Self-adaptive smart spaces by proactive means–end reasoning

Luca Sabatucci & Massimo Cossentino

Journal of Reliable Intelligent Environments

ISSN 2199-4668
Volume 3
Number 3

J Reliable Intell Environ (2017) 3:159-175
DOI 10.1007/s40860-017-0047-9

Springer

J Reliable Intell Environ (2017) 3:159–175
DOI 10.1007/s40860-017-0047-9

ORIGINAL ARTICLE

Self-adaptive smart spaces by proactive means–end reasoning

Luca Sabatucci[1] · **Massimo Cossentino**[1]

Received: 30 June 2017 / Accepted: 20 July 2017 / Published online: 31 July 2017

Abstract The ability of a system to change its behavior at run-time is one of the foundations for engineering intelligent environments. The vision of computing systems that can manage themselves is fascinating, but to date, it presents many intellectual challenges to face. Run-time goal-model artifacts represent a typical approach to communicate requirements to the system and open new directions for dealing with self-adaptation. This paper presents a theoretical framework and a general architecture for engineering self-adaptive smart spaces by breaking out some design-time constraints between goals and tasks. The architecture supports software evolution because goals may be changed during the application lifecycle. The architecture is responsible for configuring its components as the result of a decision-making algorithm working at the knowledge level. The approach is specifically suitable for developing smart space systems, promoting scalability and reusability. The proposed architecture is evaluated through the execution of a set of randomized stress tests.

Keywords Self-adaptive systems · Software architecture · Smart spaces

1 Introduction

In the past decades, technology has become pervasive in our lives due to the variety of devices which are used by people of all ages during their daily routine [32]. The miniaturization and the growth of performance of these devices constitute the technological enabler for a new generation of systems called intelligent environments [19].

An *intelligent environment* is one in which the physical environment is enriched with sensing devices, and it is supported by intelligent software that orchestrates services and provides holistic functionality that enhances occupants experiences [4].

As long as software systems grow in size, complexity, heterogeneity, and interconnection, it becomes central to design and implement them in a more versatile, flexible, resilient, and robust way. The IBM manifesto of autonomic computing [37], released in 2001, suggests a promising direction for facing software complexity through self-adaptation. The direction is shown through many research roadmaps [18,26] of software engineering for a self-adaptive system that defines *self-adaptive systems as those systems able to autonomously modify their behavior and/or their structure in response to their perception of the environment, and the operative context, to address their goals* [26].

As declared by the manifesto [4], the ability to change the behavior at run-time is one of the foundations for engineering intelligent environments. Self-adaptation has deep roots in several research fields, as artificial intelligence, biologically inspired computing, robotics, requirements/knowledge engineering, control theory, fault-tolerant computing, and so on. In the past decade, the vast and heterogeneous number of works concerning self-adaptation investigated several aspects of the problem, for instance, specific architectures for implementing adaptive control loops [49], self-organizing paradigms [5], adaptive requirements [25] and so on. However, to date, many of these problems remain significant intellectual challenges [18,26].

Among others, one point is becoming clear: until self-adaptive systems become a reality, human users (not only

✉ Luca Sabatucci
luca.sabatucci@icar.cnr.it

Massimo Cossentino
massimo.cossentino@icar.cnr.it

[1] Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR), National Research Council, Palermo, Italy

managers) will participate in the process of adaptation [8]. This aspect is of paramount importance for the models@runtime community [10] that is looking for appropriate artifacts to shorten the distance between user and system through a model of requirements and functionality at a high level of abstraction. However, traditional requirements specification languages need to evolve for explicitly encapsulating points of variability in the behavior of the system [43] and elements of uncertainty in the environment [62]. These elements must be first class entities the system can exploit to decide how to act. Currently, goal-oriented methodologies [15,25] represent the trend for specifying how a software system may adapt through the conceptualization of system's objectives and system variation points. In particular, goal-models allow describing alternative ways to address system's objectives. Goals represent "invariant points" that motivates the whole mechanism of adaptation.

In previous works we observed that functional requirements could be run-time entities, to provide to the system according to specific user needs. We also adopted goals as a primary way to describe system's objectives. Moreover, we explored a mechanism for *injecting or changing goal-models* during system execution. To this aim, we defined a human-oriented language for specifying system goals [58]. We also set up a formal background, based on the concept of *state of the world*, for allowing the system to run when the specifications of how to address goals are not provided together with the goal model. The result is the *PMR Ability*, i.e., a facility of the system for autonomously deciding how to operationalize a given goal for which it has no hard-coded knowledge [54].

This paper aims at refining the problem of proactive means–end reasoning and implementing a general architecture for adaptation that, working at the knowledge level [45], is independent of any specific application context, but it rather can be reused in many domains. A particular focus is given to atomic and self-contained portion of behavior, called *capabilities*, which implement the paradigm of full-reuse [7]. Indeed their peculiarity is of being automatically composable, on demand, to build system functionalities and to address dynamic and evolving goals. The proposed architecture integrates the MAPE-K model [16,49] to deal with three characterizations of self-adaptation: system evolution, self-configuration, and self-healing.

A prototype of the architecture has been implemented in JASON [12], a declarative programming language based on BDI theory [14]. We also randomly generated a set of stress tests to evaluate the performance of self-adaptation. The result provided us interesting findings for planning future works.

The paper is structured as follows: Sect. 2 presents the theoretical background and defines some basic concepts. Section 3 presents a knowledge-level approach for solving the proactive means–end reasoning problem through a top-down strategy combined with an algorithm for capability composition. Section 4 presents the architecture based on the ability to solve the proactive means–end reasoning problem and the MAPE-K model. Section 5 presents the results of a set of tests, compares the approach with some relevant works from the state of the art and, finally, discusses strengths and limits of the approach. Section 6 briefly summarizes the proposed architecture. Other details of the prototype are in the Appendix.

2 Background and definition

This section illustrates the theoretical background that introduces the basic concepts of this paper.

The running example is a self-adaptive management system for an exhibition center. Such a system shall manage thousands of visitors each day. The aim is to support visitors for improving their experience. Usually, there are multiple areas and stands for different companies and products, and also zones for meeting and conferences. Visitors have different interests; most of them are usually only interested in some specific topics and products. They can use their smartphone/tablet to register, declare their interests, get navigational help, and discover new and different opportunities. Big displays may help people find the right directions by detecting the presence of people in their surrounding.

The system shall also support the organizers at managing the crowd by properly distributing the different groups of people, avoiding overcrowding and granting security norms. The support in case of emergency is of paramount importance: the smart infrastructure shall also be to coordinate people to safely get out by providing information about the safest and nearest exits and by telling people to use all the emergency exits uniformly.

2.1 State of the world and goals

We consider a software system has (partial) knowledge about the environment in which it runs. The classic way for expressing this property is (Bel $a\ \varphi$) [63] that specifies that a software agent a believes φ is true, where φ is a generic state of affairs. We decided to limit the range of φ to first-order variable-free statements (facts). They are expressive enough for representing an object of the environment, a particular property of an object or a relationship between two or more objects. A fact is a statement to which it is possible to assign a truth-value. Examples are as follows: $tall(john)$ or $likes(john, music)$.

Definition 1 (*State of the world*) The state of the world in a given time τ is a set $W^\tau \subset S$ where S is the set of all the (non-negated) first-order variable-free statements (facts) $s_1, s_2 \ldots s_n$ that can be used in a given domain.

W^τ has the following characteristics:

$$W^\tau = \{s_i \in S | (\text{Bel } a\ s_i),\} \quad (1)$$

where a is the subjective point of view (i.e., the execution engine) that believes all facts in W^τ are true at time τ.

W^t describes a closed-world in which everything that is not explicitly declared as true is then assumed to be false.

An example of W^t is $\{tall(john), age(john, 16), likes(john, music)\}$.

A State of the World is said to be *consistent* when $\forall s_i, s_j \in S$

$$\text{if } \{s_i, s_j\} \models \perp \text{ then } \begin{cases} s_i \in W^\tau \Rightarrow s_j \notin W^\tau \\ s_j \in W^\tau \Rightarrow s_i \notin W^\tau, \end{cases} \quad (2)$$

i.e., it contains only facts with no (semantic) contradictions. For instance the set $\{tall(john), small(john)\}$ is not a valid state of the world since the two facts produce a semantic contradiction.

A condition $\varphi : W^\tau \longrightarrow \{true, false\}$ of a state of the world is a logical formula composed by predicates or variables, through the standard set of logical connectives $(\neg, \wedge, \vee)$. A condition may be tested against a given W^τ through the operator of unification.

For instance, the condition $\varphi = likes(Someone, music) \wedge age(Someone, 16)$ is true in the state of the world $\{tall(john), age(john, 16), likes(john, music)\}$ through the binding $Someone \mapsto john$ that realizes the syntactic equality.

In many goal-oriented requirements engineering methods the definition of *Goal* [15] is the following: "a goal is a state of affairs that an actor wants to achieve". We refined this statement to be compatible with the definition of W^t as follows: "a goal is a desired *change* in the state of the world an actor wants to achieve", in line with [1]. Therefore, to make this definition operative, it is useful to characterize a goal in terms of a triggering condition and a final state.

