UNIVERSITY
OF TRENTO

DIPARTIMENTO DI INGEGNERIA E SCIENZA DELL'INFORMAZIONE

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

Talos: an architecture for self-configuration

Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos

May 2008

Technical Report # DISI-08-026

Talos: an Architecture for Self-Configuration

Fabiano Dalpiaz, Paolo Giorgini, John Mylopoulos
University of Trento
Information Engineering and Computer Science Department
{fabiano.dalpiaz, paolo.giorgini, john.mylopoulos}@disi.unitn.it

Abstract

Autonomic computing concerns the design and development of software systems with self-maintenance capabilities and behaviors that conform to high-level policies. Self-reconfiguration is a key feature of an autonomic system. It enables self-transition from the current configuration, which may have failed or be under-performing, to a new configuration that better addresses system goals. In this paper, we propose Talos, a conceptual architecture based on the Belief-Desire-Intention (BDI) paradigm that supports various forms of self-reconfiguration. We also present a first realization of Talos in Jason, a BDI infrastructure based on the AgentSpeak programming language. Our approach follows software engineering principles and is founded on goal-oriented analysis from Requirements Engineering.

1 Introduction

Current software systems are more and more required to serve a large number of objectives switching at runtime their operational behavior. This calls for a different way of thinking and developing software, where failure handling and prompt response to low-performance results being a must. Considering low-performance as a specific case of failure (where non-functional requirements are not sufficiently met), literature addressed the failure problem through a three steps software reconfiguration process: *monitor* the current context to detect failures, *diagnose* the reason why something went wrong, and *compensate* the failure by taking appropriate countermeasures to the failure. Self-reconfiguration is the software's autonomous execution of reconfiguration, without human direction.

In the literature, many approaches have been proposed for software evolution, particularly in the area of distributed systems. Agnew et al. [1] propose Clipper, a distributed language based on C++ intended to express reconfiguration plans reacting to events generated by the application or the environment. Another effort concerning self-reconfiguration is self-adaptive software, proposed by Oreizy et al. [22]; they suggest an architecture-based approach to enable autonomous software evolution and adaptivity.

Autonomic Computing is a research initiative introduced by IBM in 2001 [11], and detailed in a vision paper by Kephart and Chess [14]. The basic idea of Autonomic Computing is to build systems with self-management capabilities (self-* properties, such as self-configuration, self-optimization, self-protection, and self-healing), in order to reduce human intervention. Self-reconfiguration can be seen as a general mechanism for the enactment of self-* properties, providing to the system the capability of switching behavior at runtime.

Research on agent technology [34] has explored for years the issue of self-reconfiguration. A software agent is a system with social abilities, that is able to cooperate and make decisions autonomously. IBM discussed and explored the use of multi-agent systems to build autonomic computing systems [15]. The most used agent architecture proposed in the literature is the Belief-Desire-Intention [26, 27] (BDI) agent architecture. Goal is a fundamental concept in the BDI architecture: it represents the intention of an agent, that drives its behavior. A BDI agent behaves accordingly to a self-reconfiguration process, where goals are continuously monitored and possibly revised.

Goals are also used in Requirements Engineering [9] and Autonomic Computing as basic concept to represent and realize variability of software [19, 17]. Goal-oriented modeling and analysis techniques explored in Requirements Engineering have been adopted to better motivate and specify the self-reconfiguration behavior of a BDI agent [24, 23].

In this paper we propose Talos, a conceptual BDI-based architecture that relates requirements models to the self-reconfiguration behavior of a software system. The emphasis of Talos is on software engineering aspects, namely it is intended to enable the engineering of actual systems, rather than providing new theories on self-reconfiguration. We describe Talos from three different perspectives, inspired by the 4+1 model of Krutchen [16]: logical, process, and scenario perspective. We present results of Talos implementation using Jason [4], demonstrating the applicability and effectiveness of the architecture to support runtime self-reconfiguration.

The paper is structured as follows: Section 2 describes related work; Section 3 describes the research baseline; Section 4 introduces Talos and presents logical and process views; Section 5 introduces smart-home, a scenario where self-reconfiguration is needed; Section 6 shows a prototype implementation of Talos to Jason; Section 7 concludes the paper and lists future work.

2 Related work

Related literature includes: comprehensive architectures, compensation techniques, and goal-oriented modeling.

In **comprehensive architectures**, the concept of self-adaptive software has been introduced by Oreizy et al. [22], and defines a class of software systems capable of modifying their own behavior in response to changes of the operating environment. Self-adaptive software supports both the consistent application of change over time (*evolution*) and the detect-plan-deploy cycle (*adaptation*). IBM's Unity [7] is an autonomic computing architecture based on *composition* to split the system into autonomous subsystems (autonomic elements), clear *communication interfaces* between autonomic elements, *utility functions* to support decision making, *policies* to express high-level guidance for the system, goal-driven self-assembly to self-configure the system according to a certain objective, choosing the best configuration of autonomic elements and resources. Li et al. [18] exploit the Monitor-Analyze-Plan-Execute cycle for service-oriented software characterized by frequent reconfigurations in response to rapid and continuous changes in requirements and in the surrounding environment. A reconfiguration process is enacted whenever *service level agreements* are violated or resource over- or under-consumption is detected.

An important concern in self-reconfiguration is **compensation**, namely handling the effects of a partially executed course of action that cannot be successfully completed. In the area of BDI agents, Unruh et al. [31] propose a framework that supports *goal-based semantic compensation*: this approach suggests the introduction of a goal-based specification to handle failures, where achieving the goal is semantically equivalent to an hypothetical undo action. This solution is very similar to the concept of Saga [21], that proposes a way to efficiently handle failures in DBMS long-lived transactions, avoiding the extensive use of rollbacks. 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 of the saga's component fails, compensation is executed for all completed or in-execution transactions of the saga. Thangarajah et al. [30] suggest to structurally provide an abort method to handle strategy abortions. This solution is also suitable to manage the cases when an external cause leads to plan failure, and the agent has to change plan regardless of its intentions.

Goal models have been used as modeling notation to express variability in requirements [19], and to support the design of autonomic application software [17]. Goal-based approaches have been applied to perform monitoring and diagnosis of software failures at runtime [32], using goal models as abstraction of the system and checking the fulfillment of the system objectives. The diagnosis is performed offline with the aid of a SAT solver, after translating both the goal model and the monitored log into boolean formulae.

