible (through pressure sensors), while detecting if she is handling a knife the wrong way is more complex.

This section is split into two sub-sections. Section 4.2.1 introduces the requirements models that are used by our architecture. Section 4.2.2 describes how the components interact to perform diagnosis.

4.2.1 Requirements models

We use different models to specify correct behaviour: goal models, a context model, plan specifications, and domain assumptions. Together, these models capture system requirements. However, this paper is not concerned with requirements elicitation and validation. We assume requirements are created using traditional elicitation and analysis methodologies.

Goal models. We exploit a variant of Tropos goal models expressly thought for runtime usage.

- *Contexts*[4] are linked to decompositions to express when certain alternatives are applicable or required. We exploit the contextual extension of Tropos [2] and associate contexts to variation points. In Fig. 2, the AND-decomposed goal g_1: Pre-breakfast morning routine includes a contextual decomposition link to goal g_{10}: Take medicine: the achievement of g_{10} is required to achieve g_1 only if context c_2 holds, that is if the patient suffers from a chronic disease. Goal g_{11}: Regulate glucose is a valid alternative to achieve the OR-decomposed goal g_{10} only if context c_5 holds (the patient suffers from diabetes).
- *Activation rules* are associated to top-level goals. An activation rule is composed of a triggering event and a precondition. The goal is activated when the triggering event happens, if the precondition holds. Fig. 2 includes activation rules for the top-level goals of the patient: goal g_1: Pre-breakfast morning routine is activated as the alarm rings, whereas goal g_{18}: Call helper is triggered as the alarm rings, provided that the patient is alone at home and is not autonomous.
- *Declarative goals* are goals that are met only if their achievement condition is fulfilled. Their satisfaction is independent of the satisfaction of sub-goals or plans linked by means-end decompositions. Achievement conditions are expressed as states over the context model. For instance, a possible achievement condition for g_6: Routine check is that an electronic medical report is sent to the medical centre. Declarative goals deal with uncertainty, as they allow for actors to achieve goals by means of unforeseen ways.
- *Time limits* are associated to top-level goals to define the maximum amount of time within which an agent has to achieve a goal. In Fig. 2, goal g_1 should be achieved within 60 minutes, whereas goal g_{18} within 40 minutes.
- *Plans* are sets of actions. In a contextual goal model, plans are connected to goals by means-end decompositions: an agent executes a plan to achieve a goal. To support plan monitoring and diagnosis, we specify plans using a simple and flexible language. Each action is performed correctly if its postcondition is achieved within a time limit and, at that time, the associated precondition holds. If an action is not performed correctly, the plan including it fails. We provide more details later in this section.

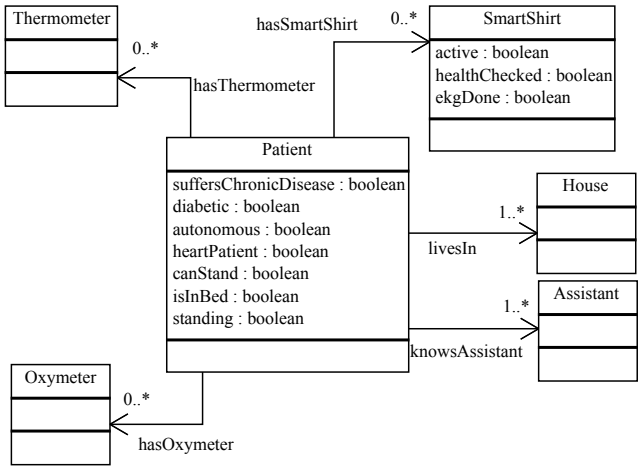


Fig. 4 Part of the context model for the scenario of Fig. 2

[4] A context is a partial state of the world that is relevant to an actor's current intentions and status [2]

Context model. Associating context descriptions to variation points doesn't suffice to verify if a certain context holds. Context analysis [2] is a refinement process to understand abstract contexts (e.g. "the patient is sick") as formulae of observable facts (e.g. "the patient's oxymeter measures a saturation level below x" and "the patient didn't wake up this morning"). Facts are expressed with respect to a context model, which describes context in terms of entities, attributes and relations. In the requirements models used by our architecture, contexts are formulae of facts on the context model, which is represented as a class diagram. Fig. 4 shows part of the context model for the smart-home scenario. The class Patient is characterized by a set of boolean attributes that express whether he suffers from a chronic disease, he is diabetic, autonomous, is a heart patient, can stand, is currently in his bed, and is currently standing. A patient has also associations to other classes: for instance, he can have multiple smart shirts (association *hasSmartShirt* to class SmartShirt), and he knows at least one assistant (association *knowsAssistant* to class Assistant).

At runtime, our architecture deals with instances of the classes in the context model. For instance, we might have two instances of Patient (*jim* and *bob*), both living in the same House *smartHome1*. Each of them might have one SmartShirt (*ss1* and *ss2*, respectively). They might be assisted by the same Assistant *mike*.

Plan specification. We specify a plan as a set of actions to be carried out by an agent. Each action is characterized in terms of (i) a *postcondition*, the expected effect produced by performing that action; (ii) a set of *preconditions*, the state of the world that should hold to enable an agent to perform that action; (iii) a *time limit* within which the action should be carried out. Preconditions can be critical or non-critical. If an action postcondition is fulfilled and a critical precondition does not hold, the action leads to plan failure.

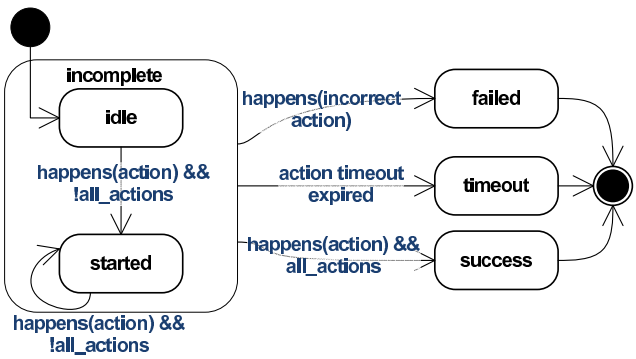


Fig. 5 Statechart showing the possible state transitions for a plan

Table 1 shows a semi-formal representation of some plans in the scenario of Fig. 2. Plan semantics is formalized in Fig. 5. The statechart represents the possible states for a plan and the transitions between these states. The meaning of each state is the following:

- *idle*: None of the actions in the specification has been executed so far, the time limit is not expired. This state includes situations in which some actions started, and none of them is done yet.
- *started*: at least one (but not all) action has been performed successfully, the time limit is not expired. This state shows the non-atomicity of plans: several actions have to be performed before completing the plan, and in that situation the plan is in progress;
- *success*: all actions in the specification have been executed successfully within the time limit. This is a terminal state: once a plan succeeds, there is no need for the system to monitor its execution;
- *failed*: at least one action in the specification has been performed, and at least one critical precondition was false. This is another terminal state. Plan failure triggers an adaptation process, unless the application policies tell to ignore such failure;
- *timeout*: the time limit expired. Timeout is a particular type of failure, and is a terminal state.

Though our specification does not prescribe a complete order for actions execution, they can be partially ordered. This is doable by using the postcondition of an action as critical precondition for another. This way, the plan is carried out correctly only if the second action is performed after the first one.

We illustrate our plan specification language on the smart-home scenario. Suppose the patient's goal is to get up, and the smart-home system has to ensure he achieves such objective. A possible way for the patient to get up is to grab a bed pole. As in Table 1, plan t_1: use pole/trapeze is specified by the following actions:

1. *Precondition*: patient p is in bed b, bed b has a pole pl; *Postcondition*: the bed pole pl is touched; *Time limit*: 5 minutes;
2. *Precondition*: patient p is in bed b (critical), bed b has pole pl, pl is touched; *Postcondition*: patient p stands up; *Time limit*: 10 minutes;

The plan is executed correctly if: (i) the bed pole is touched within five minutes since plan activation (the time the means-end decomposed goal is activated), and (ii) the patient stands up—after he touches the bed pole—within ten minutes. There are three possible violations for this plan:

- the patient stands up without being in bed before. This might represent a fault in the camera that de-

Plan name	Precondition	Action	Time
t_1: use pole/trapeze	bed b has a pole pl	pl is touched	5
	patient p is in bed b, b has pole pl, pl is touched	p stands up	10
t_2: get up without support	**patient p is in bed b**	p stands up	10
t_7: use thermometer	patient p has thermometer t, **t is active**	t measures temp $> 35°C$	35
t_8: use oxymeter	patient p has oxymeter o, **o is working**	o measures heart rate	16
	patient p has oxymeter o, **o is working**	o has measured saturation	16
t_9: use glucose meter	patient p has glucose meter gm, **gm is turned on**	gm has blood on its sensor	28
	patient p has glucose meter gm, **gm is turned on**, **gm has blood on its sensor**	gm measures glucose level	30
t_{10}: use pulse checker	patient p has pulse checker c	c measured pulse	25
t_{11}: use smart shirt EKG	patient p has smart shirt s	s performs EKG	40
t_{12}: inject with insulin pen	patient p has insulin pen i	i injects insulin	50
t_{20}: phone helper to get up	house h has phone ph	ph is dialling	10
	house h has phone ph, **ph is not dialling**	ph called a helper	15
t_{24}: activate sling	bed b has sling sl, **sl is active** and over bed	patient p stands up	45

Table 1 Semi-formal specification for some plans in Fig. 2. Preconditions in bold are critical

tects when the patient stands up, or in the bed sensor that detects if the patient is in bed;
- a time-out failure occurs if the patient does not stand up within ten minutes (e.g. since the alarm rings). This happens if the patient touches the bed pole—his intention is to stand up—and he does not stand up within the time limit;
- the pole is not touched within five minutes. Both this violation and the previous indicate a possible health problem of the patient.