Definition 2 (*Goal*) A goal is a pair: $\langle tc, fs \rangle$ where *tc* and *fs* are conditions to evaluate (over a state of the world). Respectively, the tc describes when the goal should be actively pursued and the fs describes the desired state of the world. Moreover, given a W^t we say that

$$\text{the goal is } addressed \text{ iff } tc(W^t) \wedge \Diamond fs(W^{t+k}) \text{ where } k > 0, \quad (3)$$

i.e., a goal is addressed if and only if, given the trigger condition is true, then the final state must be eventually hold true somewhere on the subsequent temporal line. Some examples of goals for the smart space case study are in the Appendix.

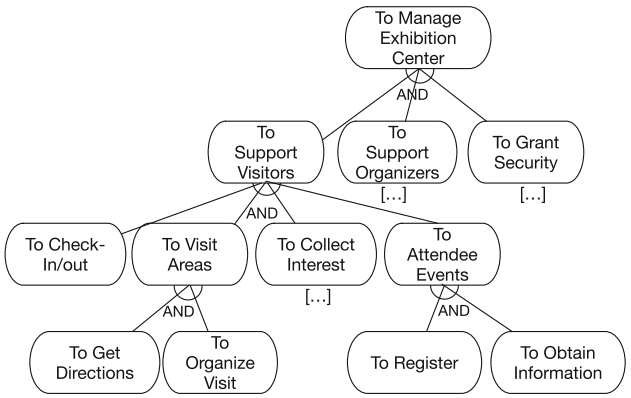

Fig. 1 Portion of goal model for the exhibition center

Definition 3 (*Goal model*) A goal model is a directed graph, (G,R) where G is a set of goals (nodes) and R is the set of Refinement relations (edges), i.e., relations that provide a hierarchical decomposition of goals is sub-goals through AND/OR operators. In a goal model there is exactly one root goal, and there are no refinement cycles.

This definition has been inspired by [24] but we explicitly removed Influence [24] relations and means–end [15] relations from the definition. The influence relation prescribes a change in the satisfaction level of a goal affects the satisfaction level of its adjacent goal. It is not currently used in our theoretical model, whereas means–end links provide a direct connection between a goal and the procedure the system would engage to address it. They are not in the definition of goal-model because the system generates them at run-time.

Figure 1 is an excerpt of the goal model for the Exhibition Center, resulting from the requirements and drawn according to the *i** notation. The example includes functional (hard) goals and AND decompositions. The root goal, to manage the exhibition center, is decomposed into three sub-goals: to support visitors, to support organizers, and to grant security.

2.2 Proactive means–end reasoning

In many goal-oriented approaches, a Task is defined as the operationalization of a Goal. It means each task, in a goal model, is associated with one (or more) leaf goal(s). This design-time association is the result of a human activity called means–end analysis. In the *i** conceptual model [64], a means–end link introduces *a means to attain an end* where *the end can be a goal, task, resource or soft-goal, whereas the means is usually a task*. The TROPOS methodology [15] introduces means–end analysis as the activity for identifying (possibly several alternatives) tasks to satisfy a goal.

The task is, therefore, an analysis entity that encapsulates how to address a given goal according to the following state-

ment: "a Task T is a means to a Goal G (G being the end) when one or more executions of T produce a post-situation that satisfies G" [34].

This paper introduces the concept of system Capability for highlighting the difference between means–end analysis made at design-time and at run-time.

Definition 4 (*Capability*) A capability $\langle evo, pre, post \rangle$ is an atomic and self-contained action the system may intentionally use to address a given evolution of the state of the world. The evolution, denoted as $evo : W \rightarrow W$, is an endogenous change of the state of the world that takes a state of the world W^t and produces a new state of the world W^{t+1} by manipulating statements in W^t. The capability may be executed only when a given pre-condition is true ($pre(W^t) = true$). Moreover, the post-condition is a run-time helper to check if the capability has been successfully executed ($post(W^{t+1}) = true$).

Explicit differences between the concepts of Capability and Task, will be discussed in the following. Some examples of capabilities for the smart space case study are in the Appendix.

Capabilities and goals Whereas a task has an explicit link to a goal, a capability is relatively independent of a specific goal. The concept of capabilities raises up as the attempt to provide goal-models at run-time (goal-injection) that do not contain tasks. The system is assumed to own a repository of capabilities to be used for addressing one of the injected goals.

The connection between Capabilities and Goals relies on the enclosed semantics. To evaluate if a capability may satisfy a goal, the system generates and tries to solve a system of equations obtained by the current state of the world, the capability's pre/post conditions, and goal's trigger/final state. Given W^k, $c_j = \langle evo_j, pre_j, post_j \rangle$ will address $g_i = \langle tc_i, fs_i \rangle$ iff:

$$\begin{cases} s = true, \forall s \in W^k \\ tc_i(W^k) = true \\ pre_j(W^k) = true \\ evo_j(W^k) = W^{k+1} \\ post_j(W^{k+1}) = true \\ fs_i(W^{k+1}) = true \end{cases} \tag{4}$$

This problem can be easily translated, through predicate resolution, into a boolean satisfiability problem [9] whose details are out of the scope of this paper.

Composition of capabilities To increase the variability of system behavior this work assumes that it is convenient to decompose functionality in its atomic (but self-contained) components. It is the contextual composition of these parts that may produce a range of possible results. For this reason, capabilities are composable entities.

Their composition is not specified in a design-time model, but it can be deducted at run-time by checking the satisfiability of pre and post conditions [9] . When capabilities are composable, then system of Eq. 4 changes for including the resulting evolution function as the sum of each single capability evolution.

Parametric capabilities a task is arranged for a particular working context and, therefore, it is scarcely reusable. Conversely, a capability is conceived with the objective of being reusable as much as possible.

To this aim a capability may be 'parametric' i.e., it may specify some input/output ports. As a consequence pre/post and evolution expressions contains some logical variables. The robotic-style capability for moving an physical object is an example of parametric capability; its pre-condition is $at(X_1, Y_1)$ whereas the post-condition is $moved\_to(X_2, Y_2)$ where X_1, X_2 and Y_1,Y_2 must be specified for making the action concrete.

Intuitively, depicting the space of solutions as a Cartesian plane where points represent states of the world, a Capability may be intuitively expressed as a vector that induces a movement from a state A to a state B. A parametric capability is drawn as a family of vectors where the initial state and the final state are subject to variability. The strength of parametric capabilities is that they could be used in different circumstances and they are more versatile in compositions.

According to the principle that capabilities have not an explicit link to goals, the proposed approach is based on delegating to the system the responsibility to establish which capability to select (in alternative, which composition of capabilities to compose) and to configure its parameters for addressing a given goal.

Definition 5 (*Operationalization*) The Operationalization is defined as the tuple $\langle g, h \rangle$ where g is the goal to address and h is the instance of a simple or composed capability, assigned for making the goal operational, where all parameters have been assigned to a ground value.

Setting the operationalization of a whole goal model is a problem formalized as follows:

Problem 1 (*Proactive means–end reasoning*) Given the current state of the world W_I, a Goal Model (G, R) and a set of available Capabilities C, the *Proactive means–end Reasoning* is the problem of finding a complete and minimal set of operationalization for the goal model.

We denote with *Configuration* a solution to the Proactive means–end Reasoning problem. A Configuration is therefore a set of tuple $\langle g_i, h_j \rangle$ where $g_i \in G$ and h_j may be a simple or composed capability.

Given a goal model (G, R), a configuration cnf is said to be

$$complete \text{ iff } \forall g_i \in G, \exists h_j : \langle g, h \rangle \in cnf; \text{ otherwise it is } partial; \quad (5)$$

$$minimal \text{ iff } \forall g_i \in G, \nexists h_k, h_r : \langle g_i, h_k \rangle \in cnf, \langle g_i, h_r \rangle \in cnf. \quad (6)$$

It is worth noting the following:

1. Next sections are going to illustrate an approach for solving Problem 1; for the sake of clarity we use the following terminology: *Proactive means-end Reasoning* is a shorthand for Problem 1, whereas *PMR ability* refers to the algorithm for solving the problem;
2. Problem 1 is different from a scheduling problem since it does not require an exact timing of the activities and it is different from a planning problem because it does not require to create a workflow for executing the activities [27];
3. when solving the Proactive means–end Reasoning problem, discovering more configurations produces an additional value for the purpose of adaptation. Indeed, it allows comparing them according to meta-properties (for instance the quality of service). This is possible under the assumption that C is a redundant set of capabilities, and, therefore, it is possible to replace a capability either with other simple or with composed ones. Indeed, redundancy represents the common operative context for several works in the area of self-adaptive systems [24,43,47].

3 Solving the problem at the knowledge level

In this section, we introduce an approach to Problem 1 that is based on the concept of *state of the world* to model a dynamic knowledge base.

We make the assumption that the solution to the Proactive means–end Reasoning problem should not depend on the actual data of the environment, but rather its flow of operations and interactions depend on how the data are represented in abstract form.

Reasoning at the knowledge level [45], it is possible to represent complex abstract data that are instantiated only at run-time. This simplifies the problem by only modeling those features of the environment that are relevant for the execution (properties to monitor and environment entities to manipulate).

In order to make the algorithm affordable we obtain the knowledge level automatically from specifications of goals and capabilities. Therefore, evaluating the contextual fulfilment of goals and the compatibility of capabilities in composition may be done through symbolic checking techniques.