3 Research baseline

Software architectures can be specified under several viewpoints, as stated by the standard ANSI/IEEE 1471-2000 [2]; a well established multi-view approach is Kruchten's 4+1 view model [16], which describes architectures in terms of logical, process, physical, development, and scenario (the "+1") views. Talos is a conceptual architecture, and we do not deal therefore with implementation-level details, keeping the architecture abstraction level general enough to enable different realizations depending on implementation-level choices. Following the principles of Kruchten's model, we present here the logical, process, and scenario architectural views, which better suit to describe a conceptual architecture.

The qualities we want to provide in Talos derive from standard software engineering principles, and are basically abstraction, modularity, and separation of concerns. One of the peculiarities of our approach is to base upon existing well-founded techniques as much as possible, instead of developing brand new techniques to face the various facets of re-configuration.

Adaptation of effective techniques contributes to give our research a software engineering perspective, where the main objective is to exploit the best solution to solve the problem, and in most cases existing work already provides successful approaches. We present now the techniques we actually took as starting point to propose our architecture.

The Belief-Desire-Intention (BDI) paradigm [26] is an agent-oriented architecture where every **agent** is characterized in terms of its **beliefs**, **desires**, **intentions**, and **plans**. *Beliefs* represent the informational state of the agent, that is what the agent believes about the world and its own state. *Desires (goals)* are the agent's motivational state, the objectives the agent wants to accomplish. *Intentions* represent the deliberative state, that is what the agent has already chosen to do; in other words, intentions are desires the agent has chosen to commit to carrying out a plan [8]. *Plans* are sequences of actions that the agent can execute to achieve its intentions. BDI systems are guided by the *BDI agent control loop* [26, 33]: roughly speaking, the agent gets a percept, updates its beliefs, decides if and how to handle the new event through a plan, selects one of the current intentions, and executes one action of the chosen intention. We believe this mechanism is suitable as starting point for Talos, since it can be easily extended to include reconfiguration mechanisms: **intention reconsideration** algorithms [5, 29] and meta-reasoning techniques [28] can be used to define how intentions are carried out, while **compensation** to failures [30, 31] is appropriate to revert the effects of the already executed actions when a reconfiguration is required.

Essential requirement to provide self-reconfiguration is **behavior variability**, that is the capability to provide alternative strategies to achieve the objectives assigned to the system. Requirements Engineering (RE) is the most authoritative candidate that provides models and techniques to represent and analyze requirements. A well established framework in RE is the **goal model** [9], where software requirements are expressed in terms of goals. Top-level goals (high level requirements) are refined into lower-level goals through and/or decomposition, defining a tree (in some goal model variants a graph). **OR-decomposition** is the basic construct enabling variability [19], since it decomposes a goal into a number of sub-goals; achieving one of these sub-goals is sufficient to achieve the decomposed goal. Goal models are exploited in agent-based software engineering methodologies, such as Tropos [6], where the system is described in terms of **socially interacting** agents depending on each other to fulfill their goals. The social aspects are very relevant to support self-reconfiguring systems: the subsystems are not isolated but cooperate to achieve their objectives, and they can sometimes choose among several candidates.

Wang et al. [32] propose **goal monitoring and diagnosis** techniques that uses goal models as abstract representation of software, with leaf-level goals (tasks) associated to actual code. Monitoring can be performed with different levels of granularity, because every goal is provided with an on/off switch defining if the achievement of the goal has to be monitored or not. The authors implicitly adopt the notion of **declarative goal**, a goal whose achievement is evaluated checking if the truth value of the goal's post-condition. We want to use the selective monitoring capabilities of Wang's framework in Talos, along with an extended usage of declarative goals. Liaskos et al. [20] propose a framework to drive the selection of the **best strategy** using goal models. This approach is based on hard- and soft-goals (the latter expressing **non-functional requirements**), plans to operationalize goals, domain concepts to express the **context**, and domain facts which are relations over domain concepts. Hierarchical planning is applied to identify the best solution, evaluating stakeholders preferences expressed in a preference specification language. In Talos we use the basic concepts of Liaskos' framework, but we pay particular attention in maintaining an acceptable computational tractability. The concept of context is important both to express (pre)-conditions for the execution of specific strategies, and to specify context-dependent satisfaction of non-functional requirements.

4 Talos

The high-level structure of Talos follows the aforementioned Monitor-Diagnose-Compensate (MDC) cycle, which is a general approach to enact software self-reconfiguration in response to failures. Talos is based on the BDI paradigm, and should then not only provide mechanisms to handle failures, but also support execution in normal circumstances. The most common way of driving autonomic systems is to use the Monitor-Analyze-Plan-Execute (MAPE) cycle, which is a generalization of the Monitor-Diagnose-Compensate cycle. The MAPE cycle monitors the environment to detect changes, analyzes the changes choosing an appropriate reaction, performs planning to enact the reaction, and executes the defined plan. MDC's monitoring is very similar to MAPE's monitoring, but we explicitly recognize that changes can involve both the environment and the agent's internal state of mind; diagnosing is a specialization of analysis, which focuses on understanding the reasons of failures; compensation is a specialization of execution dealing with reverting (undoing) the effects caused by the current failed strategy.

The logical view on Talos architecture is shown in Figure 1, through the use of an UML component diagram. This kind of diagram enables a clear structural view of the architecture, but it also gives hints about the data flow between various components, through the use of interfaces and dependency relations. Figure 2 shows the process view on Talos, through the

