Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

SELF-ADAPTATION BY PROACTIVE MEANS-END REASONING

L. Sabatucci, M. Cossentino

Rapporto Tecnico N.: ***Data:***

RT-ICAR-PA-02-06 **Febbraio 2016**

Self-Adaptation by Proactive Means-End Reasoning

L. Sabatucci and M. Cossentino
ICAR-CNR, Palermo, Italy
{sabatucci, cossentino}@pa.icar.cnr.it

April 27, 2016

Abstract

Run-time goal-model artifacts represent a notable approach to communicate requirements to the system and open new directions for dealing with self-adaptation.

This work presents a theoretical framework and a general architecture for system evolution, self-configuration and self-healing. The novelty is that of breaking design-time constraints between system goals and tasks. The user may inject, at run-time, goal-models that do not contain tasks, i.e. the description of how to address them. Therefore, the architecture is responsible to configure its components as the result of deductions made at the knowledge level. The strength of this architecture is to promote reusability and domain independence.

Finally, the proposed implementation of the architecture has been evaluated in the context of self-configuration and self-healing through the execution of a set of randomized stress tests.

1 Introduction

Modern distributed and open software systems raise the need to integrate several heterogeneous components and environments into corporate-wide computing systems, and to extend their working boundaries beyond companies into the Internet [34]. 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 [34], released in 2001, suggests a promising direction for facing software complexity through self-adaptation. Direction

that is detailed through many research roadmaps [17, 24] of software engineering for self-adaptive system that define *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, in order to address their goals* [24].

The vision of computing systems that can manage themselves is fascinating. They are able of changing their behavior at run-time in order either to maintain or enhance their functions [17]. Self-adaptation has deep roots in several research fields, as for instance, artificial intelligence, biological inspired computing, robotics, requirements/knowledge engineering, control theory, fault-tolerant computing, and so on. In the last decade, the large and heterogeneous number of works concerning self-adaptation investigated several aspects of the problem, for instance specific architectures for implementing adaptive control loops [46], self-organizing paradigms [4], adaptive requirements [23] an so on. However, to date, many of these problems still remain significant intellectual challenges [17, 24].

Among the others, one point is becoming clear: as long as self-adaptive systems become reality, human users (not only managers) will inevitably participate to the process of adaptation [7]. This point is central for the models@runtime community [9] 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 [40] and elements of uncertainty in the environment [60]. These elements must be first class entities the system can exploit to decide how to act. Currently goal-oriented methodologies [14, 23] 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's execution. To this aim we defined a human-oriented language for specifying system goals [55]. 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 [51].

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 [42], is independent from any specific application context, but it rather can be reused in many domains. A specific focus is given to atomic and self-contained portion of behavior, called *capabilities*, which implement the paradigm of full-reuse [6]. Indeed their peculiarity is of being automatically composable, on demand, in order to build system functionalities and to address dynamic and evolving goals. The proposed architecture integrates the MAPE-K model [46, 15] in order 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 [11] a declarative programming language based on BDI theory [13]. 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: Section 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, discuss strengths and limits of the approach. Section 6 briefly summarizes the proposed architecture. Other details of the prototype are in Appendix.

2 Background and Definition

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

2.1 State of the World and Goals

We consider software system has (partial) knowledge about the environment in which it runs. The classic way for expressing this property is (Bel a φ) [61] that specifies that a software agent a believes φ is true, where φ is a generic state of affair. We decided to limit the range of φ to first order variable-free statements (facts). They are enough expressive for representing an object of the environment, a particular property of an object or a relationship between

two ore more objects. A fact is a statement to which it is possible to assign a truth-value. Examples are: $tall(john)$ 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) $s_1, s_2 \dots s_n$ that can be used in a given domain.*

W^τ has the following characteristics:

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

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 it 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 $\forall s_i, s_j \in S$

$$\text{if } \{s_i, s_j\} \models \bot \text{ then } \begin{cases} s_i \in W^\tau \Rightarrow s_j \notin W^\tau \\ s_j \in W^\tau \Rightarrow s_i \notin W^\tau \end{cases} \tag{2}$$

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 logic formula composed by predicates or variables, through the standard set of logic 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* [14] is: "a goal is a state of affair that an actor wants to achieve". We refined this statement to be compatible with the definition of W^t as: "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*

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

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.