The proposed approach for implementing a PMR Ability uses a two-step strategy that combines a top-down 'divide' method with a bottom-up 'merge' method.

The top-down goal decomposition explores a hierarchy by decomposing the problem space into smaller disjoint subspaces according to the structure of the goal model and available capabilities. Then it uses a STRIPS-based [28] approach for bottom-up composition of simpler capabilities into more complex ones.

3.1 Top-down goal decomposition

Given a goal model (G, R) where $g_{root} \in G$ is the top goal of the hierarchy, the first step of the proposed procedure is to explore the hierarchy of goals, starting from g_{root} in a top-down recursive fashion. The algorithm exploits AND/OR decomposition relationships to deduct the addressability of a goal according to its sub-goals. The objective is to obtain at least a complete configuration that addresses the problem. However, when possible, it will return a set of alternative configurations.

Let us indicate with $cnf_i = (o_1, o_2, \ldots, o_n)$ a complete/partial configuration for the fulfillment of the goal model where $o_i = \langle g_i, h_i \rangle$ are the operationalizations. Therefore, we use the following notation for indicating a generic solution_set generated by the algorithm: $\{cnf_1, cnf_2 \ldots cnf_k\}$.

For instance, $\{(\langle g_A, h_1 \rangle), (\langle g_B, h_2 \rangle)\}$ indicates a solution_set made of two configurations, each one composed by only one operationalization. Conversely, $\{(\langle g_C, h_3 \rangle, \langle g_D, h_4 \rangle)\}$ represents a solution_set that contains only one configuration, made of a couple of operationalizations.

The first step of the algorithm is to check if a goal is either a leaf or it is decomposed into sub-goals.

When the goal is not a leaf, if the relationship is an AND decomposition the result is the permutation of all the solutions found for each children node. Example: if a goal g_A is AND-decomposed in two sub-goals g_B and g_C, and the algorithm finds

ALGORITHM 1: Means End Reasoning (part I - exploring goal hierarchies)

Input: GM is the goal-model to address, g_{target} is the goal analyzed at this step of the procedure, W_I is the current state of the world and C is the set of available capabilities.

Output: The set of solutions sol_set.

```
Function means_end_reasoning(GM, g_target, C) begin
    if g_target is leaf then
        h_set ← compose_capabilities(g_target, W_I, C);
        foreach h_i ∈ h_set do
            add_solution(sol_set, ⟨(g_target, h)⟩);
        end
    else
        dec_type ← get_decomposition_type(g_target, GM);
        subgoals ← get_subgoals(g_target, GM);
        foreach g_i ∈ subgoals(g_target) do
            sub_sol ← means_end_reasoning(GM, g_i, C);
            if dec_type is AND then
                sol_set ← permutation(sol_set, sub_sol);
            else if dec_type is OR then
                sol_set ← union(sol_set, sub_sol);
            end
        end
    end
    return sol_set
end
```

$$\begin{cases} sol\_set_B = \{(\langle g_b, c_1\rangle), (\langle g_b, c_2\rangle)\} \\ sol\_set_C = \{(\langle g_c, c_3\rangle)\}, \end{cases} \tag{7}$$

then the composed solution of g_A is

$$sol\_set_A = \{(\langle g_b, c_1\rangle, \langle g_c, c_3\rangle), (\langle g_b, c_2\rangle, \langle g_c, c_3\rangle)\} \tag{8}$$

If the relationship is an OR decomposition the result is the union of all the solutions found for each children node. Example: if a goal g_A is OR decomposed in two sub-goals g_B and g_C, and the algorithm finds

$$\begin{cases} sol\_set_B = \{(\langle g_b, c_1\rangle), (\langle g_b, c_2\rangle)\} \\ sol\_set_C = \{(\langle g_c, c_3\rangle)\}, \end{cases} \tag{9}$$

then the composed solution of g_A is

$$sol\_set_A = \{(\langle g_b, c_1\rangle), (\langle g_b, c_2\rangle), (\langle g_c, c_3\rangle)\} \tag{10}$$

Otherwise, when the target goal is a leaf goal then it is necessary to search for a capability or a composition of capabilities that is able to satisfy such a goal. This procedure is discussed in the next section.

3.2 Bottom-up capability composition

A capability produces a state of the world evolution. In the same way, the composition of capabilities produces a multi-step world evolution. The capability composition is a procedure that explores the potential impact of a sequence of capabilities with respect to the initial state of the world and the desired goal to address.

The outcome of composing capabilities is modeled as a state transition system where nodes are states of the world and transitions are due to component capabilities:

Definition 6 (*State of the world transition system*) A State of the World Transition System (WTS) is a 5-tuple $\langle S, W_I, C, E, \mathcal{L}\rangle$ where

- S is the finite set of reachable states of world;
- $W_I \in S$ is the initial state of the world;
- C is the finite set of available capabilities;
- E is the transition relation made as a finite set of evolution functions where $evo \in E : W \times W$
- $\mathcal{L} : S \rightarrow Score$ is the labeling function that associates each state to a score that measures (1) the distance from the final state and (2) the quality of the partial paths and therefore it estimates the global impact in satisfying the whole goal-model.

ALGORITHM 2: Means End Reasoning (part II - composing capabilities)

Input: GM is the goal-model to address, g_{target} is the goal for which we want to find a capability or a composition of capabilities, W_I is the current state of the world and C is the set of available capabilities.

Output: h is a capability or a composition of capabilities that satisfies g_{target}.

```
Function compose_capabilities(g_target, W_I, C) begin
    WTS ← initialize_space(W_I);
    while |h_set| < max_h_set AND |WTS| < max_space do
        W_i ← get_highest_scored_state(WTS);
        CS ← path_from_to(WTS, W_I, W_i);
        if check_cs_is_solution(CS, g_target) then
            add_solution(h_set, CS);
            mark_as_solution(WTS, CS);
        else
            cap_set ← get_next_capabilities(W_i, CS, WTS);
            expand_and_score(WTS, W_i, cap_set);
        end
    end
    return h_set
end
```

The procedure for incrementally building the WTS is reported in Algorithm 2. The inputs of the algorithm are the current state of the world W_I, a generic goal $g_{target} \in G$ of the goal model, and the set of available capabilities C. The objective is to explore the endogenous effects of combinations of capabilities with the aim of addressing g_{target}.

At each step the algorithm gets most promising state of the world W_i to explore (this is evaluated through a score that is discussed later in this section). Then it extracts the CS as the shortest sequence of capabilities that produces the evolution from W_I to W_i.

First, it checks if CS satisfies the goal g_{target} according to Eq. 3. In other words, given the Triggering Condition and the Final State of the goal, the sub-procedure *check_cs_is_solution* explores the evolution sequence to check if both TC and FS are satisfied by states of the world and if FS=true occurs after that TC=true (see Fig. 2). In the case CS satisfies the goal then the capability sequence represents a solution and it is added to the h_set.

Conversely, the procedure selects a set of capabilities that may be used to expand the WTS. The first criterion to select capabilities filters those that may be executed in W_i: i.e., it considers only capabilities whose pre-conditions are true in W_i:

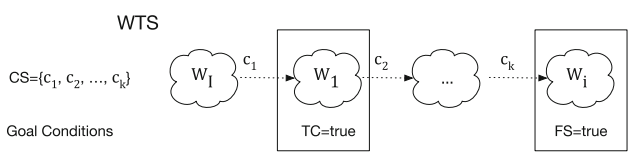

Fig. 2 Illustration of the procedure for evaluating the satisfaction of a goal along a state of the world evolution. First the algorithm searches for a state of the world in which $TC = true$. After that, it proceeds searching for a state of the world in which $FS = true$

$$cap\_set' = \{\langle evo, pre, post\rangle \in C | pre(W_i) = true\} \tag{11}$$

However this set may be further restricted to exclude irrelevant capabilities that do not produce significant changes into the state of the world:

$$cap\_set = \{\langle evo, pre, post\rangle \in cap\_set' | \\ evo(W_i) \cap \{W_I, W_1, \ldots, W_i\}\} \tag{12}$$

Finally, the sub-procedure *expand_and_score* for each $c_i \in cap\_set$ creates a new transition in the WTS from W_i to the new state of the world $evo_{c_i}(W_i)$. The generated states of the world are subsequently labeled with the score function.

The score function provides an indication of *quality* of a sequence of states of the world $seq = \{W_I, W_1, \ldots W_i\}$ with respect to the goal to address, and, therefore, it measures how promising is the corresponding sequence of capabilities CS. The score function has been designed to drive the algorithm to explore combinations that are more promising for the satisfaction of the goal, decreasing at the same time the size of the explored space. For instance, a sequence of states in which $TC = true$ is more interesting than one where $TC = false$.