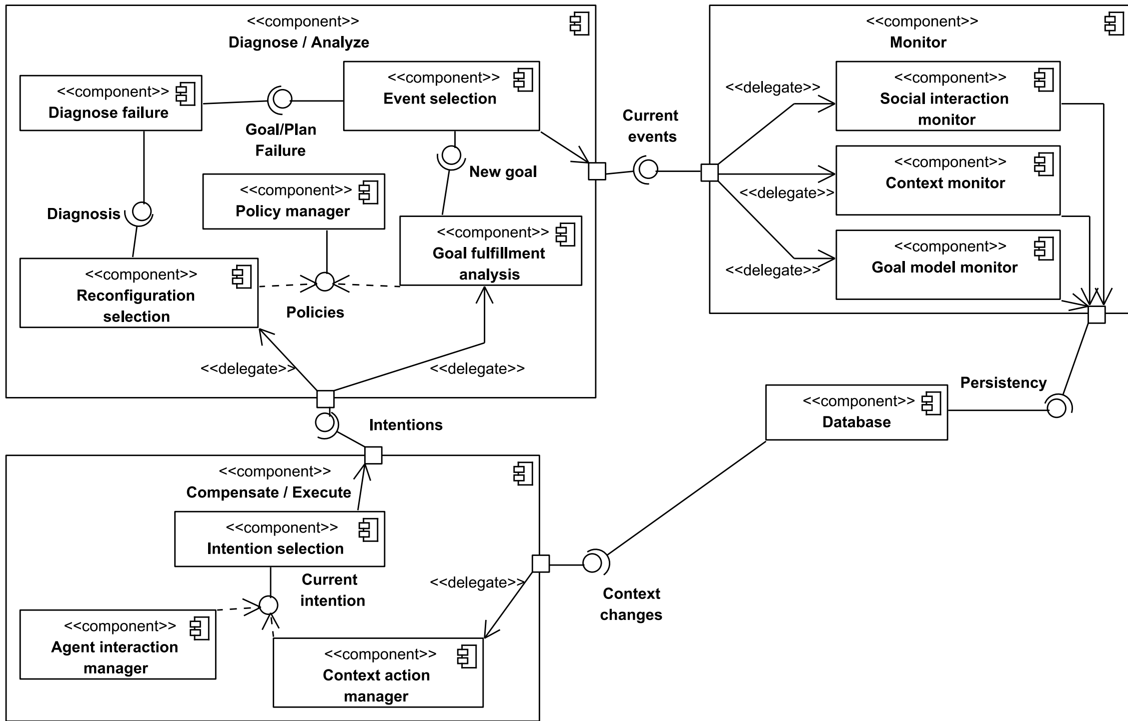

Figure 1. Logical view on Talos: detailed components are grouped according to the MAPE cycle phases.

use of an activity diagram.

Looking at the component diagram in Figure 1, we describe now the structure of Talos. There is one auxiliary component (*Database*) in charge of providing persistency; its role in the architecture is to emphasize the need of storing the current state of both the environment and the agents beliefs. In addition to the auxiliary component, Talos is characterized by three top level functional components:

- *Monitor*: it is composed of three sub-components that are responsible for social, context, and goal models monitoring. It exposes an interface *Current events*, that provides a hook to the current events of the system. It is linked to the interface *Persistency*, which enables to store the current state of the world in a database.

- *Diagnose/Analyze*: it is responsible to provide both analysis and diagnosis capabilities; it takes the current events and provides the interface *Intentions*, which gives access to the intentions the agent is committed to. The sub-components carry out the process that starts from the events and terminates with the definition of the intentions, in accordance to a set of policies giving high-level guidance to the decision-making process.

- *Compensate/Execute*: provides the execution of intentions that can either involve plans to achieve objectives or compensation actions to revert the effects of failures. This component gets the current intentions from the interface provided by the Diagnose/Analyze component, and provides an interface *Context changes* that manifests changes in the system after the execution of actions.

Before examining in detail the various sub-components of the system, we describe the process view presented in Figure 2, which should help to understand the interplay between the various components. The activity diagram is not provided with initial and final nodes, because it depicts a loop process that is repeated until the agent is terminated. The monitoring activities are carried out in parallel: *Monitor context changes* returns the set of changes in the context that are relevant to the agent; *Monitor goal state* aims at the detection of new goal instantiations, goal failures, and goal achievements; *Monitor mailbox* handles the social interactions, that is the new messages received from other agents. After monitoring the environment, the next step consists of *Event selection*, where one event is chosen among all the events to be processed. Depending

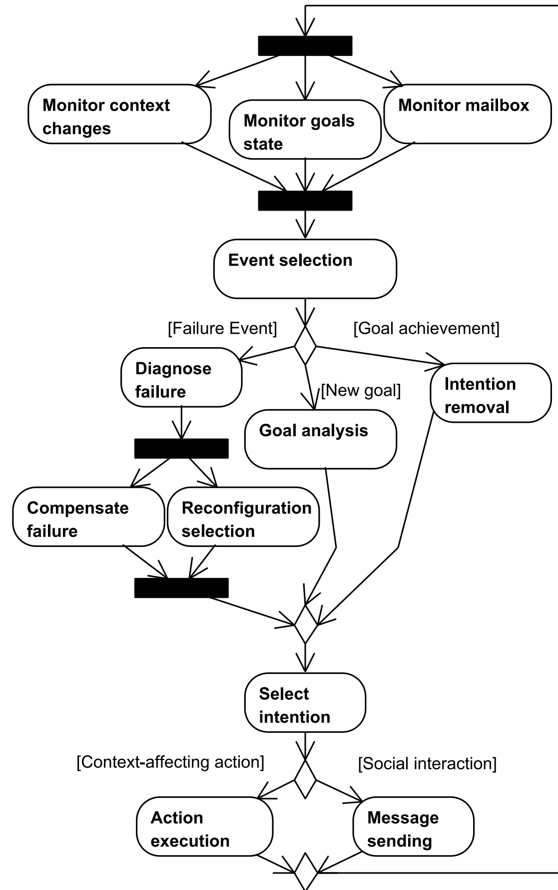

Figure 2. Process view on Talos, presented using an UML activity diagram.

on the kind of event that has been chosen for analysis, different activities are carried out: *Diagnose failure* handles failures, understanding the root cause and discovering the reason; *Goal analysis* comes from Requirements Engineering literature, and handles new goal instance by exploring the corresponding goal model looking for the most appropriate strategy; *Intention removal* is performed when the event concerns a goal successfully achieved. *Diagnose failure* requires the execution of two parallel activities in order to enable self-reconfiguration in response to failure: *Compensate failure* defines the set of actions to be performed to revert the effects of the failed activity, whereas *Reconfiguration selection* chooses an alternative strategy to achieve the top-level goal. The analysis phase is followed by the *Select Intention* activity, where one of the plans to which the agent has committed to is chosen. If the next action in the intention affects the context, *Action execution* is performed involving the agent's effectors; if the action requires social interaction, *Message sending* is carried out to enable agent communication.

We examine now in detail the role of the various sub-components of the system, highlighting existing techniques that can be exploited whenever available.