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 [22] but we explicitly removed Influence [22] relations and Means-End [14] 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 the partial goal model, represented with the i^* notation, for the meeting scheduling case study. This example, redesigned from [22], includes functional (hard) goals only, and AND/OR refinements. The root goal is to provide meeting scheduling services: it is decomposed into schedule meetings, send reminders, cancel meetings and running a website. Therefore meetings are scheduled by collecting participant timetables, choosing a schedule and a location. Such a model is useful for analysts to explore alternative ways to fulfill the root goal.

2.2 Proactive Means-End Reasoning

In many goal-oriented approaches, a Task is defined as the operationalization of a Goal. This means that each task, in a goal model, is associated to one (or more) leaf goal(s). This association is made at design time as the result of a human activity called means-end analysis. In the i^* conceptual model [62], a means-end link introduces *a means to attain an end* where *the end can be a goal, task, resource or softgoal, whereas the means is usually a task.* The TROPOS methodology [14] introduces means-end analysis as the activity for identifying (possibly several alternative) tasks to satisfy a goal.

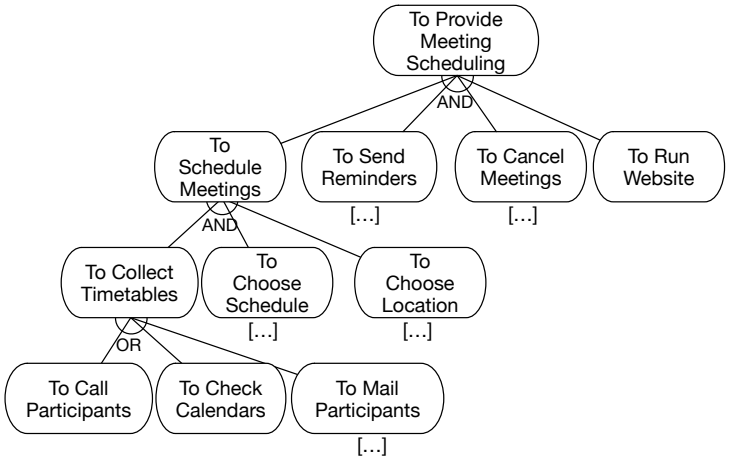

Figure 1: Portion of Goal Model for the Meeting Scheduling case study. The tree has been truncated (with respect to the original one) where the symbol [...] appears because of space concerns.

The task is therefore an analysis entity that encapsulates how to address a given goal according to the following statement: "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" [31].

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.

Capabilities and Goals. Whereas a task has an explicit link to a goal, a capability is relatively independent from 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. In order 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 [8] whose details are out of the scope of this paper.

Composition of Capabilities. In order 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 this 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 deduct at run-time by checking the satisfiability of pre and post conditions [8] . When capabilities are composable then System of equations 4 changes for including the resulting evolution function as the sum of each single capability's evolution.

Parametric Capabilities: a task is generally 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 therefore 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 (or 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

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

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

It is worth noting that:

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 shortcoming 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 [25];

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 [44, 40, 22].

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 this data is represented in abstract form.

Reasoning at the knowledge level [42], it is possible to represent complex abstract data that is 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).

For instance, even if we do not know all the users of the meeting scheduler, we are able of implementing a capability that checks whether a desired participant is available or not for the meeting through the predicate *available(User,Meeting)*. At the same way we may specify a generic goal concerning the selection of a location for the meeting through the predicate *assigned(Meeting,Location)*.

In order to make the algorithm affordable we obtain the knowledge level automatically from specifications of goals and capabilities. Therefore, evaluating the contextual fulfillment 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-steps 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 sub-spaces according the structure of the goal model and available capabilities. Then it uses a STRIPS-based [26] 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