Based on the policies defined by designers, these failures might result in different actions: the system might notify a nurse and give her access to the room webcam, gently alert the patient, or ignore the failure if his vital signs are good. Though failures are detected at the level of tasks, they are often related to social dependencies between actors. Indeed, a dependee actor is expected to execute tasks and achieve goals to fulfil a dependum.

Domain assumptions. A domain assumption is an indicative property that should not be violated by the system or its surrounding environment [28]. Though not referring to functions the system should deliver, domain assumptions should be considered during design, as they express stakeholders' needs or constraints. Our architecture supports their monitoring and diagnosis.

We specify domain assumptions as implications over the context model. The implication antecedent consists of an activation event, that triggers the domain assumption, and an arbitrary number of preconditions. If the antecedent holds (the activation event occurs while the preconditions are true), the consequent has to be verified. We allow for two types of consequent: (i) a state should hold when the activation event occurs; and (ii) an event should occur within a time limit since the antecedent event occurs.

A possible domain assumption for the smart-home scenario is "The oven should be turned off within 10 minutes if the patient is not at home". This domain assumption is represented as follows: (i) the activation event is "the patient exits home"; (ii) the antecedent precondition is "the oven is turned on"; (iii) the consequent is event "the oven is turned off", which should occur within ten minutes. Another domain assumption is that "the fridge door should be closed if the patient is not in kitchen". This can be expressed as follows: (i) the activation event is "the patient exits kitchen"; (ii) the consequent is state "the fridge is closed".

4.2.2 Diagnosis component

The *Contextual goal model diagnosis* component uses context changes to analyse goal models for identifying goals and plans that should and should not be achieved. Its output is provided through interface *Goals/Plans applicability*. *Domain assumption verifier* uses context changes to verify the list of domain assumptions. Identified violations are then exposed through interface *Violated domain assumptions*. *Interaction analyser* uses both interaction status and goals/plans applicability, and computes failed dependencies. Dependencies fail not only if the dependee cannot achieve the goal or perform the plan (e.g. the nurse cannot support the patient because she's busy with another patient), but also when changes in the context affect applicable goals and make a dependency inapplicable (e.g., the patient exits her house and thus cannot depend on a catering service any more). *Plan execution checker* requires goals/plans applicability, determines failed plans and provides them through interface *Failed plans*. For example, this component can identify failures such as "insulin

was pumped while insulin pump was not under skin", or a time-out as the patient tries to wake up.

The *Goal commitment diagnosis* component diagnoses those goals that should be achieved for which the agent took no action so far. This notion of commitment corresponds to psychological commitments [5]: whether the agent has adopted an intention [15]. For example, the patient shall have breakfast within two hours since waking up (has to commit and adopt an intention to that purpose). This component requires goals/plans applicability and provides interface *Uncommitted goals*. A goal is uncommitted if there is no evidence that the agent is committed to its achievement. *Diagnoses Selector* requires failed dependencies, failed plans, uncommitted goals, and uses interface *Tolerance policies* provided by the *Policy manager*. The policy manager handles system administrators policies, e.g. cases where failures should be tolerated and not compensated. For example, lack of commitment for washing dishes can be tolerated if the patient's vital signs are good (she may wash dishes after her next meal). The interface *Failure diagnoses* contains the diagnoses to compensate.

4.3 Reconfiguration

The reconfiguration phase defines compensation strategies in response to diagnosed failures. Its effectiveness depends on various factors, among which available alternatives and the effectiveness of suggestions and reminders on participating actors. Also, the success of compensation strategies depends on available resources. Suppose a patient feels giddy: if she lives in a smart-home provided with a door opener, the door can be automatically opened to let the rescue team enter; else, the rescue team should wait for someone to bring the door keys. The component *Prioritize diagnoses* requires selected diagnoses and priority policies, selects a subset of failures according to their priority and provides them through interface *Diagnoses to compensate*.

Common criteria to define priorities are goal criticality, failure severity, urgency of taking a countermeasure, time passed since diagnosis. For example, a priority policy might specify that a life-critical goal (e.g. patient measures glucose level) shall be reconfigured before a non-critical goal (e.g. patient brushes her teeth). The component *Reaction strategy planner* takes the diagnoses to compensate as input and selects a set of reactions to compensate for the failures. Given one or more failures, this component identifies possible reconfigurations, and selects one of them. Our architecture supports three types of reconfiguration:

- *Task reassignment reconfigurations* involve the automated enactment of dependencies on agents. The architecture acts on behalf of the supported agent to define a social relation with a dependee agent. Task reassignments work only if the dependee accepts to deliver the service (due to its autonomy, it is free to refuse). For example, if the patient has not had breakfast and the time limit for the goal is expired, the architecture can automatically call the catering service. If the catering service does not accept or later violates its task, the architecture will have to perform another reconfiguration.
- *Push system reconfigurations* remind agents of their current goals or suggest tasks to execute. Such option gives little certainty, but is an effective way to deal with social actors. A push strategy for the patient that did not have breakfast so far is sending an SMS to her mobile phone.
- *Actuate reconfigurations* are enacted via context actuators. As we said earlier, context actuators are passive entities that can be commanded. For instance, if the patient feels bad, the door can be automatically opened by activating its door opener. This way, rescuers or neighbours can easily enter and help the patient.

Each reconfiguration type feeds a specific component (*Task assigner*, *System pushing* and *Actuator manager*) that enacts the chosen reconfiguration by interacting with external components (agents and actuators).

5 Diagnosis and reconfiguration mechanisms

We describe the main algorithms to diagnose failures and to compensate for them via reconfiguration. Algorithm 1 shows how the diagnosis and reconfiguration activities are linked together in the MDRC cycle. The input consists of the supported agents in the STS, whereas the output is a *possible* strategy to enact in order to better achieve the goals of the agents. The success of the suggested strategy largely depends on whether the involved agents will accept and enact it.

The *for* cycle of lines 2-7 iterates over all agents, whereas the internal for cycle of lines 3-7 scans all top-level goals agents are expected to achieve. Table 2 provides the semantics for goals to achieve (*should_do*) and achievable sub-goals (*visible*). Line 4 calls function DiagnoseGoal (described later in Algorithm 2) to determine which plans and goals are failed, started, and done. Then (lines 5-7) the global set of failed, started, and done plans/goals are updated. Lines 8 and 9 deal

[5] Singh [43] explains the distinction between psychological and social commitments.

Algorithm 1 Reconfiguration algorithm

```
RECONFIGURE(Agent [] ags)
 1  GoalPlan [] fail, start, done
 2  for each ag ∈ ags
 3  do for each g ∈ GETSHOULDDO(ag)
 4       do ⟨f, s, d⟩ ← DIAGNOSEGOAL(g)
 5            fail += f
 6            start += s
 7            done += d
 8  Strategy [] alt ← GENERATEALTERNATIVES(ags)
 9  Strategy str ← SELECTVARIANT(alt, fail, start, done)
10  ENACTSTRATEGY(str)
```

with the reconfiguration process: line 8 calls GENERATEALTERNATIVES (Algorithm 4), a function that acts as a planner and identifies alternative configurations; line 9 calls SELECTVARIANT (Algorithm 5), which ranks existing alternatives and selects the best one (according to a specific set of criteria). Finally, the selected strategy is enacted by ENACTSTRATEGY (line 10).

5.1 Diagnosis

We detail the diagnosis mechanisms included in our architecture. Failures are identified by comparing *monitored behaviour* of the system to the *expected behaviour*. Failures occur when (a) the monitored behaviour is not allowed or (b) expected behaviour does not occur.

$$\frac{goal(a,g,P) \wedge happened(activation\_evt(a,g,P),t) \wedge \neg done(a,g,P) \wedge \nexists\, g_p, dec\ s.t.\ decomp(a,g_p,g,dec)}{\textbf{should\_do(a,g,P)}} \quad \text{i.}$$

$$\frac{should\_do(a,g,P)}{\textbf{visible(a,g,P)}} \quad \text{ii.}$$

$$\frac{goal(a,g,P) \wedge \neg done(a,g,P) \wedge \exists\, g_p\ s.t.\ goal(a,g_p,P_p) \wedge decomp(a,g_p,g,dec) \wedge visible(a,g_p,P_p) \wedge holds(context\_cond(dec)) \wedge \forall p \in P\ s.t.\ (\exists\, p_p \in P_p\ s.t.\ name(p)=name(p_p)),\ value(p)=value(p_p)}{\textbf{visible(a,g,P)}} \quad \text{iii.}$$

$$\frac{plan(a,t,P) \wedge holds(prec(a,t,P)) \wedge \neg done(a,t,P) \wedge \exists\, g\ s.t.\ goal(a,g,P_p) \wedge means\_end(a,g,t,dec) \wedge holds(context\_cond(dec)) \wedge visible(a,g,P_p) \wedge \forall p \in P\ s.t.\ (\exists\, p_p \in P_p\ s.t.\ name(p)=name(p_p)),\ value(p)=value(p_p)}{\textbf{visible(a,t,P)}} \quad \text{iv.}$$

$$\frac{visible(a,t,P) \wedge depends(a,b,t,g) \wedge \neg done(a,t,P)}{\textbf{visible(b,g,P)}} \quad \text{v.}$$

Table 2 Expected and visible goals and plans

Table 2 introduces the *should_do* and *visible* predicates in first-order logic. Goal models define goals at the class level. At runtime, each goal can have multiple instances with different parameters. A patient's goal Have breakfast is repeated every day with different values for the parameter *day*, and has different instances for each patient living in the smart-home.

- Rule i. specifies when an agent a shall achieve a top-level goal. This happens if g is a goal instance with actual parameters P, is not achieved yet, the activation event occurred, and g is a top-level goal (there is no 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 visible (i.e., it is compatible with the current context).
- Rule iii. defines when decomposed goals are visible. A goal instance g with parameters P can be achieved if it is not achieved and exists an achievable goal g_p with parameters P_p that is decomposed into g, the context condition on the decomposition holds, and the actual parameters of g are compatible with the actual parameters of g_p.
- Rule iv. defines visibility for plans. This rule is very similar ro Rule iii. There are two differences: plans are also characterized by a precondition—which has to hold to make the plan executable—and are connected to goals through means-end decompositions.
- Rule v. specifies visibility for dependencies. If a task t is visible for agent a and a depends on b for goal g, and t is not completed yet, then g is visible for b.