The *Social interaction monitor* is responsible for monitoring the social interaction between agents, that is it provides a sort of mailbox to store the sent and received message. It can also provide filtering mechanisms, the same way e-mail clients are enhanced with spam filters. For instance, a certain agent could filter social messages when all its resources are already allocated, in order to avoid delays to the requester, it could decide to accept only informative messages, or refuse any request originated by untrusted agents. There are many possible uses of the social interaction monitor, ranging from plain monitoring to advanced filtering activities. The *Context monitor* deals with contextual events, that is provides a direct link to the physical surrounding context. It can provide mechanisms to enhance efficiency, such as informing the agent only about events that are relevant to it, instead of notifying all the events. Another efficiency-related behavior can be monitoring the context at slower rates: this solution decreases the accuracy of the beliefs (and hence could limit the quality of strategy selection), but it allows

better performance. The *Goal model monitor* is in charge of monitoring goal models, identifying the goals just instantiated, goal failures, and goal achievement. The concept of on/off monitoring switches introduced by Wang et al. [32] is particularly relevant here, allowing to tune the monitoring sensitivity: if bottom-level goals (hence, plans) are monitored, the agent will detect failures immediately, but it will consume many resources to perform this activity. Conversely, monitoring higher level goals reduces the monitoring overhead, but it delays failure detection.

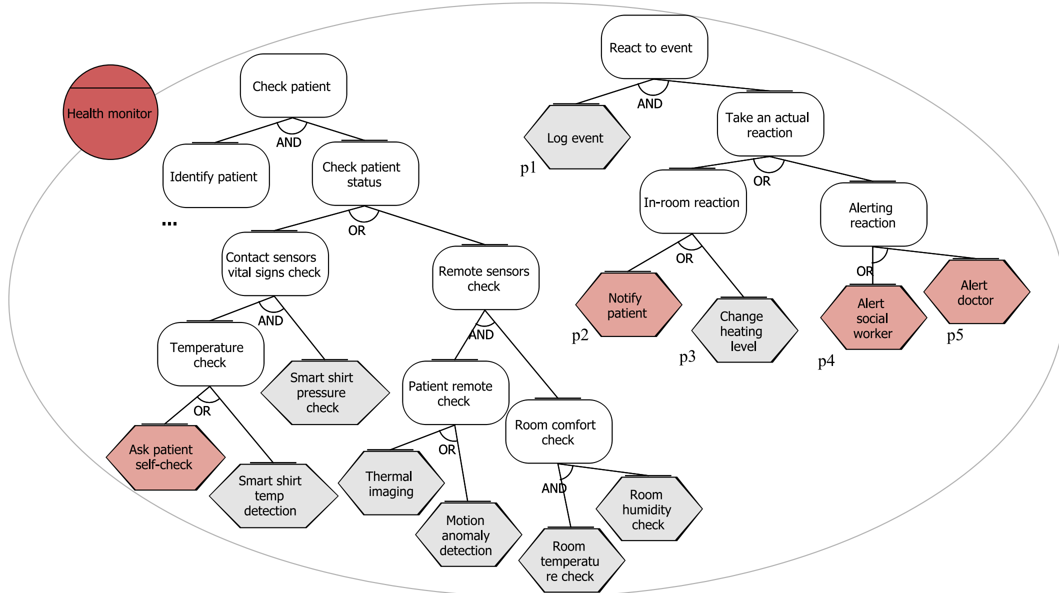

Figure 3. Goal model representing a monitoring system in a smart home. Red filled hexagons represent plans requiring social interaction.

Event selection provides mechanisms to choose among events; the selection can involve the use of priorities to process urgent events first. For example, events concerning compensation should be given higher priority than other events, in order to "undo" the failed plan before trying alternative strategies or achieving other objectives. The event selection component provides two interfaces: *Goal/Plan Failure* and *New Goal*, where the events are split according to their type. Depending on the chosen event, only one of them is filled, and the other one remains empty. The *Diagnose failure* component reads from the *Goal/Plan Failure* interface, and its role is understanding why something went wrong. There are several reasons for a goal to fail, among which explicit failure states (the plan chosen to achieve the goal returns failure), and unachieved declarative goals (the selected plan terminated with success, but the desired state of the world is not reached). *Reconfiguration selection* reads the *Diagnosis* interface provided by *Diagnose failure* component, and defines compensation actions and an alternative strategy to fulfill the goal. The choice must be consistent with the policies read in the *Policies* interface exposed by the *Policy Manager*. For instance, a certain policy could require the agent to try all the alternatives, whereas another policy could limit the maximum number of attempts. *Goal fulfillment analysis* reads from the interface *New goal* and, respecting the policies, chooses the best strategy to fulfill the goal. The best strategy should be chosen according to the contribution to soft-goals and to the applicability of plans (a variant of Liaskos et al. [20]); depending on the current beliefs, the achievement of the same goal could lead to different strategies. The *Policy Manager* component is the interface between the stakeholders and the system, providing the means to specify high level policies that guide the goal analysis (by prioritizing soft-goals, for instance) and driving the reconfiguration process when a plan fails. Although used here to drive the main steps in the self-reconfiguration process, the policy manager can be use in a broader way, enabling the definition of policies that concern different aspects of agent's execution.

Intention selection reads from the *Intentions* interface, and selects the intention to process among the existing one. The choice can be done according to many strategies, spreading from simple first-in-first-one queues, to priority mechanisms (some intentions are more urgent than others), to time-bounded executions (intention X should be completed within time t). The *Context action manager* reads the *Current Intention* interface exhibited by the *Intention selection* component, and

it handles the actions that interact with the surrounding context. This component is the interface between the agent and the context, and therefore with legacy systems and external devices. The *Agent interaction manager* handles the actions requiring interaction between agents, enabling to send messages to other agents. The development of this component involves several concerns: the selection of the communication channel to exploit (such as shared memory, RMI, SOAP), the kind of messages that are supported (informative ones, requests, acknowledges), and the provision of security properties such as non-repudiation.