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

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

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(C, GM, g_target);
        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
```

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

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. At 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 (i) the distance from the final state and (ii) the quality of the partial paths and therefore it estimates the global impact in satisfying the whole goal-model.*

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 Equation 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 Figure 2). In the case CS satisfies the goal then the capability sequence represents a solution and it is added to the *h_set*.

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

Input: GM is the goal-model to address, g_{target} is the goal for which finding 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 $compose\_capabilities(GM, g_{target}, W_I, C)$ **begin**
  $WTS \leftarrow initialize\_space(W_I)$;
  while $|h\_set| < max\_h\_set \ AND \ |WTS| < max\_space$ **do**
    $W_i \leftarrow get\_highest\_scored\_state(WTS)$;
    $CS \leftarrow path\_from\_to(WTS, W_I, W_i)$;
    if $check\_cs\_is\_solution(CS, g_{target})$ **then**
      $add\_solution(h\_set, CS)$;
      $mark\_as\_solution(WTS, CS)$;
    else
      $cap\_set \leftarrow get\_next\_capabilities(W_i, CS, WTS)$;
      $expand\_and\_score(WTS, W_i, cap\_set)$;
    end
  end
  return h_set
end

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-condition is true in W_i:

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

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

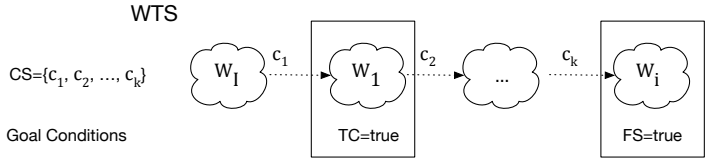

Figure 2: Illustration of the procedure for evaluating the satisfaction of a goal along a state of the world evolution. Firstly the algorithm searches for a state of the world in which $TC = true$. After that, it proceeds searching for a state of the world in which $FS = true$.

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 this observation, we state two principles for comparing states of the world obtained by capability composition:

- 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 Equation 5;
- the *principle of precision* i.e. the more a state of the world contains statements that does not provide a positive impact to a goal, the more it is *minimal* for addressing it, according Equation 6;.

As a consequence we can specify the function as follows:

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

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*). This 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.

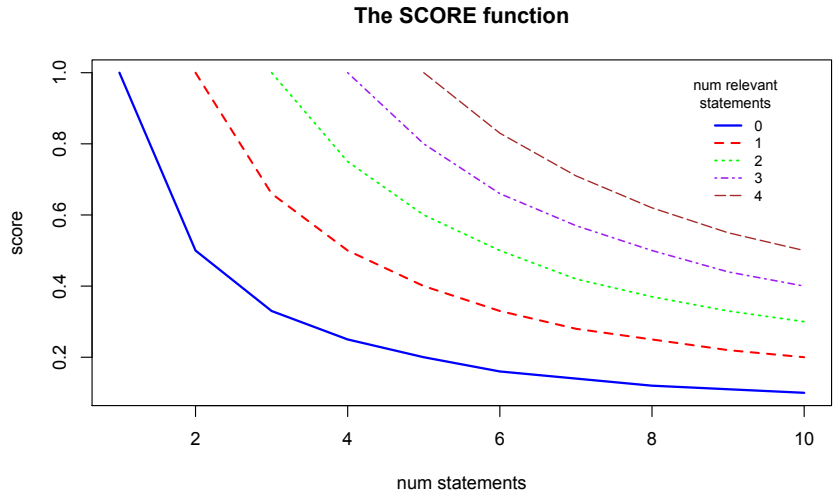

Figure 3: Line chart of the score function highlights trends of the value when making either $num\_statements(W)$ or $num\_relevant\_statements(W, g)$ constant.

The 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 relation between the approach presented in this paper and three fundamental characteristics for self-adaptive system: system evolution, self-configuration and self-healing [34, 17].

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 [16] an aspect of software evolution that refers to modification performed to keep software usable in a dynamic environment. Real world changes continuously and therefore user needs evolve over time. Software that runs in an environment is likely to evolve continuously to adapt to varying requirements and circumstances in that environment. This is translated into functional enhancement and/or into the improvement of performances in order to reflect requirements evolution.

A prominent characteristic of the proposed architecture is to handle run-time addition of new requirements and therefore to amount to system evolution [55, 53]. 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 Injection*** [53]. 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 responsibility of the system via the PMR Ability to adapt itself to the new needs. The goal injection is enabled by two components:

- on 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 environment properties. The system acquires goals from the user and maintains knowledge of them thus to be able of reasoning on expected results and finally conditioning its global behavior. Of course, existing goals may be retreated as well.

Goal injection enables *user-requirements to evolve over time* [32] without either user-management or restarting the system. This could be fundamental for some categories of domain in which continuity of service is central (financial, service providing and so on).

In addition it is possible *to increase or 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 capability evolution and may be used for different problem domains without any other specific 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 [34, 12, 45].

This subsection shows a three-layer architecture that exploits the PMR Ability for generating business logic for requirements 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. in order to solve the same problem the system may exploit different combinations of capabilities. Some of these capabilities have input/output parameters that are to be configured in order to concretely use them.

The proposed architecture is made of three layers (Figure 4): the goal layer, the capability layer and the business layer.

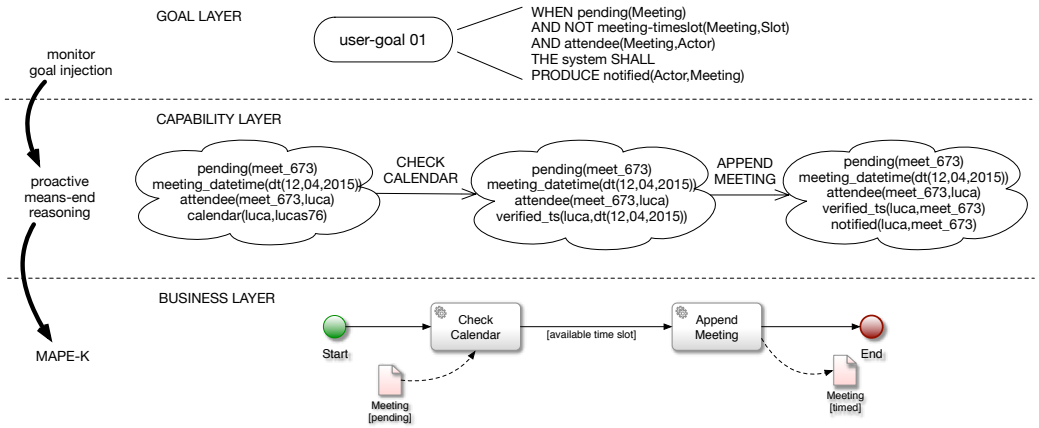

Figure 4: Overview of the three layers architecture with for Self-Configuration.

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 no 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 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. This corresponds to a strategic deliberation phase in which decisions are made according to the (incomplete) system 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 ***Business Layer*** that executes the business process generated at the second layer. This layer consists of atomic blocks of computation for acquiring and analyzing real data from the environment and to act for producing the desired state of the world. This layer may easily be implemented by the MAPE-K model [46, 15], well known in literature. It requires i) a Monitoring component that acquires information from the environment, and it updates the system knowledge accordingly; ii) an Analyze component that uses the knowledge to determine the need for adaptation with respect to expected states of the world or capabilities failure; iii) a Plan component that uses the acquired knowledge to synchronize the available capabilities according the goal hierarchy and, finally, iv) 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 [34, 33].

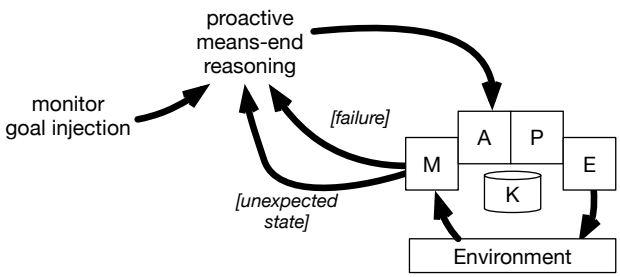

Figure 5: Graphical representation of the Self-Healing Loop.

In the previous section we have adopted the MAPE-K model [46, 15] for implementing the business layer of the presented architecture. According the roadmap of self-adaptive systems [17], one of the principles for implementing self-healing is to explicitly focus on the 'control loop' 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 in order to generate a macro-loop for self-healing, as shown in Figure 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 goal by activating the *PMR Ability* in order 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 environment conditions.

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 Algorithm 1 and Algorithm 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 Section 4 has been implemented in MUSA, a Middleware for User-driven Service Adaptation [21]. MUSA is built as a multi agent system and developed in JASON [11], a declarative programming language based on the AgentSpeak language [49] and the BDI theory [13]. 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 [52]. This enabled the development of the agent PMR Ability for reasoning on Goals and Capabilities as first class entities [51, 21]. Additional details on MUSA are provided in 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 the couple of algorithms for analyzing the goal model and exploring the space of solutions for composing capabilities. This 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.