Following this idea, given that a state of the world is made of statements, it is necessary to introduce the principle that each of these statements may provide (or not) a contribution for asserting a goal is satisfied. For instance if the goal is *to print and send a document*, the statement *printed(doc)* could produce a positive impact to the goal. According to this observation, we state two principles for comparing states of the world obtained by capability composition:

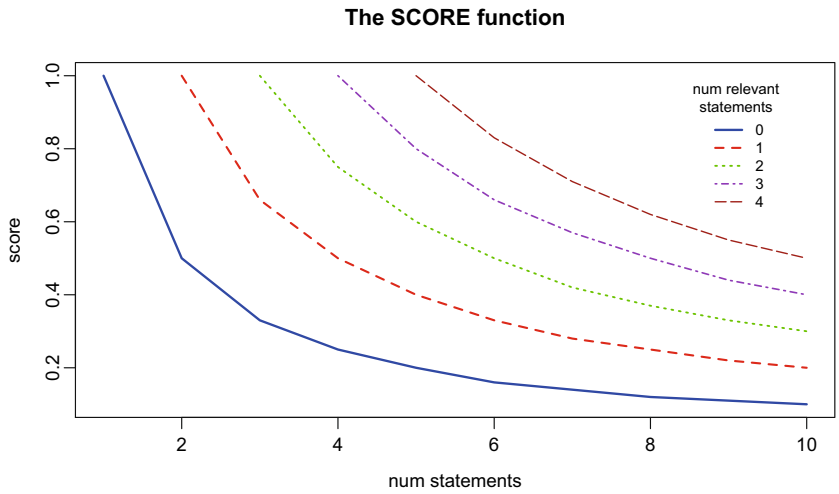

Fig. 3 Line chart of the score function highlights trends of the value when making either $num\_statements(W)$ or $num\_relevant\_statements(W, g)$ constant

– the *principle of convergence*, i.e., the more a state of the world contains statements that provide a positive impact to a goal, the more the solution is near to be *complete* for addressing it, according Eq. 5;
– the *principle of precision*, i.e., the more a state of the world contains statements that do not provide a positive impact to a goal, the more it is *minimal* for addressing it, according Eq. 6;

As a consequence we can specify the function as follows:

$$score(W_i, g_{target}) = \frac{1 + num\_relevant\_statements(W_i, g_{target})}{num\_statements(W_i)} \quad (13)$$

where, given a state W, $num\_statements(W)$ is the cardinality of W, i.e., the number of statements contained in W, whereas $num\_relevant\_statements(W, g)$ is defined as the number of statements contained in W that positively contribute to make $TC_g \wedge FS_g = true$. For instance, if $W = \{s_1, s_2, s_3, s_4, s_5\}$ and $g = \langle s_2 \wedge s_8, s_4 \vee s_5 \rangle$, then $num\_statements = 5$ and $num\_relevant\_statements = 3$ because $\{s_2, s_4, s_5\}$ are relevant for g.

Figure 3 illustrates Function 13 plotted as a stacked line chart for highlighting the score trends. Making the $num\_relevant\_statements$ constant, the value increases when the total number of statements in W_i decreases (*principle of precision*). Therefore, a state of the world that contains fewer statements is considered more promising than another that contains more statements.

At the same time, making the $num\_statements$ constant in the formula, the value is higher the more the state is close to goal satisfaction (*principle of convergence*). It means that a state of the world that contains statements relevant for a goal is considered more promising than another that does not contain relevant statements.

The algorithm terminates when a pre-defined number of solutions has been discovered, or after a maximum number of states of the world has been explored.

4 A general architecture for self-adaptation

This section illustrates how the PMR Ability may be the basis for a domain-independent self-adaptive software system. This section discusses the relationship between the approach presented in this paper and three fundamental characteristics for a self-adaptive system: system evolution, self-configuration, and self-healing [18,37]. More information about the customization of the architecture for a smart space case study is in the Appendix.

4.1 System evolution

Software evolution is a discipline of software engineering that aims at modifying existing software for ensuring its reliability and flexibility over time.

In particular, we focus on adaptive maintenance [17] an aspect of software evolution that refers to modification performed to keep software usable in a dynamic environment. The real-world changes continuously and, therefore, user's needs evolve. Software that runs in an environment is likely to continually adapt to varying circumstances. This is translated into functional enhancement and into the improvement of performances to reflect requirements evolution.

A prominent characteristic of the proposed architecture is to handle the run-time addition of new requirements and, therefore, to amount to system evolution [56,58]. The PMR Ability allows moving a step forward traditional system defined for satisfying a fixed set of hard-coded requirements. It allows adding or changing requirements during run-time (in the form of goal-models). We called this mechanism *Goal*

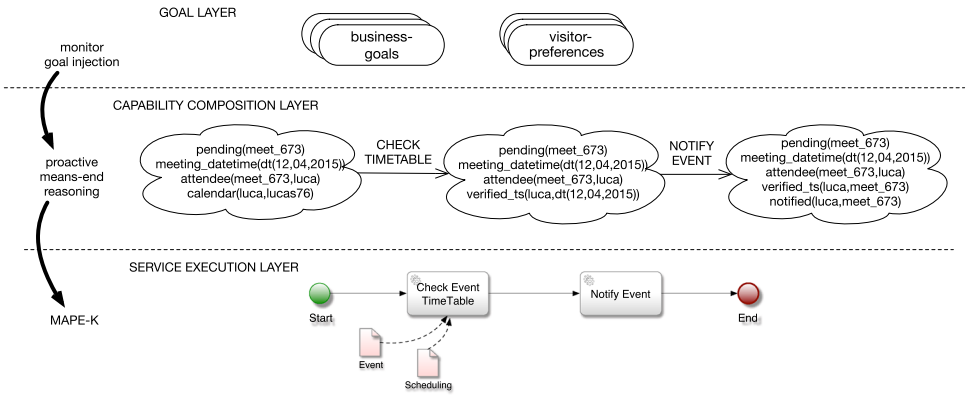

Fig. 4 Overview of the three-layer architecture for self-configuration

Injection [56]. The user may specify new requirements to inject into the system at run-time, and they become a stimulus for modifying its behavior. It is a responsibility of the system via the PMR Ability to adapt itself to the new needs. The goal injection is enabled by two components:

- on the one hand, the system owns a *goal injection monitor* that waits for goals from the user;
- on the other hand, user-goals are run-time entities, as well as other environmental properties. The system acquires goals from the user and maintains knowledge of them thus to be able to reason on expected results and finally conditioning its global behavior. Of course, existing goals may retreat as well.

Goal injection enables *user-requirements to evolve over time* [35] without either user-management or restarting the system. It could be fundamental for some categories of domains in which continuity of service is central (financial, service providing and so on).

In addition, it is possible *to increase or to enhance the functions of the system* just injecting a new set of requirements and updating the repository with new domain-specific capabilities. Given that connections between goals and capabilities are discovered on demand, the architecture is robust to evolution and it may be used for different problem domains with little customization.

4.2 Self-configuration

Self-configuration is the ability of the system to automatically set up the parameters of its components thus to ensure the correct functioning with respect to the defined requirements [13,37,48].

This subsection shows a three-layer architecture that exploits the PMR Ability for generating business logic for requirement fulfillment. In other words, the proposed architecture implements self-configuration intended as the ability of as system to autonomously (without explicit management) select and compose a subset of its capabilities to achieve user's goals.

The operative hypothesis is to consider the system owns a repository of capabilities. This set is redundant, i.e., to solve the same problem the system may exploit different combinations of capabilities. Some of these capabilities have input/output parameters that must be configured to be used.

The proposed architecture is made of three layers (Fig. 4): the goal layer, the capability composition layer, and the service execution layer.

The uppermost layer of this architecture is the *Goal Layer*, in which the user may specify the expected behavior of the system in terms of high-level goals, according to Definition 2. Goals are not hard-coded in a static goal-model defined at design time. The goal injection phase allows the introduction of user-goals defined at run-time. Goals are interpreted and analyzed and, therefore, trigger a new system behavior.

The second layer is the *Capability Composition Layer*, based on the problem of Proactive means–end Reasoning. It aims at selecting capabilities and configuring them as a response to requests defined at the top-layer. It corresponds to a strategic deliberation phase in which decisions are taken, according to the (incomplete) knowledge about the environment. However, this layer does not reason on concrete data, and it does not consider possible changes in the environment because it would be very costly from a computational perspective.

Algorithm 2 is explicitly built for self-configuration; indeed, in the meanwhile a $Configuration$ solution is discovered, it searches for dependencies among the capabilities that are selected and it also resolves these dependencies by connecting their input/output ports. The consequent output is a concrete business process obtained by instantiating capabilities into task and data into data objects. In this phase, the procedure also specifies dependencies among tasks and how data items are connected to task input/output ports.

The third layer is the *Service Execution Layer* that executes the business process generated at the second layer. It consists of atomic blocks of computation, used for acquir-

ing and analyzing data from the environment and to act for producing the desired state of the world. This layer is implemented as a MAPE-K model [16,49], well known in the literature. It requires (1) a Monitoring component that acquires information from the environment, and it updates the system knowledge accordingly; (2) an Analyze component that uses the knowledge to determine the need for adaptation with respect to expected states of the world or capabilities failure; (3) a Plan component that uses the acquired knowledge to synchronize the available capabilities according to the goal hierarchy and, finally, (4) an Execute component that modifies the environment by using the appropriate capability.

4.3 Self-healing

Self-healing is the ability of the system to automatically discover whenever requirements fail to be fulfilled and to work around encountered problems, thus to restore fulfillment of the requirements and to grant continuous functioning with respect to the defined requirements [36,37].