5 A scenario for self reconfiguration: smart home

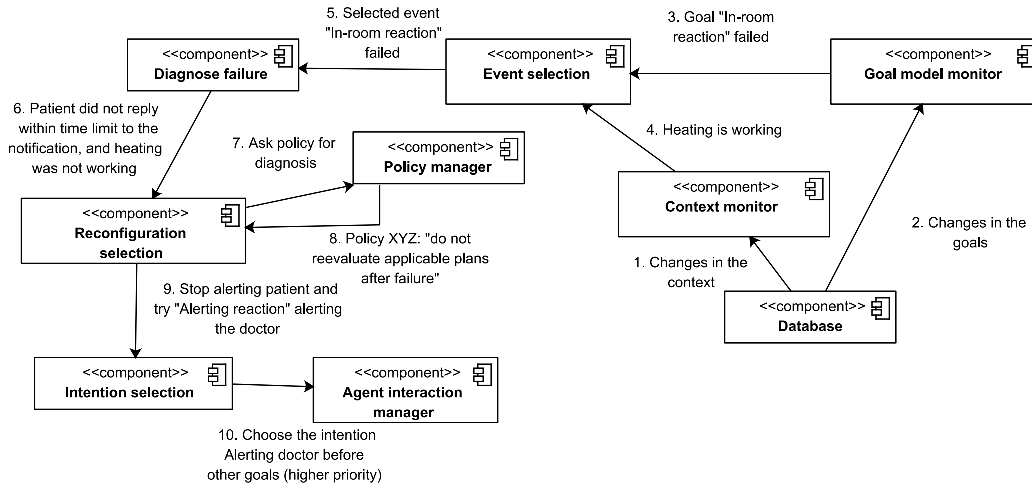

Figure 4. Scenario view on Talos, describing self-reconfiguration in the smart-home scenario.

This section plays a twofold role: on the one hand, it proposes an Ambient Intelligence scenario suitable to examine self-reconfiguration; on the other hand, it explores the "+1" view in Kruchten's architecture description framework [16]. Our scenario concerns smart home environments (see Harp [10] for a comprehensive overview), that is houses assisting human life by using with Ambient Intelligent (AmI) devices such as sensors and effectors. The degree of adaptivity of smart homes strongly depends on the capabilities of the AmI devices installed, and on the way the data the sensors provided by sensors is combined to take decisions and act (through devices called actuators). In particular, we focus on home automation for elderly and disabled people, which is a relevant application of smart-homes.

The scenario we examine is relatively simple, and relates to monitoring activities for a single room. The smart home system is handled by a *health monitor* software, operating in an environment provided with a number of sensors. The patient (typically an elderly person) can wear a *smart shirt* to monitor its vital signs; a *camera* visually supervises the room, and it can be provided with thermal imaging and motion detection capabilities; the room temperature and humidity can be checked by appropriate sensors. The system can also react to events, either by directly acting in the room (e.g., change the temperature) or remotely alerting a social worker or a doctor reporting the detected event.

Since Talos is founded on goal models, in Figure 3 we show a Tropos-like goal model (we don't stress the exact syntax, since the paper's emphasis is not on modeling) depicting the goals and plans inside the room monitor responsible for the patient's health. The health monitor has two top-level goals, that is *Check patient* and *React to event*. The former goal can be fulfilled by achieving the sub-goals *Identify patient* and *Check patient status*; we do not refine the former sub-goal. Checking the patient status can be fulfilled either by checking vital signs using contact sensors, or exploiting remote checking capabilities. The goal *Contact sensors vital signs check* is AND-decomposed into the goal *Temperature check* and the plan *Smart shirt pressure check*. The goal *Temperature check* can be achieved either by asking the patient a self-check (the plan is red-filled to indicate social interaction, that differs from other plans because the social partner can refuse to help the requesting agent), or by using smart shirt temperature detection. The goal *Remote sensors check* involves both patient remote check and room comfort check. The patient can be remotely check using *Thermal imaging* capabilities of the camera or *Motion anomaly detection* features. The room comfort involves checking both the temperature and the humidity level.

The second top-level goal, *React to event*, is triggered when an event arises by executing the plans to achieve the goal *Check patient status*. React to event is fulfilled by logging the event, and taking an actual reaction. Depending on the severity of the event and the smart home's self-adjusting capabilities, the reaction can be either in-room or an external alert. The *In-room reaction* goal can be achieved either by changing the temperature in the room or notifying the patient of the current event. The *Alerting reaction* can be fulfilled either by alerting a social worker or a doctor, depending on the severity of the event.

We use the above presented description of the smart-home scenario to present the scenario view on Talos. According to Krutchen [16]), this view should be described using a variation of the logical view where the logical entities are connected by directed links representing the information flow. This diagram (called *script* by Krutchen) should refer to a particular scenario, and is aimed at showing the applicability of the architecture.

In Figure 4 we exploit a component diagram enhanced with labeled numbered arrows, describing the sequential ordering of interactions between components. The script we propose describes a self-reconfiguration process responding to goal failure. The Database component signals changes in the context to the Context monitor (1) and changes in the goals to the Goal model monitor (2). The Goal model monitor detects that one of the changes is the failure of the goal "In-room reaction", and notifies this failure to the Event selection component (3). The Context monitor identifies a relevant event (heating is working) and informs the Event selection of this fact (4). The Event selection checks the events to be processed, among which there are at least the two events just received; it selects the failure of in-room reaction, maybe because failure handling has the highest priority, and informs the Diagnose failure component (5). The diagnosis is that the patient did not reply within time limit to the notification (the last attempted plan to achieve the goal was "Notify patient"), and the heating was not working (therefore the alternative plan "Change heating level" to achieve the goal was not applicable). This diagnosis is sent to the Reconfiguration selection component (6), which asks a policy to the Policy manager (7) and receives the policy XYZ (8) stating that applicable plans don't have to be reevaluated after failure. This implies that the goal "In-Room reaction" is not achievable anymore; the reaction we suggest is bottom-up exploring the goal model, that is finding an alternative plan for the goal "Take an actual reaction". The Reconfiguration selection chooses to alert the doctor (according to its selection criteria, such as the contribution to non-functional requirements), and informs the Intention selection component (9); the last step is to select an intention among the active ones, and the agent chooses "Alert doctor" because reconfigurations have higher priority than normal operation (10).

6 Prototype implementation of Talos on Jason

Jason [4] is a BDI infrastructure interpreting an extended version of AgentSpeak [25], an agent-oriented programming language intended to fill the gap between BDI theory and BDI (efficient) usage in practice, and founded on a restricted first-order logic language with events and actions. The authors of Jason have provided a formal semantics to the fragments of AgentSpeak not well defined, and have added some further constructs. Jason enables the usage of non-logic programmed behaviors, using Java as external programming language. For instance, interfaces with the context (external devices or legacy systems) can be programmed in Java, and associated to logic-specified AgentSpeak actions, enabling to deploy action effects on the real environment. Jason can be run either centralized as a Java program, or in a decentralized way, exploiting MAS infrastructures such as JADE [3] or SACI [12]. One of the features that make Jason suitable as testbed for Talos is that it is highly extensible and customizable: apart from being an open-source project [13], it provides various points where the interpreter behavior can be customized and extended. An important concept in Jason is the *Environment*, that represents the world where the system agents are situated and they execute; the Environment can be programmed to define the effects of actions and the way changes are perceived by agents. Jason's Environment corresponds to the concept of context in Talos.