Algorithm 2 (DIAGNOSEGOAL) uses the rules in Table 2 to diagnose goals and plans failures. It is invoked by Algorithm 1 for each top-level goal agents are expected to achieve, and explores a goal tree recursively.

Algorithm 2 starts by declaring arrays to contain failed, started and done goals/plans, also one array for the sub-goals or means-end decomposed tasks (*children*). Then (line 2) the status of goal g is set to `uncommitted`, since no information is initially available. Lines 3-10 define the recursive structure of the algorithm. If the goal is AND/OR-decomposed (line 3), the array *children* is initialized to contain all the sub-goals of g (line 4), and the function DIAGNOSEGOAL is recursively called for each sub-goal (lines 5-6), updating the goal arrays. If the status of all the sub-goals is `success`, then the status of g is also set to `success`, and g is added to the array `done` (lines 7-9). If the goal is means-end decomposed (line 10), G is assigned to the set of plans that are means to achieve the end g. If the diagnosed status of g is still uncommitted (line 11) each sub-goal (or means-end decomposed plan) in *children* is examined (lines 12-37). If g is not visible, the algorithm does not examine the goal further and contin-

Algorithm 2 Identification of goal and plan failures

```
DiagnoseGoal(GoalPlan g)
 1  GoalPlan [] fail, started, done, children
 2  g.status ← uncommitted
 3  if IsGoalDecomposed(g)
 4     then children ← Getsubgoals(g)
 5          for each g_i in children
 6          do ⟨fail, started, done⟩ += DiagnoseGoal(g_i)
 7          if ∀g_i in children, g_i.status = success
 8            then g.status ← success
 9                 done += {g}
10     else children ← GetMeansToEnd(g)
11  if g.status = uncommitted
12     then for each g_i in children
13          do if !g_i.visible
14                then continue
15             if IsAndDec(g)
16                then switch g_i.status
17                       case fail :
18                            g.status ← fail
19                            break
20                       case in_progress :
21                            g.status ← in_progress
22             if IsMeansEndDec(g) then
23                if DependsOn(g_i, g′) then
24                   g_i.status ← DiagnoseGoal(g')
25                else g_i.status ← CheckPlan(g_i,g)
26             if not IsAndDec(g)
27                then switch g_i.status
28                       case success :
29                            g.status ← success
30                            done += {g}
31                            break
32                       case in_progress :
33                            g.status ← in_progress
34                       case fail :
35                            fail += {g_i}
36                            if g.status = uncommitted
37                               then g.status ← fail
38  if g.status = in_progress then started += {g}
39  if g.should_do and g.status = uncommitted and
40     Timeout(g) then g.status ← fail
41  if g.status = fail then fail += {g}
42  return ⟨fail, started, done⟩
```

ues with the next element in *children*. Indeed, invisible goals cannot be achieved in the current context. If g is AND-decomposed (lines 15-21), two cases are handled: if g_i failed, then the status of is g is set to *fail* and no other element in *children* has to be examined (lines 17-19); else if g_i is in progress, the status of g is set to `in_progress` (lines 20-21). If g is means-end decomposed (line 22), two cases are possible. If the task originates a dependency on another agent for goal g' (line 23), the algorithm DiagnoseGoal is invoked recursively on g' (line 24). If the task originates no dependency, the algorithm calls CheckPlan (Algorithm 3) to diagnose plan status (line 25). If g is not AND-decomposed—i.e. it is either OR-decomposed or means-end decomposed—(lines 26-37), it succeeds if at least one sub-goal (or plan) succeeds. If the status of g_i is `success`, the status of g is also set to `success`, g is added to the set of done plans and the cycle is terminated (lines 28-31). If g_i is in progress, the status of g is set to `in_progress` (lines 32-33). 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` (lines 34-37). If the loop ends and the status of g is `in_progress`, g is added to the started goals (line 38). If g is a top-level goal to achieve, its status is `uncommitted`, and the time limit expired, then its status is set to `fail` because the agent adopted no intention for it (lines 39-40). If the status of g is `fail`, it is added to the list of failures (line 41). Finally, the algorithm returns failed, started and succeeded goals (line 42).

Algorithm 3 Plan execution diagnosis

```
CheckPlan(GoalPlan means, GoalPlan end)
 1  int start_time ← GetActivationTime(end)
 2  Event [] events ← GetSpecification(means)
 3  plan_status ← uncommitted
 4  for each evt in events
 5  do if ∃time1 > start_time s.t. Happened(evt, time1)
 6        then if time1 > start_time + evt.time_limit
 7                then return fail
 8             boolean done ← true
 9             for each prec in evt.preconditions
10             do if !HoldsAt(prec, time1)
11                   then if IsCritical(prec)
12                           then return fail
13                           else done ← false
14             if plan_status = uncommitted and done
15               then plan_status ← success
16        else if plan_status = success
17               then plan_status ← in_progress
18  return plan_status
```

Algorithm 3 (CheckPlan) describes plan diagnosis. Its parameters are a plan and a goal linked by a means-end decomposition. Lines 1-3 initialize the variables used by the algorithm: *start_time* contains the activation time of the goal, *events* is the set of events that specifies the plan, *plan_status* is the return value of the algorithm, and it is initially set to `uncommitted`. Then, all events in the specification are examined (*for* cycle in lines 4-17). If the examined event occurs after the activation of the goal (lines 5-15), the event is further analysed. If the event occurs after the event time limit, the algorithm returns failure (lines 6-7); otherwise, the event preconditions are examined (lines 8-13). Line 8 initializes the variable *done* to true. If the precondition does not hold (line 10) and it is critical (line 11) the algorithm returns failure (line 12), whereas if it is not critical the variable *done* is set to false. After preconditions are checked, if the plan is still uncommitted and

the variable *done* is still true (line 14), the plan status is set to success (line 15). If the event has not occurred after goal activation and the plan status is `success` (line 16), the plan status is set to `in_progress`, since not all events have happened. Line 18 returns the plan status if the algorithm has not returned failures in the cycle.

Domain assumptions are diagnosed when their triggering event (the antecedent) happens and its preconditions are true. Their failure is determined depending on their consequent type:

- *Event:* a failure occurs if the event does not happen within its time limit. This kind of violation is detected as the consequent event time limit expires;
- *State:* a failure occurs if the consequent state does not hold at the time the triggering event happens.

5.2 Reconfiguration algorithms

As shown in the *Reconfiguration* component in Fig. 3, our reconfiguration mechanisms include both reconciliation (planning) and compensation (enactment). While enactment is a domain-specific activity, which depends on the environment and participating agents, planning mechanisms can be abstractly defined regardless of the setting where the architecture will operate. Our mechanisms are inspired by Artificial Intelligence (AI) planning techniques, though we address a simpler problem. Indeed, our problem can be reduced to searching all sets of leaf nodes that support the root node in a set of AND/OR trees. This is doable by identifying all root-to-leaf paths in each tree. As the output of this activity is a set, differently from traditional AI planning, we are not concerned with the ordering of leaf nodes (tasks).

When planning for new configurations, the first decision to make is whether planning should be local (from a single agent's perspective) or global (from the perspective of the overall STS). With respect to the notion of agency, local planning is better, for global planning requires a centralized view that conflicts with the autonomy of the agents. Global planning is better in terms of planning efficacy, since it can find a solution that is the best for the entire system. The computational cost is lower in local planning, since this approach avoids loops that can arise from mutual dependencies. We combine the two approaches: we take a global perspective as our planning suggests dependees to adopt a commitment for the dependums, while we evaluate locally the best plan for goal achievement. Given the autonomy of each agent, our architecture does not force them to execute tasks or to achieve goals. The architecture can do no better than *trying* to make agents committed, but there can be no guarantee that they will carry out the suggested plan.

There are three main planning mechanisms to identify alternative variants in goal models in response to a specific goal or plan failure:

- *Backtracking*: the goal tree is explored bottom-up in order to find an alternative plan. Options in the same tree branch are preferred to options in other branches. Backtracking preserves stability—it ensures that the new variant differs as little as possible from the current one—but does not guarantee the optimization of cross-cutting concerns, such as soft-goals. This approach was experimented in related work [16], where this technique was applied by extending the agent-oriented programming language Jason [6]. Backtracking for failure handling is extensively discussed by Sardina and Padgham [42].
- *Tree planning*: this class of mechanisms corresponds to planning from scratch. Tree planning identifies the best solution in the goal tree (based on the metric used to determine the quality of a solution), regardless of how much the new variant differs from the previous one.
- *Tree replanning*: this approach combines tree planning and variant stability. Algorithms in this category do not only consider plan optimality, but also minimize changes from the previous configuration. The impact of each of these two factors (optimality and stability) can be balanced depending on the agent's preferences.

We describe a replanning algorithm that evaluates plan optimality on the basis of soft-goals contributions. Reconfiguration is enacted by Algorithms 4 and 5, that are invoked by Algorithm 1 in lines 8 and 9, respectively.