In the previous section, we have adopted the MAPE-K model [16,49] for implementing the business layer of the presented architecture. According to the roadmap of self-adaptive systems [18], one of the principles for implementing self-healing is to explicitly focus on the 'control loop', to be used as an internal mechanism for controlling the system's dynamic behavior. The most famous control architecture is the MAPE-K model and we propose to place the PMR Ability on top of the MAPE-K architecture to generate a macro-loop for self-healing, as shown in Fig. 5. The macro activities of the resulting architecture are *monitor goal injection*, *proactive means–end reasoning* and *MAPE-K loop*.

In the *Goal injection* phase the user communicates her requirements to the system. The system reacts to the injection of new goals by activating the *PMR Ability* to assemble a solution for addressing the whole goal model, and if at least one solution is discovered, then the system selects the highest scored $Configuration$ and instantiates the corresponding business process, reserving proper resources for its execution. At this stage, it is impossible to predict all possible changes in the environmental conditions.

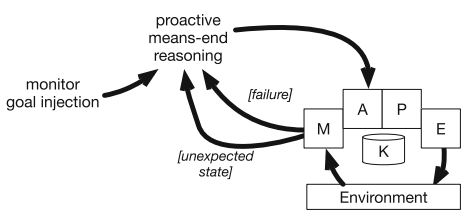

Fig. 5 Graphical representation of the self-healing loop

Therefore, the agent activates a sub-cycle of monitoring, analyzing, plan and execution driven by the knowledge of the environment (*MAPE-K*). If everything goes as planned, the goal will eventually be addressed. However, given that the Algorithms 1 and 2 do not consider exogenous changes of the state of affairs, it is possible that unexpected events occur in the environment, during the execution. When system's monitors capture an unexpected state of the world, and the capabilities in the $Configuration$ are not sufficient to deal with that, then the system recognizes a situation of failure for one of the requirements. This raises *a need for adaptation* event and the PMR Ability executes again with a different W_I (the current one). The result will be a different $Configuration$ (if possible) for overcoming the unexpected state. The self-adaptation cycle also considers cases in which the execution of a capability terminates with errors. In this case, the PMR Ability is re-executed with the shrewdness to mark the capability that failed as 'unselectable'.

5 Evaluation and discussion

The architecture, presented in Sect. 4, has been implemented in MUSA, a Middleware for User-driven Service Adaptation [23]. MUSA is built as a multi-agent system and developed in JASON [12], a declarative programming language based on the AgentSpeak language [51] and the BDI theory [14]. The state of an agent together with its knowledge of the operative environment is modeled through its belief base, expressed by logical predicates. Self-awareness is supported by translating high-level goals' and capabilities' specifications into agent's beliefs [55]. This enabled the development of the agent PMR Ability for reasoning on Goals and Capabilities as first class entities [23,54]. Additional details on MUSA are provided in the Appendix.

The rest of this section presents and discusses an evaluation benchmark for MUSA in the context of self-configuration and self-healing.

5.1 Evaluating self-configuration

The proposed architecture relies on a couple of algorithms for analyzing the goal model and exploring the space of solutions for composing capabilities. The latter algorithm incrementally builds a state transition system where each edge is generated through the evolution function of a capability and each node is a possible state of the world. The state transition system takes the form of a tree where each branch is a different partial/complete configuration for the fulfillment of a given goal. Exploring the whole space of solutions would take an exponential time to complete. However, the score function has been designed to drive the order of exploration, thus exploring first most promising directions.

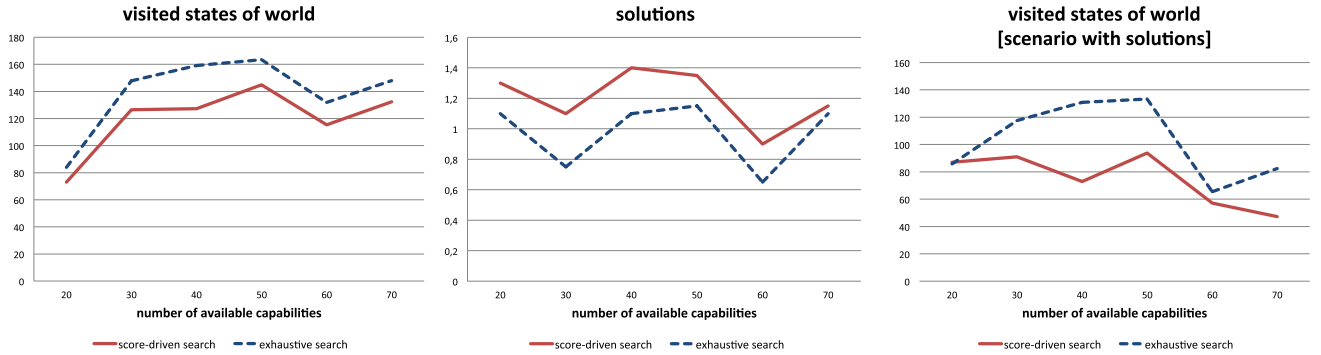

Fig. 6 Data obtained by comparing the algorithms presented in Sect. 3 with a breath-first strategy. Configurations are the result of 120 executions with random input and increasing of 10 the number of capabilities every 20 runs

Here we present the methodology we adopted to generate sequences of stress test to evaluate the algorithms with respect of self-configuration and self-healing.

1. Random generation of a working context: this step consists in randomly extracting a fixed number of statements from a repository. That context represents the dictionary of terms describing an abstract working context.
   Example: $Dictionary = [b(e), q(u), l(a), g(a), v(o), z(u), r(u), z(a), v(i), d(e)]$.
2. Random generation of goals to satisfy: each goal is generated by randomly selecting terms from the dictionary.
   Example: $goal(''g38'', condition(not(z(u))), condition(z(a)))$ triggers when the state of the world does not contain the statement $z(u)$ and it is addressed when the state of the world does contain $z(a)$.
3. Random generation of the current state of the world: picking an arbitrary number of statements from the dictionary generates a random W_I. Example: $world([r(u)])$.
4. Finally, random generation of a repository of capabilities. Each capability is produced by selecting couples of terms from the dictionary. The first term is the pre-condition and the second term is the post-condition. The evolution function is built consequently.
   Example: $cap(''c1'', evo([remove(r(u)), add(z(u))]), condition(r(u)), condition(z(u)))$.

For operating a comparative benchmark we selected (1) the couple of algorithms presented in Sect. 3 in which capabilities are filtered (see Equations 11 and 12) and WTS nodes are scored (thereafter "score-driven search") , and (2) the same algorithms where the score function is replaced with a breath-first strategy (thereafter "exhaustive search").

Therefore, we ran a series of tests with an incremental number of capabilities, starting from 20, until 70. Each test executes both the score-driven search and the exhaustive search with the same input. We measured the number of visited nodes in the WTS and the number of discovered solutions. Charts of Fig. 6 reports the results obtained by repeating the test 120 times, starting from 20 capabilities and increasing of 10 after every 20 runs. We used a paired t-test for verifying that *visited nodes* (obtained through the two methods) are significantly different ($p\ value = 0.01$) (Table 1).

Table 1 Analysis of means (t test) of *Visited States of World* obtained by the two methods

	Name	Mean	Median	SD	p value	Effect size
1	Score	128.23	200	86.04		
2	Breath	148.80	201	83.76		
3	Difference	−20.57	−1	49.50	0.01	−0.42

The number of visited nodes (and, therefore, the time-to-complete) is polynomial, compared the number of capabilities in both the score-driven search and the exhaustive search (see 'visited states of the world' in Fig. 6). To some extent, this was surprising because we expected an exponential time, given the algorithm is in the class of combinatorial search. A deeper analysis shows that the activity of capability filtering (Eqs. 11 and 12), done at each step of the algorithm, greatly reduces the space of evolution and therefore state explosion is limited.

Figure 6 reveals that the exhaustive search represents an upper boundary for the score-driven algorithm for what concerns performance. Indeed the score-driven search provides better results, in the *number of visited nodes*, and for what concerns the *number of discovered solutions*.

We also noted that taking in consideration only those settings in which at least one solution exists, the average number of nodes visited through the score function is definitively better than an exhaustive search strategy (see 'scenario with solutions' in Fig. 6).

5.2 Evaluating self-healing

For evaluating this property, we have added other three items to the previous methodology for testing.

Fig. 7 Result of the sequence of tests for self-healing. The *dark gray area* represents the size of the space of solutions discovered at the first run of self-configuration for each scenario. The *light gray area* represents the additional space of solutions built as a result of self-healing

5. Execute the PMR Ability with the input obtained at previous steps and select one output configuration.
6. Simulate the execution of the configuration and randomly generate an adaptation event.
7. Update the initial state of the world to the current situation at the moment of failure and execute again point 5.

Therefore, we ran a sequence of tests with a fixed number of 40 capabilities, measuring the number of solutions discovered: i) at the first run of self-configuration and ii) after the self-healing.

Figure 7 represents as filled areas the space of configurations obtained by executing the PMR Ability before and after the self-healing event. Among the 13 scenarios, the adaptation failed only in three cases (scenarios 3,4 and 13). The cause was the available capabilities, not enough to repair the failure. In all the other cases the procedure performed well, increasing the space of configuration to allow the goal fulfillment.

As a final note, we calculated that new configuration, obtained for overcoming a failure, in average reuses the 72.25% of capabilities used in the first configuration.