Jason execution is lead by the Jason reasoning cycle, which slightly extends the BDI reasoning cycle, and defines the sequence of activities each agent carries out in every cycle. We explain now its dynamics (for further explanation look at [4]), and define a mapping with Talos, showing which parts of Talos are implemented by the default behavior, and which other features require extension or customization.

1. *Perceive the environment:* changes in the Environment produce events specified via facts addition or removal. Talos handles this step in the Monitor component, and in particular through the *Context monitor* and the *Goal model monitor*. Jason does not provide explicit modules to separately handle these two different types of monitoring, but AgentSpeak syntax helps because it distinguishes goals from beliefs.

2. *Update the belief base:* the agent's beliefs are updated in response to the perceived environmental changes. The default strategy adds all perceivable events to the beliefs. Talos implicitly contains this activity in the Monitor component. We

decided not to explicitly deal with belief update and revision problems, and suggest to exploit simple ways to handle these aspects avoiding belief conflicts and inconsistencies. The default belief update function provided by Jason is a possible realization of our suggestion.

3. *Receive communication from other agents:* every agent has a mailbox storing the messages received from other agents, and at each step the mailbox is updated with new messages. The *Social interaction monitor* in Talos handles this activity, and moreover it can provide additional filtering features.

4. *Select "socially acceptable" messages:* agents can implement customized social filtering functions, to define different handling strategies for various types of messages. As explained in step 3, Talos can implement filtering activities in the *Social interaction monitor*. Moreover, the *Diagnose/Analysis* component can be exploited to define ad-hoc reactions to interaction events.

5. *Select an event:* only one event per cycle can be processed; the default algorithm in Jason is FIFO-based, but priority-based mechanisms can be defined by the user customizing Jason. This activity can be 1-to-1 mapped to the Talos *Event selection* component.

6. *Retrieve all relevant plans:* the agent retrieves all the plans that can be used to process the selected event. Talos has two different components responsible for various types of event: the *Diagnose failure* component handles failures, whereas the *Goal fulfillment analysis* deals with new goals.

7. *Determine all applicable plans:* applicable plans are a subset of relevant plans in BDI literature, that is those plans whose precondition (context condition) evaluates to true. Talos does not explicitly differentiate this activity from step 6, because it is a conceptual architecture that does not deal with implementation-level choices.

8. *Select one applicable plan:* the agent chooses one of the applicable plans, according to its selection strategy. The default behavior in Jason is choosing plans in the order they are declared in the code, but supports customized selection, which we believe a key feature to enact effective self-reconfiguration. This step corresponds to the components *Reconfiguration selection* and *Goal fulfillment analysis* in Talos, where we differentiate between reconfigurations and new goals.

9. *Select an Intention:* an intention is an instantiated plan the agent commits to carry out to achieve a goal; agents (may) have multiple concurrent intentions, among which they have to choose before acting. This mechanism is supported in Talos, where intentions are provided by the *Diagnose/Analyze* component, and they become input for the *Intention selection* sub-component of *Compensate/Execute*.

10. *Execute one step of the intention:* the chosen intention is actually carried out executing the next action in the intention. Both Jason and Talos have separate actuators to handle actions affecting the environment/context (*Context action manager* in Talos) and agent interaction (*Agent interaction manager* component).

It should be clear that the realization of Talos on Jason is possible and many parts can be mapped almost 1-to-1; we explore now in more detail those parts that require more effort, and the work-in-progress in extending/customizing Jason to better support self-reconfiguration. Jason has been modified using its default extension mechanism, that is by extending the Java class *Agent* and overriding the methods corresponding to the different steps of its reasoning cycle.

An important decision to be taken is **when to reconfigure**: this can be interpreted in a wider sense, defining various types of event triggering plans; for instance, new goal instances can be treated as reconfiguration events, since they may requires a reallocation of the agent's resources. Jason enables to define plans in response to belief additions and removals, goal instantiation, and goal failure. In Talos we have chosen to use goal models as driving force to define software behavior, and hence *new goals* and *goal/plan failures* play the most relevant role. We also use *declarative goals* to provide further reconfiguration triggers: a reconfiguration is required either if a plan fails, or if it succeeds but it does not reach the objective (the goal is defined in terms of a desired state of the world) it was supposed to achieve. Declarative goals can be rendered in Jason with the aid of test goals. For instance, the following AgentSpeak code snippet reacts to the addition (+) of an achievement goal (!) named *alert*.

```
+!alert(Evt,Pat) : severity(Evt,low) <-
alert_soc_worker(Evt,Pat); ?alert(Evt,Pat).
```

```
1  softgoal(eff,0.8).
2  softgoal(cost,0.2).

3  contrib(p2,eff,Evt,Pat,V) :- age(Pat)>=75 & V=-0.3.
4  contrib(p2,eff,Evt,Pat,V) :- age(Pat)<75 & severity(Evt,high) & V=0.3.
5  contrib(p2,eff,Evt,Pat,V) :- age(Pat)<75 & severity(Evt,low) & V=0.7.
6  contrib(p2,cost,V) :- V=1.0.
7  contrib(p3,eff,Evt,Pat,V) :- severity(Evt,low) & V=0.6.
8  contrib(p3,eff,Evt,Pat,V) :- severity(Evt,high) & V=-0.2.
9  contrib(p3,cost,V) :- V=1.0.
10 contrib(p4,eff,Evt,Pat,V) :- severity(Evt,low) & V=1.0.
11 contrib(p4,cost,V) :- V=-0.7.
12 contrib(p5,eff,Pat,V) :- V=1.0.
13 contrib(p5,cost,V) :- V=-1.0.

14 @g1[tgoal] +!react(Evt,Pat) :
15    contrib(p2,eff,Evt,Pat,V_p2_eff) &
16    contrib(p2,cost,V_p2_cost) &
17    contrib(p3,eff,Evt,Pat,V_p3_eff) &
18    contrib(p3,cost,V_p3_cost) &
19    contrib(p4,eff,Evt,Pat,V_p4_eff) &
20    contrib(p4,cost,V_p4_cost) &
21    contrib(p5,eff,Pat,V_p5_eff) &
22    contrib(p5,cost,V_p5_cost)
23    <- !log_event(Evt,Pat); !take_reaction(Evt,Pat).
24 @p1[plan] +!log_event(Evt,Pat) : true <- log(Evt,Pat).
25 @g2[goal] +!take_reaction(Evt,Pat) : true <- !in_room_react(Evt,Pat); ?take_reaction(Evt,Pat).
26 @p2[plan] +!in_room_react(Evt,Pat) : pda_active(Pat) <- notify_patient(Evt,Pat).
27 @p3[plan] +!in_room_react(Evt,Pat) : works(heating) <- adjust_heating(Evt,Pat).
28 @g3[goal] +!take_reaction(Evt,Pat) : true <- !alert(Evt,Pat); ?take_reaction(Evt,Pat).
29 @p4[plan] +!alert(Evt,Pat) : severity(Evt,low) <- alert_soc_worker(Evt,Pat); ?alert(Evt,Pat).
30 @p5[plan] +!alert(Evt,Pat) : true <- alert_doctor(Evt,Pat); ?alert(Evt,Pat).

31 @c1 -!in_room_react(Evt,Pat) : true <- log_failure(Evt,Pat).
32 @c2 -!take_reaction(Evt,Pat) : severity(Evt,high) <- sound_alert.
33 @c3 -!take_reaction(Evt,Pat) : severity(Evt,low) <- log_failure(Evt,Pat).
```