Algorithm 4 Alternatives generation

```
GenerateAlternatives(Agent [] ags)
 1  PlanSet [][] options ← null
 2  for each ag ∈ ags
 3  do for each g ∈ GetShouldDo(ag)
 4      do PlanSet [] opts ← GeneratePlans(g)
 5          options.Add(opts)
 6  PlanSet [][] cartProduct ← options[1] × ... × options[n]
 7  Strategy [] strategies ← null
 8  for option ∈ cartProduct
 9  do Strategy s
10      for planSet ∈ option
11      do s.AddPlanSet(planSet)
12      strategies.Add(s)
13  return strategies
```

Algorithm 4 takes as input the supported agents in the STS. Line 1 initializes to `null` the variable *options*

that will contain the alternatives for top-level goals. *options* is an array of arrays of type `PlanSet`. One planset is a viable strategy to achieve a top-level goal, an array of plansets contains all strategies to achieve a top-level goal, an array of arrays of plansets contains all strategies to achieve a set of top-level goals. The *for* cycle in lines 2-5 explores all agents, the inner cycle in lines 3-5 explores all top-level goals to be achieved, whereas lines 4-5 initialize the variable *options*. Line 4 creates an array of plansets that contains all plans for a specific top-level goal. The invoked function GENERATEPLANS explores the AND/OR goal tree and identifies all valid minimal plans. Minimality means that whenever an OR-decomposition is encountered, only one sub-goal is selected. In line 5, the algorithm adds the possible plans for the examined top-level goal to variable *options*. As the cycles are over, the variable *cartProduct* is initialized to the Cartesian product of the sets in *options*; this way, each array of plansets in *cartProduct* is a plan to achieve all top-level goals. Line 7 initializes the variable *strategies* to null; this variable is a data structure that allows for simpler handling of alternative than *cartProduct*. Lines 8-12 populate *strategies*: for each option (line 8) a `Strategy` is declared, the plansets in that option are added to the strategy (lines 10-11), and the strategy is added to *strategies* (line 12). Once the array of strategies is populated, the algorithm returns it (line 13).

Algorithm 5 Selection of the best reconfiguration strategy

```
SELECTVARIANT(Strategy[] alt, Plan[] failed, started, done)
 1  for each str in alt
 2  do str.cost ← Σ_{p∈failed} p.compCost
 3      for each sp in started
 4      do if p in GETALLPLANS(str)
 5            then str.cost += p.compCost
 6      for each gt in str.goalTrees
 7      do for each pl in gt.plans
 8          do int contrib, totPriority ← 0
 9              for each c in pl.contribs
10              do contrib += c.val * c.softgoal.priority
11                  totPriority += c.softgoal.priority
12              contrib ← contrib/totPriority
13              str.contrib += contrib/SIZE(gt.plans)
14              if pl not in started ∪ done
15                then str.cost += GETBESTREACTION(pl)
16  minCost ← min(str.cost | str ∈ alt)
17  maxCost ← max(str.cost | str ∈ alt)
18  minCont ← min(str.contrib | str ∈ alt)
19  maxCont ← max(str.contrib | str ∈ alt)
20  bestVal ← −∞
21  bestStrategy ← null
22  for each str in alt
23  do nCost ← (str.cost − minCost)/(maxCost − minCost)
24      nCont ← (str.contrib − minCont)/(maxCont − minCont)
25      if GETALLPLANS(str) ∩ failed ≠ ∅
26        then strValue ← strValue − 4
27      strValue ← strValue + nCont − nCost
28      if strValue > bestVal
29        then bestVal ← strValue
30             bestStrategy ← str
31  return bestStrategy
```

Algorithm 5 takes as input the array of strategies *alt* and three arrays of plans with self-explanatory names: *failed*, *started*, and *done*. The *for* cycle in lines 1-15 computes the cost and the contribution to soft-goals for each strategy in *alt*. Lines 2-5 determine the cost of the examined strategy, which corresponds to the sum of the compensation costs for failed plans (we assume that each failed plan has to be compensated) and the compensation cost for started plans that will not be in the new configuration. Lines 6-15 iterate the goal trees in the strategy and compute contribution to soft-goals. Line 7 iterates all plans in the goal tree, line 8 initializes the variables *contrib* and *totPriority* used in the cycle, whereas line 9 iterates all the soft-goals the examined plan contributes to. In line 10, the contribution value from the plan to the soft-goal is multiplied by the soft-goal priority and the product is added to variable *contrib*. In line 11, the soft-goal priority is added to the total priority for the considered plan. Then (line 12) the plan contribution is divided by the total priority to compute the average soft-goal contribution for the examined plan, and the contribution value for the whole strategy is updated (line 13). Lines 14-15 update the reconfiguration cost by adding the minimum reaction cost for the plans that are not started nor done (GETBESTREACTION) if more than one reaction is available (e.g., both task assignment and system pushing are viable). Lines 16-19 initialize the variables that contain the minimum/maximum strategy cost (16-17) and the minimum/maximum contribution (18-19). Lines 20-21 initialize the variable to contain the best value to $-\infty$ and the best strategy to null. The *for* cycle in lines 22-30 performs a min-max normalization of costs and contribution in the $[-1, +1]$ range, in order to make them comparable and to compute a single value for the strategy. Lines 23 and 24 compute the normalized value for a strategy's cost and contribution, respectively. Line 25 deals with strategies that contain failed plans: any of these strategies is worse than all those without failed plans. The strategy value (line 27) is the normalized contribution minus the normalized cost. Lines 28-30 update the variable for the best strategy if the examined strategy is better than all those examined earlier. Line 31 returns the best strategy.

6 Implementation and evaluation

Our architecture falls into the paradigm of design science. Thus, its evaluation can be based on the methods suggested by Hevner *et al.* [25]. A comprehensive evaluation has to consider five perspectives: observational, analytical, experimental, testing, and descriptive.

We aim to evaluate the feasibility, applicability and effectiveness of our approach. To this extent, we developed a prototype implementation of our architecture (described in Section 6.1) and tested it in a simulated smart-home STS based on the case study in Section 3.

We detail on the application to the case study in Section 6.2; we exploit a descriptive and experimental evaluation approach, since we construct scenarios the architecture has to successfully cope with and perform simulations to experiment whether it is actually the case. We report on scalability results in Section 6.3; this type of evaluation belongs to dynamic analysis, as we aim to assess the performance of our artefact.

The main limitation of our evaluation is that we haven't conducted a case study in a business environment yet. Thus, we cannot evaluate the interaction of users with the architecture: (i) how well designers use the interfaces to instrument an STS with the adaptive architecture; (ii) the impact of the adaptation mechanisms on end users in terms of unobtrusiveness and efficacy. We will conduct these evaluations in future work.

6.1 Implementation

We developed a prototype implementation of our architecture to show its feasibility. We used Java 1.6 as main programming language, the Eclipse Modeling Framework (EMF)[6] to define the requirements meta-models, the DLV-complex reasoner [10] to support diagnosis and reconfiguration, and the H2 embedded database[7] to consider the effect of context changes on active goals.

EMF enables to define the meta-model for requirements models as a class diagram with OCL constraints. The class diagram provides the coarse-grained structuring of requirements models, whereas OCL constraints restrict the allowed syntax. For instance, in our EMF meta-model, a `Contribution` is represented as a class. Such class has an attribute called *value*, whose range is between -1 (full negative) and +1 (full positive), which represents the contribution degree. Class `Plan` is linked to `Contribution` via an aggregation link: in our framework, contributions start from plans. A contribution is linked—via an association—to a `SoftGoal` class: a softgoal is contributed by zero or more contributions, but can exist without any contribution. A contribution has a set (possibly empty) of `Conditions`: the contribution is valid only if the contextual conditions hold. Using the OCL constraint shown in Code 1 (an invariant), we enrich the specification of a contribution. The OCL invariant says that the contextual conditions of a contribution (if any exists) should refer to entities that belong to the parameters of the plan from which the contribution starts. This is a well-formedness constraint: it guarantees that contextual conditions in the contextual goal model refer to entities that can be correctly associated to the specific goal instance at runtime. For example, if a contextual contribution depends on the temperature of a room, but the decomposed goal has no parameter associated to a "room" entity, it will not be possible to guess which room should be considered.

Code 1 An OCL invariant applied to a contribution

```
context Contribution inv:
self.conditions->forAll(x | let y:ContextEntity =
  x.oclAsType(Condition).entity in
    self.contribFrom.parameters->exists(z |
      z.oclAsType(Parameter).entity=y))
```

Our prototype uses Java Emitter Templates (as in [17]) to transform the EMF meta-model—in an *ecore* file—into a requirements model editor deployable as an Eclipse application. The resulting application—a generated EMF editor—enables the definition of requirements models, and provides integrated validation to check the well-formedness of the model with respect to the meta-model and its OCL constraints. The analyst creates requirements models using the Eclipse default tree editor for XML files, as shown in Fig. 6.

At runtime, requirements models become integral part of the architecture for self-reconfiguration. Before the architecture starts, the analyst specifies the model to load, and the architecture translates it to the DLV-complex input format. Requirements models in the *ecore* file are expressed at the class-level, whereas the translation to Datalog supports instances, namely real agents and goal instances that constitute the application at runtime. Code 2 contains an example of how the prototype maps a contextual contribution to the DLV-complex input format. The first rule says that, if task useThermometer is visible according to the current context, and the patient has no flu, the contribution to soft-goal reliability is positive with value +0.7 (represented as 700, as DLV supports only integer numbers). The DLV internal predicate `#memberNth` is used to extract specific parameters from the parameters list P.

[6] http://www.eclipse.org/modeling/emf/
[7] http://www.h2database.com/

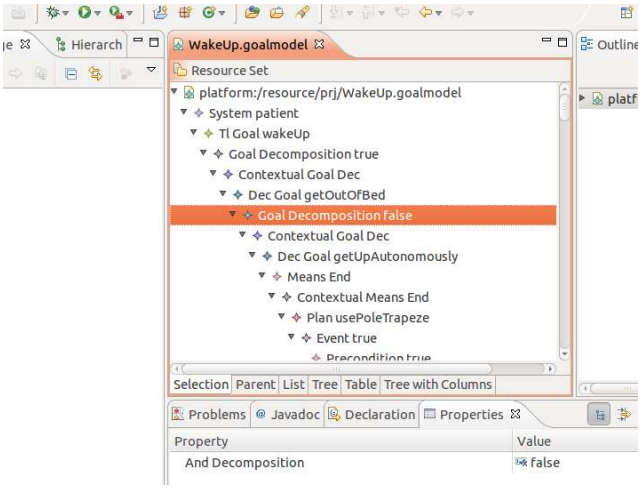


Fig. 6 Screenshot of the Eclipse-based requirements editor

The second rule states that, if the patient has flu, the contribution to soft-goal reliability is negative (−0.4).