5.3 Related works

This section describes similar approaches to implement self-adaptation.

The architecture we have presented in Sect. 4 is based on the concept of goals. Other works in the literature adopt similar run-time elements. Morandini et al. [43,44] propose an approach based on goal models extended with an operational semantics for specifying their dynamics and, at the same time, maintaining the flexibility of using different goal types and conditions.

Dalpiaz et al. [24] propose a new type of goal model, called runtime goal model (RGM). It extends the former with annotation about an additional information about the fulfillment of goals. For instance, it explains when and how different instances of the same goals and tasks need to be created. The common element of these couple of approaches is that the behavior of the system is wired into tasks that in turn are wired to goals of the model. Therefore even if the system may select many alternative OR decomposition relationships, it can adapt its behavior but it can not evolve over the pre-defined tasks.

Baresi et al. [6] introduce the concept of adaptive goals as means to conveniently describe adaptation countermeasures in a parametric way. An adaptive goal is described as an objective to be achieved, a set of constraints and a sequence of actions to fulfill the aforementioned objective. The same author proposes A-3 [5], a self-organizing distributed middleware aiming at dealing with high-volume and highly volatile distributed systems. It focuses on the coordination needs of complex systems, yet it also provides designers with a clear view of where they can include control loops, and how they can coordinate them with the aim of global management. As well as our approach they consider requirements as run-time entities even if they do not propose a dynamic execution model in which their goals are injected at run-time. Also, they introduce fuzzy goals for expressing the satisfaction degree of requirements that is a possible future direction for extending our definition of goals.

SAPERE [65] (Self-Aware Pervasive Service Ecosystems) is a general framework inspired from natural self-organizing distributed ecosystems. SAPERE does promote adaptivity by creating a sort of systemic self-awareness. As well as our approach, their components have, by design, an associated semantic representation. These live semantic annotations are similar to service descriptions and enable dynamic unsupervised interactions between components.

Gorlick et al. [31] present an approach to handle runtime change called Weaves. A weave is an arbitrary network of tool fragments that communicate asynchronously. Similar to our concept of capability, a tool fragment is a small software component that performs a single, well-defined function and may retain state.

Blanchet et al. [11] present the WRABBIT framework that supports self-healing for service orchestration through conversation among intelligent agents. Each agent is responsible for delivering services of a participating organization. Globally they can discover when one agent's workflow changed unilaterally because it may incur conversation errors with other agents. An agent also recognizes mismatches between

its workflow model and the models of other agents.The limit of such approach is that it is domain oriented since the possible errors must be defined at design-time. Extending the WRABBIT's approach for handling unexpected *not-understood* situations could be an interesting direction for our work.

Kramer and Magee [38] propose a three-layer architecture for self-adaptation inspired from robotics. The architecture includes (1) a control layer, i.e., a reactive component consisting of sensors, actuators and control loops, (2) a sequencing layer which reacts to changes from the lower levels by modifying plans to handle the new situation and (3) a deliberation layer that consists in time consuming planning which attempts to produce a plan to achieve a goal. The main difference with our architecture is that we introduce a layer for handling goal evolution.

Gomaa and Hashimoto [30], in the context of the SASSY research project, look into software adaptation patterns for Service-Oriented applications. Their intuition is that dynamic reconfiguration can be executed by assembly architectural patterns. The objective is to dynamically adapt distributed transactions at run-time, separating the concerns of individual components of the architecture from concerns of dynamic adaptation, using a connector adaptation state-machine. Like our approach, SASSY provides a uniform approach to adaptive software systems, however, to date, goals evolution is out of the scope of their work.

Souza et al. [60] focus on evolution requirements that play an important role in the lifetime of a software system in that they define possible changes to requirements, along with the conditions under which these changes apply.

Ghezzi et al. [29] propose ADAM (ADAptive Model-driven execution), a mixed approach between model transformation techniques and probability theory. The modeling part consists in creating an annotated UML Activity diagram whose branches can have a probability assigned, plus an annotated implementation. Then an activity diagram becomes an MDP (Markov Decision Process). It is possible to calculate the possible values for the different executions and thus to navigate the model to execute it.

The MUSIC middleware [53] provides a self-adaptive component-based architecture to support the building of systems in ubiquitous and SoA environments where changes may occur in service providers and service consumers contexts. Applications are assembled through a recursive composition process. The middleware uses utility functions to calculate utility scores for each application variant. The highest utility score indicates the most suitable variant for the current context and it is selected for adaptation.

5.4 Strengths, weakness, and future works

The main strengths of the proposed architecture are summarized below:

Reusability capabilities support the paradigm of Full-Reuse [7]. Capabilities are atomic, self-contained, and created for being composed. They must be designed for being usable in several contexts, and parameters are the key to achieve a finer tuning for a specific problem. Self-configuration is obtained by handling any change by reusing available capabilities. In practice, capabilities are the key element of reuse.

Support for evolution the approach relies on the idea that goals, capabilities, and their links are not hard-coded. Indeed goals and capabilities are decoupled, and goals are injected at run-time. The dynamic connection between capabilities and goals must be discovered at run-time. In addition, the repository of capability can be evolved without restarting the system.

Domain independence Working at the knowledge level, the problem is modeled through those features of the environment that are relevant for the execution (elements to monitor and to manipulate). The adopted solution is to enclose all the necessary semantics into goals and capabilities. The PMR Ability does not require further information for producing a configuration. The proposed architecture exploits general representation of knowledge for reasoning about capabilities that is independent of the particular application that is driving it [50]. Therefore, it is possible to *translate from a domain to another one* just injecting a new set of requirements and updating the repository with new domain-specific capabilities. The same architecture may serve different problem domains, even at the same time, without any other specific customization.

Concluding, a critical analysis of the approach highlights some issues that could be the starting point for improving the proposed architecture.

In this approach, as well in state-change models [28], actions are instantaneous, and there is no provision for asserting what is true while an action is in execution (transitory). Such systems cannot represent the situation where one action occurs while some other event or action is occurring [3]. As a future work, we intend to extend this state-of-world based model towards one that includes times, events, and concurrent actions [3]. For instance, it will be possible to add temporal operators and to test a predicate over some time interval [2,39].

Another point of discussion concerns the real degree of decoupling between Capabilities and Goals. The authors have introduced the use of an ontology for enabling semantic compatibility between these two elements during the Proactive means–end Reasoning.

We already employed MUSA in five research projects with heterogeneous application contexts, from dynamic workflow [56] to a smart travel system [57]. However, in our *in-vitro* evaluation, the same development team created both Capabilities and Goals and thus the ontology commitment was ensured. Our experimental phase is based on the assumption that the ontology is built correctly, thus allowing the system to work properly.

Another interesting aspect to consider is the impact of the maintenance phase over the ontology, and as a direct consequence, the degree of degradation of capabilities. We experienced that even changing the definition of a single predicate in the ontology has a detrimental impact on the reliability of the system in using its capabilities.

6 Conclusion

We have presented a theoretical framework for specifying the problem of Proactive means–end Reasoning regarding states of the world, goals, and capabilities. Solving the problem at the knowledge level provided us the opportunity to define a general architecture for engineering self-adaptive smart spaces. This architecture is based on the idea that a user, at run-time, may inject his goals in a high-level language. The smart space will (re-)configure its services as the result of reasoning and deductions made at the knowledge level. Moreover, system evolution is the result of a process of goals management, obtained through the ability to solve the proactive means–end reasoning. The strengths of the proposed architecture are to be domain independent and to support reusability across many application contexts.

Appendix: engineering the exhibition center with MUSA

MUSA (Middleware for User-driven Service Adaptation) [23] is a multi-agent system for the composition and orchestration of services in a distributed and open environment. It aims at providing run-time modification of the flow of events, dynamic hierarchies of services, and integration of user preferences together with a system for run-time monitoring of activities that is also able to deal with unexpected failures and optimization.

The middleware[1] is coded in JASON [12], a declarative programming language based on the AgentSpeak language [51] and the BDI theory [14]. The state of an agent together with its knowledge of the operative environment is modeled through its belief base, expressed by logical predicates. Desires are states the agent wants to attain according to its perceptions and beliefs. When an agent adopts a plan, it transforms a desire to an intention to be pursued.

In JASON the specification of plans is strictly connected to the desire that triggers its execution. Therefore, we developed a high-level language (GoalSPEC [58]) with the twofold aim of (1) allowing the user to specify requirements in the form of goals and (2) supporting the idea of decoupling *what* the system has to do, and *how* it must do that.

The theory of self-knowledge and action [42] asserts an agent achieves a goal by doing some actions if the agent knows what the action is and it knows that doing the action would result in the goal being satisfied [40]. Therefore, we also defined a high-level language to specify system's capabilities.

Since software agents are deployed in a distributed environment, MUSA implements a distributed version of Algorithms 1 and 2. The knowledge level is supported by goals' and capabilities' specifications that are translated from high-level languages into agent's beliefs [55]. A configuration represents a contract among the agents specifying how to collaborate. Therefore, service composition is obtained at run-time, as the result of a self-organization phenomenon.

In the following, we detail the ingredients needed to achieve our purpose: the way we depict the problem domain using an ontology, a goal specification language that refers to ontological elements as keys for grounding the goals on the problem and, finally, a capability language that supports the separation between the abstract description and the concrete implementation.