Table 1. AgentSpeak code augmented with Talos concepts handling the goal "React to Event" in Figure 3.

The goal has two parameters: *Evt* and *Pat*, representing the event to notify and the patient the event relates to; *severity(Evt,low)* is the precondition requiring the event's severity to be low; the plan consists of the action *alert_soc_worker(Evt,Pat)* (defined in the Environment), and the declarative nature of the goal is ensured by the test goal *?alert(Evt,Pat)*, which returns false if the plan fails. This may happen in two cases: either the plan returns failure, or it succeeds but the desired effect is not achieved (in our example, the social worker cannot help the patient because she is busy).

Another concern to be addressed is **which choice to select**, that is which are the most suitable algorithms to select the best strategy, when more options are available to satisfy a certain goal. We developed an algorithm based on a simple version of goal models, where a top-level goal is and/xor decomposed into sub-goals till the level of plans. Plans are considered as atomic entities for the purpose of selecting the best strategy, although they can be complex and include variability in their internals. Our variant of goal models is therefore *two-layered*: the top level consists of goal refinement till the level of plans, the bottom level defines internal structure of plans. Here we deal only with the top level, because our aim is selecting the best strategy to fulfill a goal. We can resume the algorithm recipe as follows:

- *Soft-goals have a relative weight* in the interval $[0, 1]$, and represent the criteria driving the selection. The sum of soft-goals relative weight must be equal to 1;
- *Plans contribute to soft-goals*: the contribution may vary depending on the current beliefs of the agent (among which is also the perceived context), or be belief-independent;
- *And-decomposition propagates minimum contributions* from the sub-goals to the parent goal;

- *Xor-decomposition propagates maximum contribution* from the sub-goals to the parent goal, because the best option only is considered. We used Xor instead of Or to keep the algorithm's complexity within $O(n)$, where n is the number of goals and plans;
- *The selection is bottom-up in linear time*: it starts from the plans, and intermediate goals are annotated with the best plan (from that point down to the level of plans) and contribution degree, until reaching the top-level goal. The algorithm uses a post-order visit.

Adding our algorithm to Jason required us to introduce many concepts deriving from goal models, and map them to AgentSpeak code; Table 6 shows a code snippet that enables our customized Agent class to perform adequate reasoning and select among alternatives, related to the goal *React to event* in the goal model of Figure 3.

Lines 1-2 are beliefs declaring the soft-goals in the smart-home scenario and their relative weight: the efficacy of the strategy (*eff*) has relative weight 0.8, whereas its *cost* has weight 0.2.

Lines 3-13 contain the rules used to define the contribution formulae from plans to soft-goals; the plan names p1-p5 follow the labels in Figure 3. The contribution from p2 (*notify patient*) to efficacy (lines 3-5) depends on the age of the patient and the severity of the event, and ranges from -0.3 to +0.7. Note the use of variables in rules: Jason considers variables the literals beginning with a capital letter, such as *Evt*, *Pat*, and *V*. The contribution from p2 to the cost (line 6) is belief-independently +1.0, because no additional human resource is involved. For the plan p3 (*Change heating level*) the situation is similar, with no cost but limited efficacy, especially when the severity is high. The plans p4 (*Alert social worker*) and p5 (*Alert doctor*) have negative contribution to cost, but high efficacy, except for p4 when the severity is not low (no rules imply zero contribution).

Lines 14-30 declare the goal model, using Jason annotations to distinguish Tropos-like goals from regular Jason goals: @*label*[tgoal] for top-level goals, @*label*[goal] for intermediate goals, and @*label*[plan] for plans. Annotations are used to let our extension of the Agent class process goals as goal models, using the above presented algorithm. Line 14 declares the top level goal *react* with parameters *Evt* (event) and *Pat* (patient); lines 15-22 are needed to bind the contribution rules to the current goal instance, obtaining the current contributions from plans to soft-goals that are needed to apply the algorithm; line 23 is the body of the goal, that is an and-decomposition. Lines 24-30 define the goal model refinements; we remark the usage of plan preconditions (e.g., in line 27, adjusting the heating can be performed only if the heating works), the handling of xor-decompositions defining more plans to handle the same event (e.g., lines 26 and 27 are two plans to provide in-room reactions), and the use of declarative goals (line 25, for instance). The fulfillment of declarative goals is not expressed using AgentSpeak code; the actions composing the executed plans, defined in the system's environment, will determine the achievement or not of the expected state of the world. When the goal model is instantiated, our algorithm is run on the goal model, and it records all the possible choices to fulfill or-decompositions, keeping them ordered according to the soft-goal contributions.

Lines 31-33 introduce the third issue to be addressed, that is **how to compensate failures and reconfigure**. Jason includes a mechanism that deals with goal failures, using the construct "-!" to handle them, allowing the definition of plans to enact compensation. The failure of a plan is propagated bottom up through the corresponding intention till finding a failure handling costruct. For instance, the failure of plans p4 and p5 is not directly handled defining a "-!alert" plan: their failure is managed by the one of the two higher-level "-!take_reaction" plans (the selection between them depends on their context conditions, which are mutually exclusive). This solution fits well to monitor goal models, allowing the designer to define the points in the goal model where failures should be handled. We extended this mechanism enabling actual self-reconfiguration, namely the choice of another strategy to achieve the goal G after one or more strategies failed:

- if G is or-decomposed, the agent looks for an alternative option to fulfill G, choosing among the applicable ones that have not been attempted yet. If no further option is available, backtracking in the goal model is performed (the parent of G is recursively explored);
- if G is and-decomposed, all the *plans* already executed to achieve G should be compensated, if compensation plans are defined. This step requires to explore G top-down till the level of plans.