Code 2 Representation of a plan to soft-goal contextual contribution in DLV-complex

```
contr(patient,useThermometer,P,reliability,Val,Neg)
:- visible(patient,useThermometer,P),
   #memberNth(P,1,Patient), #memberNth(P,2,House),
   #memberNth(P,3,ActTime),
   holdsNow(hasFlu,Patient,false,CT), currTime(CT)
   Val=700, Neg=false.
contr(patient,useThermometer,P,reliability,Val,Neg)
:- visible(patient,useThermometer,P),
   #memberNth(P,1,Patient), #memberNth(P,2,House),
   #memberNth(P,3,ActTime),
   holdsNow(hasFlu,Patient,true,CT), currTime(CT)
   Val=400, Neg=true.
```

In addition to requirements models, the analyst can optionally specify an input simulation trace for the architecture. After the analyst launches the architecture, the input of the architecture consists of both the simulation trace and the events received from the running application (agents and sensors). We developed an editor to enable designers to specify simulation traces. This editor permits to define the contextual entity instances that are considered at runtime, and provides a visual tool to specify the initial state and the evolution of those entities (by manipulating their attributes). The simulation is defined in terms of time steps; each tick occurs every n milliseconds. In our prototype, input from agents and sensors is read via socket listeners. The actual monitors that observe contextual events are domain-specific, as well as the normalization of events to the context model. In our experimentations, our prototype receives events that are already normalized over the context model.

Failures are handled in two steps. First, the architecture uses the DLV-complex reasoner as a planner to generate alternatives. Second, a dedicated thread selects the best option and enacts it. Enactment is a domain specific activity, which depends on available effectors and the capabilities of the agents in the system. We have chosen DLV-complex, rather than an off-the-shelf AI planner, for two reasons: (i) DLV-complex can be easily customized and extended; (ii) our alternative generation problem is simpler than the traditional planning problem, as it has a smaller search space (see Section 5.2).

6.2 Experiments on the case study

We experimented our architecture on the case study in Fig. 2. The goal model is composed of two agents, 25 goals, 28 plans, and 3 soft-goals. The context model consists of 18 entities having 67 attributes. Overall, 67 different event types are relevant to our architecture, such as “phone is dialling”, “bed rails are active”, “patient enters bathroom”, “patient exits home”. We ran our experiments on a machine equipped with an AMD Athlon(tm) 64 X2 Dual Core Processor 4200+ processor, 2GB RAM, and running Linux ubuntu 2.6.31-16-generic ♯53-Ubuntu SMP i686, Java OpenJDK Runtime Environment (IcedTea6 1.6.1) (6b16-1.6.1-3ubuntu1).

We tested the requirements models over several realistic scenarios. We show here three simulated scenarios that involve failure diagnosis and compensation. We express these scenarios in terms of time-steps. To automate testing, we developed a module that simulates the enactment of a chosen reconfiguration. The purpose of these scenarios is to demonstrate the capability of our architecture to detect and to react (by selecting a proper alternative configuration) to diverse failures that threaten the actors’ requirements: plan execution implying dependency failure, plan time-out, and goal time-out. An orthogonal problem, that our architecture does not deal with, is how to respond to device failures, e.g. how to ensure faulty devices will be fixed.

Scenario 1 *Marco is a diabetic patient that lives in a smart-home. Due to a recent flu, he needs assistance to stand up. His alarm rings at time-step 1. His phone starts dialling at time-step 2. At time-step 3 the phone stops dialling and a helper is called. Plan* t_{20} *is carried out without failure: Marco successfully calls a helper. This indicates that Marco depends on a helper to stand up. At time-step 4 he is not alone in his house any more. The sling is over his bed. At time 9, Marco stands up with the support of a helper.*

The event sequence described in Scenario 1 leads to a failure. Specifically, plan t_{24} fails. This, in turn, results in a failure of the social dependency on the helper. The patient is standing and the sling is over the bed, but the sling is not active. According to the monitored events, Marco could hardly be standing up, for the sling is above the bed and inactive. This might have different interpretations: either the patient is actually standing and the sling activation sensor is faulty, the patient standing sensor failed, or maybe the assistant is not there. Regardless of the real cause, the architecture plans possible alternatives and selects the best one, which includes notifying the helper to lift the patient. This corresponds to reminding the helper that the patient is depending on him. The system acts unobtrusively, as its intervention is invisible to Marco. If Marco does not achieve his other goals, more obtrusive actions will be taken.

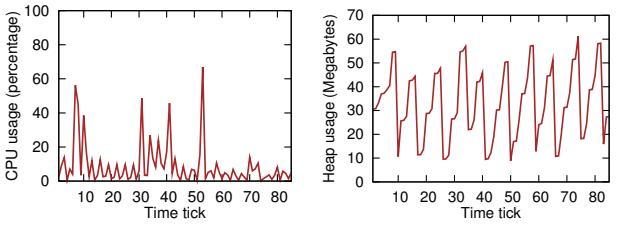
Scenario 2 *Marco knows a social worker named Mike. Marco is in bed, he is alone at home. At time-step 1 the alarm rings in his bedroom. Since Marco is still weak, he should call a helper and perform his pre-breakfast morning routine:* g_{18} *and* g_1 *are instantiated. At time-step 2 the phone dials; this gives evidence that Marco is calling a helper, maybe Mike. At time-step 17 no helper has been called yet, and the phone is still dialling.*

The architecture detects a plan time-out failure at time 17. Indeed, plan t_{20} is not performed correctly (see Table 1). While the first action is correctly performed at time-step 2 (the phone is dialling), the second action is not completed within the time limit: no helper is called. This might be a dangerous situation: perhaps Marco felt giddy and could not contact any helper. In reaction to this failure, the architecture decides to automate plan t_{21} and sends an SMS to Mike on behalf of Marco. Later, Mike acknowledges he received the text by sending an SMS to Marco's mobile. The smart-home interprets this message as the fact a helper has been called, and a dependency is established.

Scenario 3 *Marco has to perform his regular health check (he is expected to achieve* g_6*). At time-step 10 the oxymeter measures both heart rate and saturation. The actions are carried out correctly, for their critical preconditions are true. However, at time 61 the medical centre has not received any electronic medical report.*

This scenario includes a goal time-out failure for the top level goal g_1: Pre-breakfast morning routine. Indeed, at least one of its sub-goals is not achieved within one hour. The scenario does not satisfy the declarative goal g_6: Routine check, for its achievement condition is not met. The successful execution of task t_8: use oxymeter does not suffice to achieve g_6. The architecture detects this kind of failure as the time limit expires, and reacts by selecting a different plan, which involves automating task t_6: use smart-shirt, if Marco is wearing his smart-shirt.

Fig. 7 shows a snapshot of the architecture running Scenario 1. The current simulation trace is on the left side, whereas requirements monitoring is on the right side. Goal models are represented as a tree, the status of each goal is represented by a coloured circle. Plans are leaf-level nodes in goal trees. The status of domain assumptions is shown in a different tab (hidden in Fig. 7). In Fig. 7, the architecture has started its reaction to t_{24} failure: Marco is lifted by the helper, the smart shirt performs a routine check, Marco is using the glucose meter to measure his blood, he has taken his medicine.



	Average	Maximum	Minimum
CPU	8.93%	66.9%	0%
Heap	32.73MB	61.19MB	8.84MB
Diagnosis	863ms	1215ms	787ms
Reconfiguration	203ms	203ms	203ms

Fig. 8 The line charts show CPU and Heap Memory usage for Scenario 1. The table below shows statistics about CPU, Heap, diagnosis time, and reconfiguration time

Fig. 8 outlines the performance of our architecture when executed on Scenario 1. During this simulation, the prototype performed well during diagnosis and reconfiguration. On average, CPU usage was below 9%, with a maximum value of almost 67% and some other peaks, but also some values close to 0%. Memory (heap) usage follows a pattern where heap allocation (high peaks are ∼61MB) is immediately followed by heap deallocation (low peaks are ∼9MB). On average, heap usage is less than 33MB. For what concerns performance, we can conclude that the size of the models used for this case study is not critical for our prototype. On average, diagnosis took 863ms, with a maximum of 1215ms and a minimum of 787ms. Planning for and selecting alternatives took 203ms. Future work will involve a more comprehensive evaluation, e.g. via random generation of scenarios to use as test cases.

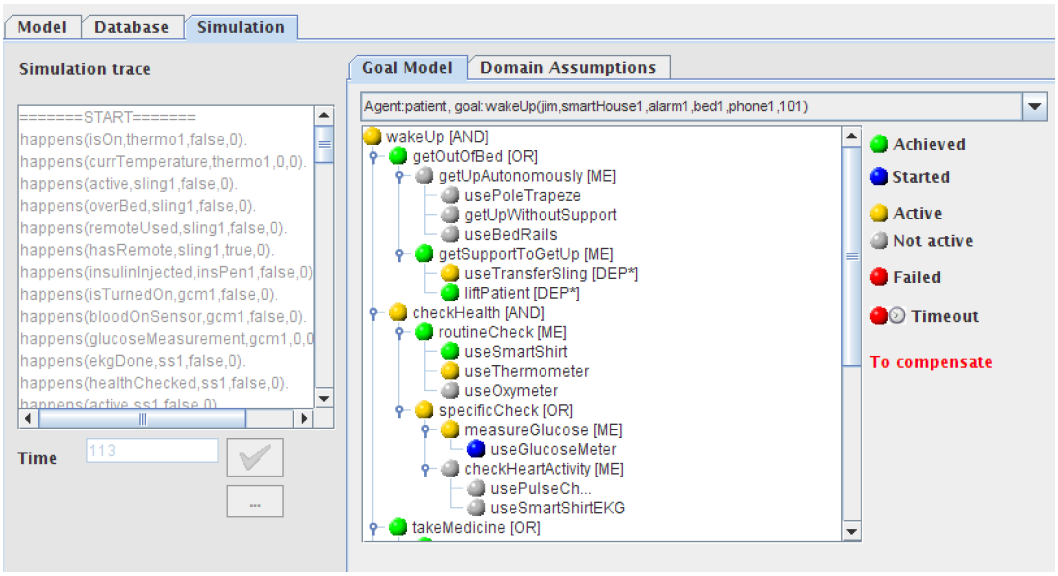


Fig. 7 Requirements monitoring GUI of the architecture applied to the smart-home case study

6.3 Scalability

We evaluate scalability along two dimensions: (i) failure diagnosis, i.e. how long the architecture takes to determine failures (Algorithms 2 and 3); (ii) system reconfiguration, i.e. the time needed to derive alternatives (Algorithm 4) and to choose the best one (Algorithm 5).