6.1 The domain ontology description

Working at the knowledge level implies an ontology commitment between who develops Capabilities and who specifies Goals. An ontology is a specification of a conceptualization made for the purpose of enabling knowledge sharing and reuse [59]. An ontological commitment is an agreement to use a thesaurus of words in a way that is consistent with the theory specified by an ontology [33].

A Problem Ontology (PO) [21,52] is a conceptual model (and a set of guidelines) used to create an ontological commitment to developing complex distributed systems [22]. This artifact aims at visualizing an ontology as a set of concepts, predicates, and actions and how these are related to one another. An example is shown in Fig. 8.

The metamodel of a PO artifact, inspired by the FIPA (Foundation for Intelligent Physical Agents) standard [46], is briefly summarized as follows:

– a *Concept* is a general term commonly used in a broad sense to identify "*anything about which something is said*" [20] that has a unique meaning in a subject domain;

[1] Available, as open source, at https://github.com/icar-aose/musa_2

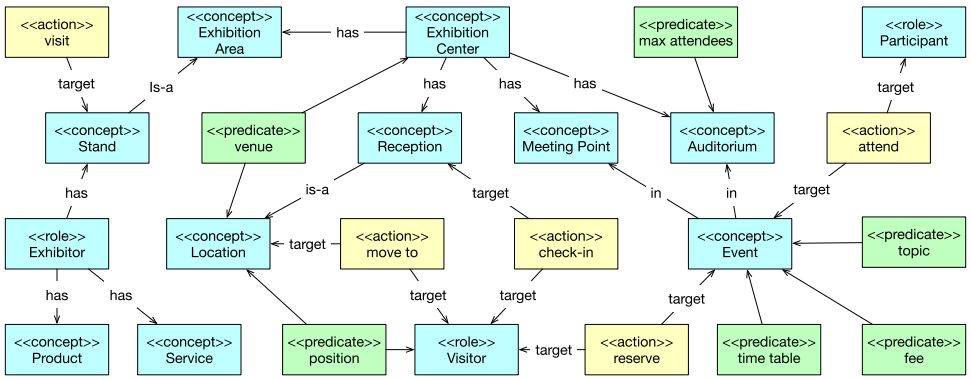

Fig. 8 Example of problem ontology for the exhibition center. Ontology elements represented without stereotypes are to be read as concepts by default

- a *Predicate* is the expression of a property, a quality or a state of one (ore more) concept(s);
- an *Action* is the cause of an event by an acting concept [41]);
- a *Position* is a specialization of concept performing Actions;
- finally, an *Object* represents physical or abstract things.
- the relationship *is-a* (or is-a-subtype-of) defines which objects are classified by which class, thus creating taxonomies;
- the relationship *part-of* (or the counterpart *has-part*) represents the structure by composition;
- the relationship *association* establishes links between ontological elements.

6.2 A goal specification language

The GoalSPEC language [58] has been specifically designed for enabling runtime goal injection and software agent reasoning. It takes inspiration from languages for specifying requirements for adaptation, such as RELAX [62]. However, GoalSPEC is in line with Definition 2 and adopts a domain-independent core grammar with a basic set of keywords that may be extended with a domain ontology.

The main entity of the GoalSPEC language is the *Goal* that is wanted by some Subject and it is structurally composed of a *Trigger Condition* and a *Final State*. It is worth underlining that both Trigger Conditions and Final States must be expressed by using predicates defined in a domain ontology.

Some examples of GoalSPEC productions for the domain of the Exhibition Center are listed below. For a complete specification of the syntax of GoalSPEC, see [58].

1. WHEN registered(Usr,Event) AND NOT attending (Urs,Event) THE system SHALL PRODUCE notification(Event,Usr)
2. BEFORE moving(Urs,Loc) AND exhibition_area(Loc) THE visitor SHALL PRODUCE moving(Urs,Reception) AND check-in(Urs)

We use uppercase for the keywords of the language, and lowercase for domain-specific predicates. Logical variables start with an uppercase letter. Goal 1 indicates that 'if a visitor has registered for an event, the user will be notified about time-table until he goes to the event'. Goal 2 states that 'A condition to enter the exhibition center area is to go to the reception and check-in'.

6.3 A capability specification language

There is an obvious need for a semantic-based language to describe agent capabilities in a common language in a way that (1) the agent knows how to execute the capability and (2) it knows the effects of executing it [42].

So far we use a refinement of LARKS [61] a language for advertisement and request for knowledge sharing used in the context of web services.

A Capability is made of two parts: an abstract description, and a concrete body implementation (a set of plans for executing the job). The abstract description is defined through the following fields: (1) **Name**: unique label used to refer to the capability, (2) **Input/OutputParams**: variables that is necessary to instantiate for the execution, (3) **Constraints**: structural constraints on input/output variables, (4) **Pre/Post Condition**: conditions that must hold in the current state of the world and in the final state of the world and finally, (5) **Evolution**: function $evo : W \longrightarrow W$ as described in Section 2.2.

We do not provide any language for the body, leaving the choice of the specific technology to the developer. We frequently used Java to code this part of our capabilities because of the smooth integration with Jason on the one side and the flexibility in the invocation of web services on the other side.

Figure 9 shows two examples of capabilities. The *Alert Sender* capability that uses smartphones to advert about the event and provide alerts about the timetable. The second capability is the *Cloud Calendar Check* capability that interacts with a calendar application for retrieving information about an event timetable.

Fig. 9 A couple of capabilities described through the capability specification language

Name
ALERT SENDER
InputParams
CONTENT : TEXT, USERMAIL : STRING
OutputParams
NONE
Constraints
$format(UserMail, RFC\_5322\_Address\_Specification)$
Pre-Condition
$email(Usr, UserMail)$
Post-Condition
$notified(Content, Usr)$
Evolution
$evo = \{add(notified(Msg, Usr)), add(mailed(UserMail, Content))\}$

Name
CLOUD CALENDAR CHECK
InputParams
EVENTID : INTEGER, USERCALENDAR : CALENDAR
OutputParams
SLOT: STRING
Constraints
$format(Slot, slot(dt(year, month, day, hour, minute), dt(year, month, day, hour, minute)))$
Pre-Condition
$calendar(Usr, UserCalendar)$
Post-Condition
—
Evolution
$evo = \{add(time\_table(Event, Slot))\}$

References

1. Abeywickrama DB, Bicocchi N, Zambonelli F (2012) Sota (2012) Towards a general model for self-adaptive systems. In: Enabling technologies: infrastructure for collaborative enterprises (WETICE), 2012 IEEE 21st international workshop, IEEE, pp 48–53
2. Allen JF (1983) Maintaining knowledge about temporal intervals. Commun ACM 26(11):832–843
3. Allen JF (1991) Planning as temporal reasoning. KR 91:3–14
4. Augusto JC, Callaghan V, Cook D, Kameas A, Satoh I (2013) Intelligent environments: a manifesto. Hum Centric Comput Inf Sci 3(1):12
5. Baresi L, Guinea S (2011) A3: self-adaptation capabilities through groups and coordination. In: Proceedings of the 4th India software engineering conference, ACM, New York, pp 11–20
6. Baresi L, Pasquale L, Spoletini P (2010) Fuzzy goals for requirements-driven adaptation. In: Requirements engineering conference (RE), 2010 18th IEEE international, IEEE Press, Sydney, pp 125–134
7. Basili VR (1990) Viewing maintenance as reuse-oriented software development. Softw IEEE 7(1):19–25
8. Bennaceur A, France R, Tamburrelli G, Vogel T, Mosterman PJ, Cazzola W, Costa FM, Pierantonio A, Tichy M, Akşit M et al (2014) Mechanisms for leveraging models at runtime in self-adaptive software. Models@ run. time. Springer, pp 19–46
9. Biere A, Heule M, van Maaren H (2009) Handbook of satisfiability, vol 185. IOS Press, Amsterdam
10. Blair G, Bencomo N, France RB (2009) Models@ run. time. Computer 42(10):22–27
11. Blanchet W, Stroulia E, Elio R (2005) Supporting adaptive web-service orchestration with an agent conversation framework. In: Web services, ICWS 2005. Proceedings 2005 IEEE International Conference, IEEE
12. Bordini RH, Hübner JF, Wooldridge M (2007) Programming multi-agent systems in AgentSpeak using Jason, vol 8. Wiley, Hoboken
13. Braberman V, D'Ippolito N, Kramer J, Sykes D, Uchitel S (2015) Morph: a reference architecture for configuration and behaviour self-adaptation. arXiv:1504.08339
14. Bratman ME, Israel DJ, Pollack ME (1988) Plans and resource-bounded practical reasoning. Comput Intell 4(3):349–355
15. Bresciani P, Perini A, Giorgini P, Giunchiglia F, Mylopoulos J (2004) Tropos: an agent-oriented software development methodology. Auton Agents Multi-Agent Syst 8(3):203–236
16. Brun Y, Serugendo GDM, Gacek C, Giese H, Kienle H, Litoiu M, Müller H, Pezzè M, Shaw M (2009) Engineering self-adaptive systems through feedback loops. Softw Eng Self-Adapt Syst, Springer, pp 48–70
17. Chapin N, Hale JE, Khan KM, Ramil JE, Tan WG (2001) Types of software evolution and software maintenance. J Softw Maint Evol Res Pract 13(1):3–30
18. Cheng BHC, De Lemos R, Giese H, Inverardi P, Magee J, Andersson J, Becker B, Bencomo N, Brun Y, Cukic B et al (2009) Software engineering for self-adaptive systems: a research roadmap. Softw Eng Self Adapt Syst, Springer, pp 1–26
19. Coen MH et al (1998) Design principles for intelligent environments. In: AAAI '98/IAAI '98 Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence, American Association for Artificial Intelligence, Menlo Park, pp. 547–554
20. Corcho O, Gómez-Pérez A (2000) A roadmap to ontology specification languages. Knowl Eng Knowl Manag Methods Models Tools 2000:80–96
21. Cossentino M, Dalle Nogare D, Giancarlo R, Lodato C, Lopes S, Ribino P, Sabatucci L, Seidita V (2014) GIMT: a tool for ontology and goal modeling in BDI multi-agent design. In: Workshop "Dagli Oggetti agli Agenti"
22. Cossentino M, Gaud N, Hilaire V, Galland S, Koukam A (2010) ASPECS: an agent-oriented software process for engineering complex systems. Auton Agents Multi-Agent Syst 20(2):260–304
23. Cossentino M, Lodato C, Lopes S, Sabatucci L (2015) Musa: a middleware for user-driven service adaptation. In: Proceedings of the 16th workshop "From objects to agents", Naples, 17–19 June 2015
24. Dalpiaz F, Borgida A, Horkoff J, Mylopoulos J (2013) Runtime goal models: keynote. In: Research challenges in information science (RCIS), 2013 IEEE seventh international conference, IEEE, pp 1–11
25. Dalpiaz F, Giorgini P, Mylopoulos J (2013) Adaptive socio-technical systems: a requirements-based approach. Requir Eng 18(1):1–24
26. De Lemos R, Giese H, Müller H, Shaw M, Andersson J, Litoiu M, Schmerl B, Tamura G, Villegas NM, Vogel T et al (2013) Software engineering for self-adaptive systems: a second research roadmap. Springer. Softw Eng Self Adapt Syst II:1–32
27. Dean TL, Kambhampati S (1997) Planning and scheduling. In: CRC handbook of computer science and engineering. CRC Press, Boca Raton, pp 614–636
28. Fikes RE, Nilsson NJ (1972) Strips: a new approach to the application of theorem proving to problem solving. Artif Intell 2(3):189–208
29. Ghezzi C, Pinto LS, Spoletini P, Tamburrelli G (2013) Managing non-functional uncertainty via model-driven adaptivity. In: Proceedings of the 2013 international conference on software engineering, IEEE Press, Piscataway, pp 33–42