7 Final remarks

In this paper we have presented Talos, a conceptual architecture that supports software self-reconfiguration. Talos uses the Belief-Desire-Intention paradigm for the general architecture of the system, goal-oriented modeling techniques to relate

requirements to the actual implementation, and failure compensation techniques to guide the self-reconfiguration process. We have discussed the extension of Jason BDI infrastructure for the realization of Talos, namely the encoding of goal models and failure handling techniques to represent the behavioral variability of the system, drive the selection process, and enable self-reconfiguration.

In the extension of Jason we have mainly focused on a local view of self-reconfiguration where agents act according to their utility (expressed in terms of soft-goals contribution). We are currently exploring self-reconfiguration from a more global perspective in terms of multiple interacting agents. We are exploiting goal delegation and plan transfer mechanisms, available in Jason, to increase the reconfiguration capabilities of the entire system.

Future work includes the definition and the use of specific criteria for the evaluation of the architecture in real-world scenarios. We will focus on quality aspects of the abstract architecture, and performance of its actual implementation. We are working on a smart-home scenario, where the self-reconfiguration is a critical feature for the effectiveness of the software system.

References

[1] B. Agnew, C. Hofmeister, and J. Purtilo. Planning for change: A reconfiguration language for distributed systems. *Distributed Systems Engineering*, 1(5):313–322, 1994.

[2] ANSI/IEEE 1471-2000. IEEE recommended practice for architectural description of software-intensive systems, 2000.

[3] F. Bellifemine, A. Poggi, and G. Rimassa. Jade–a fipa-compliant agent framework. *Proceedings of PAAM*, 99:97–108, 1999.

[4] R. H. Bordini, M. Wooldridge, and J. F. Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

[5] M. Bratman. *Intention, plans, and practical reason*. Harvard University Press Cambridge, Mass, 1987.

[6] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[7] D. Chess, A. Segal, I. Whalley, and S. White. Unity: experiences with a prototype autonomic computing system. *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 140–147, 2004.

[8] P. Cohen and H. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(2-3):213–261, 1990.

[9] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Selected Papers of the Sixth International Workshop on Software Specification and Design table of contents*, pages 3–50, 1993.

[10] R. Harper. *Inside the Smart Home*. Springer, 2003.

[11] P. Horn. Autonomic computing: Ibms perspective on the state of information technology. *IBM TJ Watson Labs, NY, 15th October*, 2001.

[12] J. Hubner and J. Sichman. Saci programming guide. *Polythecnic School, University of So Paulo (EPUSP), Technics Intelligents Laboratory*, 2003.

[13] Jason Website. http://jason.sf.net/.

[14] D. Kephart, J.O.; Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan 2003.

[15] J. Kephart and W. Walsh. An artificial intelligence perspective on autonomic computing policies. *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, pages 3–12, 2004.

[16] P. Kruchten. The 4+ 1 view model of architecture. *Software, IEEE*, 12(6):42–50, 1995.

[17] A. Lapouchnian, Y. Yu, S. Liaskos, and J. Mylopoulos. Requirements-driven design of autonomic application software. *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, 2006.

[18] Y. Li, K. Sun, J. Qiu, and Y. Chen. Self-reconfiguration of service-based systems: a case study for service level agreements and resource optimization. *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, 1:266–273, 2005.

[19] S. Liaskos, A. Lapouchnian, Y. Yu, E. Yu, and J. Mylopoulos. On goal-based variability acquisition and analysis. *Proc. 14th IEEE International Requirements Engineering Conference, Minneapolis, USA, Sep*, pages 11–15, 2006.

[20] S. Liaskos, S. McIlraith, and J. Mylopoulos. Representing and reasoning with preference requirements using goals. Technical report, Tech. rep. CSRG-542, Computer Science Department, University of Toronto, 2006.

[21] H. Molina and K. Salem. Sagas. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1987.

[22] P. Oreizy, N. Medvidovic, and R. Taylor. Architecture-based runtime software evolution. *Software Engineering, 1998. Proceedings of the 1998 (20th) International Conference on*, pages 177–186, 1998.

[23] L. Penserini, A. Perini, A. Susi, M. Morandini, and J. Mylopoulos. A design framework for generating bdi-agents from goal models. *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, 2007.

[24] L. Penserini, A. Perini, A. Susi, and J. Mylopoulos. High variability design for software agents: Extending tropos. *ACM Trans. Auton. Adapt. Syst.*, 2(4):16, 2007.

[25] A. Rao. Agentspeak (l): Bdi agents speak out in a logical computable language. *Agents Breaking Away: 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'96, Eindhoven, Netherlands, January 22-25, 1996: Proceedings*, 1996.

[26] A. Rao and M. Georgeff. An abstract architecture for rational agents. *Proceedings of Knowledge Representation and Reasoning (KR&R-92)*, pages 439–449, 1992.
[27] A. Rao and M. Georgeff. Bdi agents: From theory to practice. *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, 1995.
[28] S. Russell and E. Wefald. Principles of metareasoning. *Knowledge Representation*, 40(1001):361–305, 1992.
[29] M. Schut, M. Wooldridge, and S. Parsons. The theory and practice of intention reconsideration. *Journal of Experimental & Theoretical Artificial Intelligence*, 16(4):261–293, 2004.
[30] J. Thangarajah, J. Harland, D. Morley, and N. Yorke-Smith. Aborting tasks in bdi agents. *Proceedings of the 6th International Conference on Autonomous Agents and Multi-Agent Systems*, pages 8–15, 2007.
[31] A. Unruh, J. Bailey, and K. Ramamohanarao. Managing semantic compensation in a multi-agent system. *The 12th International Conference on Cooperative Information Systems*, 2004.
[32] Y. Wang, S. McIlraith, Y. Yu, and J. Mylopoulos. An automated approach to monitoring and diagnosing requirements. *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 293–302, 2007.
[33] M. Wooldridge. *Reasoning About Rational Agents*. MIT Press, 2000.
[34] M. Wooldridge and N. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.