Diagnosis scalability is verified on goal models of growing size. Specifically, we increase: (i) the number of top-level goals of an agent; (ii) the number of agents in the model; (iii) the depth of a goal tree. We report results concerning all these dimensions.

Trees	Goals	Rules	Time	$\frac{Time}{Goals}$	$\frac{Time}{Rules}$
1	15	131	218	14.533	1.664
2	30	255	318	10.600	1.247
3	45	379	526	11.689	1.387
6	90	751	799	8.878	1.064
12	180	1495	2146	11.922	1.435
24	360	2983	7101	19.725	2.380
48	720	5959	25615	35.576	4.299

Table 3 Diagnosis scalability: increasing the number of top-level goals. Time in ms

Table 3 presents scalability results obtained by increasing the number of top-level goals. We take a basic goal tree composed of 15 goals and clone it to obtain multiple goal trees, thus increasing the total number of goals. The six columns in the table represent the number of goal trees, the number of goals, the number of datalog rules, the diagnosis time, the time per goal, and the time per rule, respectively. The results show that the tool scales very well till 180 goals (12 top-level goals), for the time per goal is always below 15ms. With larger goal models the time per goal increases to 35ms for 720 goals. However, this is still a good result given that (i) time does not grow exponentially; (ii) typical goal models for applications such as the smart-home STS are smaller than the largest we tested.

Agents	Goals	Rules	Time	$\frac{Time}{Goals}$	$\frac{Time}{Rules}$
2	20	291	213	14.550	1.366
3	25	320	291	12.800	1.099
5	40	340	379	8.500	0.897
11	70	463	525	6.614	0.882
21	120	793	915	6.608	0.867
41	220	1964	1695	8.927	1.159
81	420	5870	3255	13.976	1.803
161	820	17831	6375	21.745	2.797

Table 4 Diagnosis scalability: increasing the number of agents. Time in ms

Table 4 reports on scalability for requirements models with multiple agents. We keep one agent unchanged and make it depend on an increasing number of other agents. The agents acting as dependee are cloned: each agent has one small goal tree composed of five goals. The table columns represent the number of agents, the number of goals, the number of rules, the diagnosis time, the time per goal and the time per rule, respectively. Diagnosis scales linearly till 420 goals (81 agents); the result becomes slightly worse with 820 goals (161 agents). However, the growth is still not exponential.

Table 5 details scalability results for goal models with increasing depth. To increase depth, we generated goal decompositions with just one sub-goal. In this setting, the diagnosis mechanisms scale less well than in the other two experiments: good scalability is measured

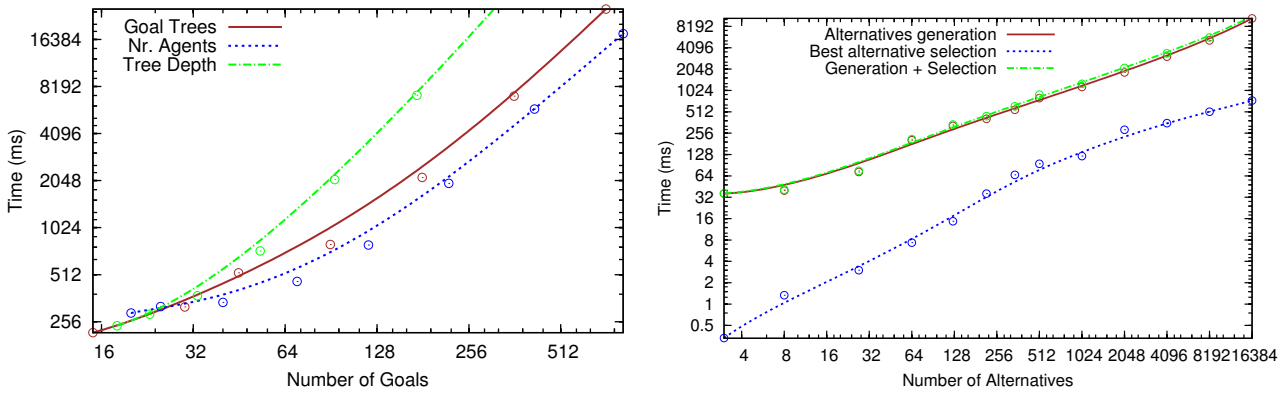


Fig. 9 Scalability evaluation for diagnosis (left side) and reconfiguration (right side) mechanisms

Depth	Goals	Rules	Time	$\frac{Time}{Goals}$	$\frac{Time}{Rules}$
5	18	241	156	13.388	1.545
10	23	286	191	12.435	1.497
20	33	375	261	11.364	1.437
40	53	727	401	13.717	1.813
80	93	2076	681	22.323	3.048
160	173	7184	1241	41.526	5.789
320	333	30071	2361	90.333	12.737

Table 5 Diagnosis scalability: increasing goal model's depth. Time in ms

till 93 goals (depth 80). However, these are unrealistically deep goal models.

The left-hand side of Fig. 9 shows a chart plot summarizing diagnosis scalability. The chart uses a logarithmic scale on both axes. The three plots represent the three conducted experiments: the chart shows that diagnosis works better with many small goal models (even with many agents) than with a single large goal model (increasing its depth). Overall, the diagnosis mechanisms perform well for medium-sized requirements models.

To assess reconfiguration scalability, increasing the number of goals—as we did to verify diagnosis—is not meaningful. Reconfiguration is the selection of an alternative configuration to achieve current goals. Therefore, reconfiguration corresponds to (i) generating possible alternatives and (ii) selecting the best alternative. Scalability is measured with respect to the number of alternatives (variants). To check reconfiguration scalability we create goal models with increasing number of alternatives. We increase the number of options at variation points: more sub-goals in OR-decompositions and more plans in means-end decompositions.

Table 6 reports on scalability for reconfiguration mechanisms. The five columns represent the number of alternatives, the time taken to generate alternatives, the time taken to select the best alternative, the generation time per alternative, and the selection time per alternative, respectively. We repeated each test three times; the overall time is approximated to the millisecond in columns 2 and 3. The implemented reconfiguration mechanisms scale very well. The growth is linear with the number of alternatives, as can be seen in the ratio columns. Both generation and selection perform efficiently. In particular, selection time is much lower than generation time. The right-hand side of Fig. 9 graphically resumes the scalability results for reconfiguration, also shows the overall reconfiguration time.

♯ alt	Gen time	Sel time	$\frac{Gen\ time}{\sharp alt}$	$\frac{Sel\ time}{\sharp alt}$
3	36	<1	12.000	0.111
8	40	1	4.958	0.166
27	73	3	2.691	0.111
64	204	7	3.182	0.115
125	325	15	2.597	0.117
216	408	36	1.867	0.166
343	547	66	1.594	0.193
512	798	95	1.559	0.186
1024	1148	122	1.121	0.119
2048	1839	286	0.898	0.140
4096	3064	354	0.747	0.111
8192	5198	514	0.635	0.063
16384	10604	738	0.647	0.045

Table 6 Reconfiguration scalability: increasing the number of alternatives. Time is expressed in ms

7 Creating the architecture for an existing STS

To exploit our architecture, requirements engineers and software designers need guidance to apply it to existing STSs. Once created, the architecture can be deployed to add self-reconfiguration capabilities to the STS by helping participating agents achieve their objectives.

Fig. 10 shows the proposed process as a SPEM 2 [34] diagram representing the sequence of activities and the input-output flow in terms of artefacts. Specific methodologies shall be adopted or devised to refine the

process according to the application domain. For example, requirements elicitation and analysis methods are required to correctly gather and specify requirements.

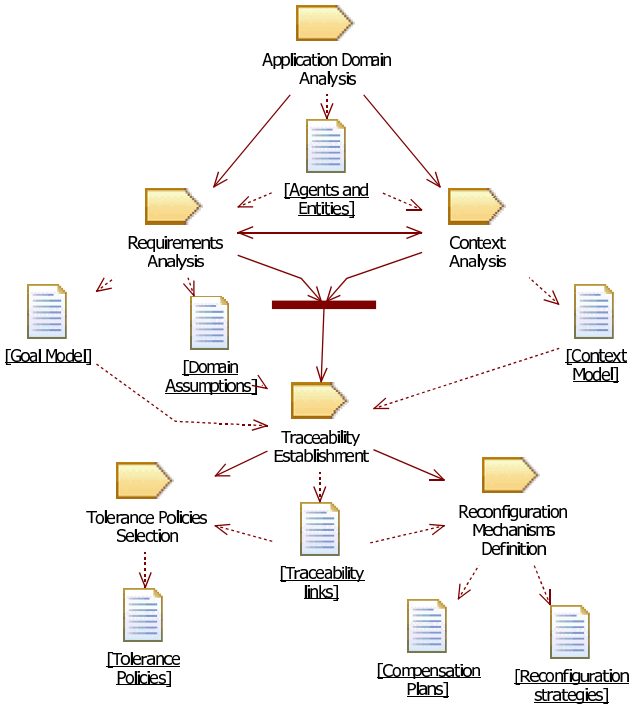


Fig. 10 SPEM 2 diagram showing the high-level process to create the architecture for an STS

The first task is *Application Domain Analysis*: acquiring knowledge about the STS in terms of humans, autonomous software agents, and non-autonomous entities such as sensors and actuators. The resulting artefact is the set of agents and entities in the application.