30. Gomaa H, Hashimoto K (2012) Dynamic self-adaptation for distributed service-oriented transactions. In: Software engineering for adaptive and self-managing systems (SEAMS), 2012 ICSE Workshop, IEEE Press, Piscataway, pp 11–20
31. Gorlick M, Razouk RR (1991) Using weaves for software construction and analysis. In: Software engineering. Proceedings 13th international conference, IEEE, pp 23–34
32. Gu T, Wang XH, Pung HK, Zhang DQ (2004) An ontology-based context model in intelligent environments. In: Proceedings of communication networks and distributed systems modeling and simulation conference, San Diego, vol 2004, pp 270–275
33. Guarino N, Carrara M, Giaretta P (1994) Formalizing ontological commitment. AAAI 94:560–567
34. Guizzardi R, Franch X, Guizzardi G (2012) Applying a foundational ontology to analyze means-end links in the i framework. In: Research challenges in information science (RCIS), 2012 sixth international conference, IEEE, pp 1–11
35. Harker SDP, Eason KD, Dobson JE (1993) The change and evolution of requirements as a challenge to the practice of software engineering. In: Requirements engineering. Proceedings of IEEE international symposium, IEEE, pp 266–272
36. Jureta IJ, Borgida A, Ernst NA, Mylopoulos J (2014) The requirements problem for adaptive systems. ACM Trans Manag Inf Syst TMIS 5(3):17
37. Kephart JO, Chess DM (2003) The vision of autonomic computing. Computer 36(1):41–50
38. Kramer J, Magee J (2007) Self-managed systems: an architectural challenge. In: Future of software engineering, 2007. FOSE'07, IEEE, pp 259–268
39. Lamport L (1994) The temporal logic of actions. ACM Trans Progr Lang Syst TOPLAS 16(3):872–923
40. Lesperance Y (1989) A formal account of self-knowledge and action. IJCAI, Citeseer, pp 868–874
41. Lowe EJ (2002) A survey of metaphysics. Oxford University Press, Oxford
42. Moore RC (1979) Reasoning about knowledge and action. PhD thesis, Massachusetts Institute of Technology
43. Morandini M, Penserini L, Perini A (2008) Towards goal-oriented development of self-adaptive systems. In: Proceedings of the 2008 international workshop on software engineering for adaptive and self-managing systems, ACM, New York, pp 9–16
44. Morandini M, Penserini L, Perini A (2009) Operational semantics of goal models in adaptive agents. In: Proceedings of the 8th international conference on autonomous agents and multiagent systems, International Foundation for Autonomous Agents and Multiagent Systems, vol 1, ACM, Richland, pp 129–136
45. Newell A (1982) The knowledge level. Artif Intell 18(1):87–127
46. O'Brien PD, Nicol RC (1998) Fipa—towards a standard for software agents. BT Technol J 16(3):51–59
47. Oreizy P, Gorlick MM, Taylor RN, Heimbigner D, Johnson G, Medvidovic N, Quilici A, Rosenblum DS, Wolf AL (1999) An architecture-based approach to self-adaptive software. IEEE Intell Syst 3:54–62
48. Oreizy P, Medvidovic N, Taylor RN (1998) Architecture-based runtime software evolution. In: Proceedings of the 20th international conference on software engineering, IEEE Computer Society, Washington, pp 177–186
49. Patikirikorala T, Colman A, Han J, Wang L (2012) A systematic survey on the design of self-adaptive software systems using control engineering approaches. In: Software engineering for adaptive and self-managing systems (SEAMS), 2012 ICSE workshop, pp 33–42
50. Pistore M, Marconi A, Bertoli P, Traverso P (2005) Automated composition of web services by planning at the knowledge level. IJCAI 19: 1252–1259
51. Rao AS (1996) Agentspeak (l): BDI agents speak out in a logical computable language. In: Agents breaking away, Springer, pp 42–55
52. Ribino P, Cossentino M, Lodato C, Lopes S, Sabatucci L, Seidita V (2013) Ontology and goal model in designing BDI multi-agent systems. WOA@ AI* IA 1099:66–72
53. Rouvoy R, Barone P, Ding Y, Eliassen F, Hallsteinsen S, Lorenzo J, Mamelli A, Scholz U (2009) Music: middleware support for self-adaptation in ubiquitous and service-oriented environments. Softw Eng Self Adapt Syst, Springer, pp 164–182
54. Sabatucci L, Cossentino M (2015) From means-end analysis to proactive means-end reasoning. In: Proceedings of 10th international symposium on software engineering for adaptive and self-managing systems, Florence, 18–19 May 2015
55. Sabatucci L, Cossentino M, Lodato C, Lopes S, Seidita V (2013) A possible approach for implementing self-awareness in Jason. EUMAS, Citeseer, pp 68–81
56. Sabatucci L, Lodato C, Lopes S, Cossentino M (2013) Towards self-adaptation and evolution in business process. AIBP@ AI* IA, Citeseer, pp 1–10
57. Sabatucci L, Lodato C, Lopes S, Cossentino M (2015) Highly customizable service composition and orchestration. In: Dustdar S, Leymann F, Villari M (eds) Service oriented and cloud computing. Lecture notes in computer science, vol 9306. Springer, Berlin, pp 156–170
58. Sabatucci L, Ribino P, Lodato C, Lopes S, Cossentino M (2013) Goalspec: a goal specification language supporting adaptivity and evolution. Eng Multi-Agent Syst, Springer, pp 235–254
59. Saeki M (2010) Semantic requirements engineering. Intent Perspect Inf Syst Eng, Springer, pp 67–82
60. Souza VES, Lapouchnian A, Mylopoulos J (2012) (Requirement) evolution requirements for adaptive systems. In: Software engineering for adaptive and self-managing systems (SEAMS), 2012 ICSE workshop, pp 155–164
61. Sycara K, Widoff S, Klusch M, Jianguo L (2002) Larks: dynamic matchmaking among heterogeneous software agents in cyberspace. Auton Agents Multi-Agent Syst 5(2):173–203
62. Whittle J, Sawyer P, Bencomo N, Cheng BH, Bruel JM (2009) Relax: incorporating uncertainty into the specification of self-adaptive systems. In: Requirements engineering conference. RE'09. 17th IEEE international, IEEE, pp 79–88
63. Wooldridge MJ (2000) Reasoning about rational agents. MIT Press, Cambridge
64. Yu E (2011) Modelling strategic relationships for process reengineering. Soc Model Requir Eng 11:2011
65. Zambonelli F, Castelli G, Mamei M, Rosi A (2014) Programming self-organizing pervasive applications with SAPERE. Intell Distrib Comput VII, Springer, pp 93–102