Domain analysis is followed by two concurrent tasks: *Requirements Analysis* and *Context Analysis*. Their output consists of the models for the STS: a context model results from context analysis, while goal models and domain assumptions are provided by requirements analysis. These two steps are typically performed iteratively: the corresponding models are not isolated, given that requirements models are specified on the basis of context. For example, domain assumptions are implication over context entities, whereas the status of goals (visible, started, done, failed) is computed on the basis of contextual events. Given the tight connection between these models, changes in the context (requirements) model often requires to modify to the requirements (context) model. The interested reader might look at recent work on contextual goal-oriented requirements engineering [2].

After requirements and context analysis are completed, *Traceability Establishment* is conducted. This task defines what to monitor at runtime for determining requirements satisfaction and violation. The objective of this step is to ensure traceability links between implementation and requirements. In particular, this traceability corresponds to requirements reflection [5], a system's awareness of its requirements. We propose to carry out this step by associating a specification—in terms of observable events expressed over the context model—to goal model tasks. Thus, the status of a task is computed by monitoring the status of its event-based specification, whereas the status of a goal is derived from the status of the tasks that operationalize it.

The following steps are to select tolerance policies and to define reconfiguration mechanisms. Task *Tolerance Policies Selection* specifies tolerance policies for failures and under-performance. Some failures are addressed through reconfiguration, others are always tolerated, others again are tolerated under certain circumstances. Policies are first defined at design-time, then they are typically tuned at runtime. For instance, system administrators might realize that too many failures are considered, and system performance is affected by frequent reconfigurations.

Task *Reconfiguration Mechanisms Selection* defines how failures are addressed by the architecture. The activity produces two artefacts: (i) *Compensation Plans* revert the effects of the failed strategies, and (ii) *Reconfiguration Strategies* describe possible alternatives to achieve current goals. Both steps depend on the actuation capabilities of the existing application. Possible reactions are scenario-dependent: the architecture controls actuators or communicates with agents.

8 Related work

Most research in architectures for adaptive and autonomic systems does not support socio-technical systems where participating agents are autonomous and largely uncontrollable. Some proposals consider these features to some extent. Bryl and Giorgini recognize the dynamism of STSs and propose an approach for the redesign of an STS in response to changes such as actors joining or leaving the system [9]. Their approach inspires our architecture: it states the problem and provides general principles, but these are not applied to an architecture for self-reconfiguration. Cetina *et al.* propose an architecture for autonomic computing and apply it to smart-homes [12]. Unlike ours, their architecture is based on feature models and focuses mainly on the technological aspects of an STS. The two approaches might be combined into an architecture that relies on both high-level concepts—requirements—and technical configuration concerns—features.

Wang's architecture for self-repairing software [46] uses a single goal model as a software requirements model, and exploits SAT solvers to check the current execution log against the model to diagnose plan failures. We propose a broader approach, adopting part of a more comprehensive ontology for RE [28]. We use more expressive goal models, provide detailed specification for plans, allow for specifying multiple contexts that modify expected behaviour, and support dependencies on other actors/systems.

Feather *et al.* propose an approach addressing system behaviour deviations from requirements specifications [21]; they introduce an architecture (and a development process) to reconcile requirements with behaviour. The reconciliation process is enacted by jointly anticipating deviations at specification time and solving unpredicted situations at runtime. Our architecture uses different requirements models, supports a wider set of failures (theirs is focused on obstacle analysis [32]), and deals with interacting autonomous agents.

Baresi and Pasquale [3] propose an approach based on extended KAOS goal models for adaptive service compositions. They introduce the notion of adaptive goals, which are responsible for the actual adaptation and evolution at runtime, and specify countermeasures to address violations of conventional goals. Their framework is further extended by FLAGS (Fuzzy Live Adaptive Goals for Self-adaptive systems) [4], which distinguishes between crisp goals with boolean satisfaction value and fuzzy goals whose satisfaction is specified via fuzzy constraints. Fuzzy goals exploit a temporal language inspired by the theory of fuzzy sets. Though supporting complex goal types, this approach does not consider social relationships between actors in an STS.

Whittle *et al.* [48] propose RELAX, a requirements specification language for self-adaptive systems. RELAX relies on requirements relaxation to ensure that the requirements handle uncertainty factors. Such language can be used as an alternative requirements specification language to deal with uncertainty during adaptivity. However, RELAX does not exploit any social abstraction, and is therefore more adequate for purely technical systems.

ReqMon [41] is a requirements monitoring framework for enterprise systems. ReqMon integrates KAOS requirements models and software execution monitoring, and provides tools to support the development of requirements monitors. Though ReqMon's architecture covers all the reconfiguration process, it details only the monitoring and analysis phases. Our approach targets different application settings, can diagnose a different set of failure types, and is equipped with implemented reconfigurations mechanisms.

An important aspect to consider during adaptation is customization to specific users. Hui *et al.* [27] investigate how a goal model can be customized to user skills and preferences. The selection between alternatives is based on the user skills to execute atomic tasks and on user preferences (which are expressed over soft-goals). Their approach can be combined with our architecture so that to take into account the profile of specific users while reconfiguring the STS.

Though not focused on STSs, literature in adaptive software proposes many relevant architectures and algorithms. Self-adaptive software [36] is an approach to develop systems that modify their architecture in response to changes in the operating environment. This is a model-based approach in which the model is the architecture itself. Basic reconfigurations consist of adding, removing and replacing components. The building units for self-adaptive software are components and connectors. Compared to our work, this solution exploits architectural knowledge rather than requirements knowledge, thereby providing no assurance about the satisfaction of the system's goals. Moreover, their approach presumes full control of the system, which is not possible in STSs.

Rainbow [24] is an architecture for self-adaptation based on an externalized approach and software architecture models. Rainbow considers architecture models as the most suitable abstraction level to abstract away unnecessary 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 the externalized approach, but differs in that we use models about the application requirements. Our choice has both pros and cons: on the one side we allow for a more useful representation of a system, on the other side the establishment of traceability links between requirements and code is complex.

In the area of self-managed software [29], a three-layer architecture is proposed [45] to combine goals with software components. This approach is based upon a Sense-Plan-Act architecture made up of three layers: the goal management layer defines system goals, the change management layer executes plans and assembles a configuration of software components, the component layer handles reactive control concerns of the components. Our proposal exploits a richer goal modelling language based on means-end reasoning rather than plan composition, and enacts a reconfiguration processes that takes into account agents' autonomy.

Existing work in AI planning includes algorithms for reconfiguration. CPEF (Continuous Planning and Execution Framework) [33] combines reasoning and planning systems into a comprehensive framework that sup-

ports execution, monitoring, evaluation, and repair of plans at runtime. CPEF uses different models to guide agents reconfiguration, and defines an initial rough plan that is continuously refined rather than selecting the most adequate based on some criteria (e.g., soft-goals).

A similar approach is taken in [30], where plans are reconfigured by adding new actions (refinement) and removing obstacles (unrefinement) using heuristics. The same authors propose the Action Resource Framework [31] to characterize systems that perform continual planning based on the production and usage of resource by actions. These two works rely on a sound algorithmic theory that ensures efficiency and quality of plans; differently, we aim to provide a comprehensive framework to engineer self-reconfiguring systems—which might exploit these approaches.

Decker and Lesser propose a family of algorithms that enable coordination in multi-agent systems called Generalized Partial Global Planning (GPGP) [19]. This approach applies to heterogeneous agents that cooperate to perform a set of tasks. Planning is based on the notion of episode: a set of tasks each with a deadline. Their algorithms can be used during reconciliation to generate variants. However, unlike us, they do not consider the autonomy of the agents: they presume agents can be assigned tasks.

MADAM [22] considers self-adaptivity for mobile systems in response to context changes. They exploit architecture models to keep track of the current system configuration, and use utility functions to define the policies that lead adaptation. Their approach is very specific, both in application scenarios and in the adaptation triggers. By contrast, we exploit state-of-the-art requirements models rather than generic policies.

9 Conclusions and future directions

We propose an architecture for adaptive socio-technical systems. The architecture uses a set of models that represent the correct behaviour of the system in terms of the requirements of participating agents. The architecture also collects data about changes in the operational environment and diagnoses failures by checking monitored data against requirements models. Once a failure is identified, our architecture tries to reconcile the behaviour of the system with a correct one. In doing so it considers that an STS is made up of autonomous and uncontrollable agents, which cannot be forced to execute tasks. A key element of reconciliation plans is the establishment of social dependencies between actors. An existing STS can be reengineered to exploit our proposed architecture by adding a new agent whose goal is to support other participants.

The paper (i) presents the logical view on the architecture for self-reconfigurable STSs; (ii) defines the set of requirements models used at runtime to represent correct behaviour; (iii) introduces algorithms to perform diagnosis and reconfiguration; (iv) describes a prototype implementation of our architecture; (v) applies the prototype to a smart-home case study and evaluates the framework's scalability; and (vi) shows how an existing STS can be reengineered to exploit our architecture. The implementation shows the feasibility of our conceptual architecture. The evaluation on the case study highlights its applicability to a real-life scenario. Scalability experiments demonstrate that the prototype can be effectively used for STS with medium-sized requirements models.

Our approach suffers from some limitations, which open the way for future research directions:

- *Customization*: our approach focuses on reconfiguration in response to failures and under-performance. However, a major aspect of STSs is that human agents have very different preferences and skills. These specificities should be taken into account when reconfiguring an STS;
- *Requirements evolution*: we assume here that requirements do not change over time. An orthogonal aspect to be addressed by future work is reconfiguration in response to the evolution of requirements;
- *Complex requirements*: requirement types might be supported. For instance, goal types (maintain, cease, avoid, like in KAOS [18]) would enrich the expressiveness of the requirements language. Another dimension to consider comprises complex temporal relations and goal durations;
- *Extensive evaluation*: first, to determine if the approach has general applicability, it shall be applied to several case studies in different areas. Second, the evaluation methodology has to be systematic and include automated generation of test cases, feedback from users, requirements model validation;
- *Heuristics*: adaptation performance is a crucial factor to guarantee prompt response to threats. Thus, diagnosis and reconfiguration algorithms should as fast as possible. Solution optimality can be sacrificed to increase performance. Ad-hoc heuristics might be developed to generate good-enough solutions;
- *User intefaces*: our approach does not focus on how users interact with technical systems. The successful operation of an STSs largely depends on the quality of the user interfaces, which might be either graphical or in the spirit of disappearing computing.

References

1. Ali, R., Dalpiaz, F., Giorgini, P.: Location-based Software Modeling and Analysis: Tropos-based Approach. In: Q. Li, S. Spaccapietra, E. Yu, A. Olivé (eds.) Proceedings of the 27th International Conference on Conceptual Modeling (ER 2008), *LNCS*, vol. 5231, pp. 169–182. Springer (2008)
2. Ali, R., Dalpiaz, F., Giorgini, P.: A Goal-based Framework for Contextual Requirements Modeling and Analysis. Requirements Engineering **15**(4), 439–458 (2010). URL http://dx.doi.org/10.1007/s00766-010-0110-z
3. Baresi, L., Pasquale, L.: Live Goals for Adaptive Service Compositions. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010), pp. 114–123 (2010). DOI http://doi.acm.org/10.1145/1808984.1808997
4. Baresi, L., Pasquale, L., Spoletini, P.: Fuzzy Goals for Requirements-driven Adaptation. In: Proceedings of the 18th International IEEE Requirements Engineering Conference (RE 2010) (2010)
5. Bencomo, N., Whittle, J., Sawyer, P., Finkelstein, A., Letier, E.: Requirements Reflection: Requirements as Runtime Entities. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010), pp. 199–202. ACM (2010)
6. Bordini, R.H., Wooldridge, M., Hübner, J.F.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons (2007)
7. Bratman, M.E.: Intention, Plans, and Practical Reason. Harvard University Press Cambridge, Massachusetts (1987)
8. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An Agent-Oriented Software Development Methodology. Autonomous Agents and Multi-Agent Systems **8**(3), 203–236 (2004)
9. Bryl, V., Giorgini, P.: Self-Configuring Socio-Technical Systems: Redesign at Runtime. International Transactions on Systems Science and Applications **2**(1), 31–40 (2006)
10. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable Functions in ASP: Theory and Implementation. In: Proceedings of the 24th International Conference on Logic Programming (ICLP 2008), *LNCS*, vol. 5366, pp. 407–424. Springer (2008)
11. Campadello, S., Compagna, L., Gidoin, D., Holtmanns, S., Meduri, V., Pazzaglia, J.C.R., Seguran, M., Thomas, R.: Serenity Deliverable A7.D1.1: Scenario Selection and Definition (2006)
12. Cetina, C., Giner, P., Fons, J., Pelechano, V.: Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes. IEEE Computer **42**(10), 37–43 (2009)
13. Chess, D.M., Segal, A., Whalley, I., White, S.R.: Unity: Experiences with a Prototype Autonomic Computing System. In: Proceedings of the 1st IEEE International Conference on Autonomic Computing (ICAC 2004), pp. 140–147. IEEE Computer Society (2004)
14. Clark, J., et al.: XSL Transformations (XSLT) Version 1.0. W3C Recommendation **16**(11) (1999)
15. Cohen, P.R., Levesque, H.J.: Intention is Choice with Commitment. Artificial Intelligence **42**(2-3), 213–261 (1990)
16. Dalpiaz, F., Giorgini, P., Mylopoulos, J.: Software Self-Reconfiguration: a BDI-based Approach. In: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), pp. 1159–1160. IFAAMAS (2009)
17. Damus, C.W.: Implementing Model Integrity in EMF with MDT OCL. Eclipse Corner Articles, online at: http://www.eclipse.org/articles/article.php?file=Article-EMF-Codegen-with-OCL/index.html (2007)
18. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed Requirements Acquisition. Science of computer programming **20**(1-2), 3–50 (1993)
19. Decker, K.S., Lesser, V.R.: Designing a Family of Coordination Algorithms. In: Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS 1995), pp. 73–80 (1995)
20. Emery, F.E.: Characteristics of Socio-Technical Systems. Tech. Rep. 527, London: Tavistock Institute (1959)
21. Feather, M.S., Fickas, S., van Lamsweerde, A., Ponsard, C.: Reconciling System Requirements and Runtime Behavior. In: Proceedings of the 9th International Workshop on Software Specification and Design (IWSSD'98), pp. 50–59. IEEE Computer Society Washington, DC, USA (1998)
22. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using Architecture Models for Runtime Adaptability. IEEE Software **23**(2), 62–70 (2006). DOI http://dx.doi.org/10.1109/MS.2006.61
23. Garcia-Molina, H., Salem, K.: Sagas. In: Proceedings of the 1987 SIGMOD Annual Conference, pp. 249–259 (1987)
24. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based Self-Adaptation with Reusable Infrastructure. IEEE Computer **37**(10), 46–54 (2004). DOI 10.1109/MC.2004.175
25. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design Science in Information Systems Research. MIS Quarterly **28**(1), 75–105 (2004)
26. Horn, P.: Autonomic Computing: IBM's Perspective on the State of Information Technology. Talk given at IBM TJ Watson Labs, NY (2001)
27. Hui, B., Liaskos, S., Mylopoulos, J.: Requirements Analysis for Customizable Software Goals-Skills-Preferences Framework. In: Proceedings of the 11th IEEE International Conference on Requirements Engineering (RE'03), pp. 117–126. IEEE Computer Society (2003)
28. Jureta, I.J., Mylopoulos, J., Faulkner, S.: Revisiting the Core Ontology and Problem in Requirements Engineering. In: Proceedings of the 16th IEEE International Conference on Requirements Engineering (RE 2008), pp. 71–80 (2008)
29. Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. In: Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), pp. 259–268. IEEE Computer Society (2007)
30. van der Krogt, R., de Weerdt, M.: Plan Repair as an Extension of Planning. In: Proceedings of the 2005 International Conference on Automated Planning & Scheduling (ICAPS 2005), pp. 161–170 (2005)
31. van der Krogt, R., de Weerdt, M., Witteveen, C.: A Resource Based Framework for Planning and Replanning. Web Intelligence and Agent Systems **1**(3), 173–186 (2003)
32. van Lamsweerde, A., Letier, E.: Handling Obstacles in Goal-oriented Requirements Engineering. IEEE Transactions on Software Engineering **26**(10), 978–1005 (2000)
33. Myers, K.L.: CPEF: A Continuous Planning and Execution Framework. AI Magazine **20**(4), 63–69 (1999)
34. Object Management Group: Software & Systems Process Engineering Metamodel specification (SPEM) Version 2.0. Tech. rep. (2008)

35. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An Architecture-Based Approach to Self-Adaptive Software. IEEE Intelligent Systems **14**(3), 54–62 (1999)
36. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based Runtime Software Evolution. In: Proceedings of the 20th International Conference on Software Engineering (ICSE 1998), pp. 177–186 (1998)
37. Penserini, L., Perini, A., Susi, A., Mylopoulos, J.: High Variability Design for Software Agents: Extending Tropos. ACM Transactions on Autonomous and Adaptive Systems **2**(4), 16 (2007). DOI http://doi.acm.org/10.1145/1293731.1293736
38. Preece, J., Rogers, Y., Sharp, H.: Interaction Design: Beyond Human-Computer Interaction. John Wiley (2002)
39. Rao, A.S., Georgeff, M.P.: Modeling Rational Agents within a BDI-Architecture. In: Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91), pp. 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA (1991)
40. Rao, A.S., Georgeff, M.P.: BDI Agents: From Theory to Practice. In: Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS-95), pp. 312–319. San Fransisco, USA (1995)
41. Robinson, W.N.: A Requirements Monitoring Framework for Enterprise Systems. Requirements Engineering **11**(1), 17–41 (2006)
42. Sardina, S., Padgham, L.: Goals in the Context of BDI Plan Failure and Planning. In: Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2007), pp. 16–23 (2007)
43. Singh, M.P.: Social and Psychological Commitments in Multiagent Systems. In: AAAI Fall Symposium on Knowledge and Action at Social and Organizational Levels, pp. 104–106 (1991)
44. Singh, M.P.: An Ontology for Commitments in Multiagent Systems: Toward a Unification of Normative Concepts. Artificial Intelligence and Law **7**(1), 97–113 (1999)
45. Sykes, D., Heaven, W., Magee, J., Kramer, J.: From Goals to Components: a Combined Approach to Self-Management. In: Proceedings of the 2008 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008), pp. 1–8 (2008)
46. Wang, Y., McIlraith, S., Yu, Y., Mylopoulos, J.: An Automated Approach to Monitoring and Diagnosing Requirements. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07), pp. 293–302. ACM New York, NY, USA (2007)
47. Wang, Y., Mylopoulos, J.: Self-repair Through Reconfiguration: A Requirements Engineering Approach. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09), pp. 257–268. IEEE Computer Society (2009)
48. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B., Bruel, J.M.: RELAX: a Language to address Uncertainty in Self-adaptive Systems Requirement. Requirements Engineering **15**(2), 177–196 (2010)
49. Yu, E.S.K.: Modelling Strategic Relationships for Process Reengineering. Ph.D. thesis, University of Toronto, Toronto, Ont., Canada, Canada (1996)