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: $goal(\text{"}g38\text{"}, condition(not(z(u))), condition(z(a)))$ 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: $cap(\text{"}c1\text{"}, evo([remove(r(u)), add(z(u))]), condition(r(u)), condition(z(u)))$.

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

Therefore we run series of tests with an incremental growth of number of capabilities, starting from 20, till 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 Figure 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: 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

As it can be seen, the number of visited nodes (and therefore the time-to-complete) is polynomial with respect to the number of capabilities both for the score-driven search and for the exhaustive search (see 'visited states

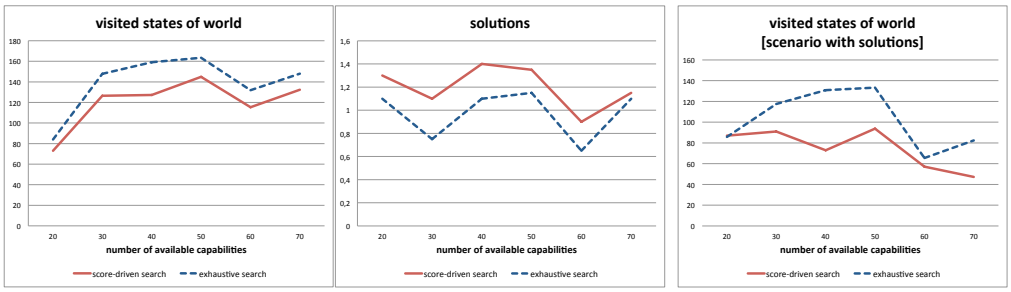

Figure 6: Data obtained by comparing the algorithms presented in Section 3 with a breath-first strategy. Configurations are the result of 120 executions with random input and increasing of 10 the number of capabilities every 20 runs.

of world' in Figure 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 (Equations 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 both from the point of view of the *number of visited nodes* and for what concerns the *number of discovered solutions*.

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

5.2 Evaluating Self-Healing

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

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 generating 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 run 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.

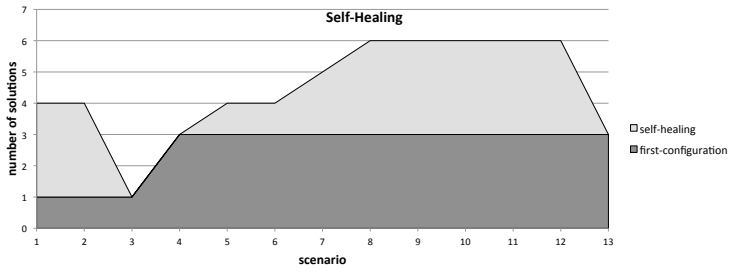

Figure 7: Result of the sequence of tests for self-healing. The dark grey area represents the size of the space of solutions discovered at the first run of self-configuration for each scenario. The light grey area represents the additional space of solutions built as a result of self-healing.

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, only in three cases (scenarios 3,4 and 13) the adaptation failed because the available capabilities were not enough to repair the failure. In all the other cases the procedure performed well, increasing the space of configuration just enough to allow the correct 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

To date we identified a semantic gap exists between requirement specifications defined at design-time [58, 48] and the concept of goal used at run-time [9]. This represents a limitation especially in the development of self-adaptive and evolving systems.

Morandini et al. [41, 40] propose to extend the operational semantics of goal models by characterizing the behavior of run-time goals thus to be directly implemented. The solution is that of enriching the definition of goal by specifying their dynamics and maintaining the flexibility of using different goal types and conditions. Dalpiaz et al. [22] propose a new type of goal model, called runtime goal model (RGM) which extends the former with annotation about additional state, behavioral and historical information

about the fulfillment of goals, for instance explaining when and how many instances of the 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.

SAPERE [63] (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.

Baresi et al. [5] 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 [4], 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 for 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. In addition they introduce fuzzy goals for expressing the satisfaction degree of requirements that is a possible future direction for extending our definition of goal.

Gorlick et al. [29] present an approach to manage 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. [10] 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 are able of discovering when one agent's workflow changed unilaterally because it may incur conversation errors with other agents. An agent also recognizes mismatches between its own 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* situa-

tions could be an interesting direction for our work.

Kramer and Magee [35] propose a three-layer architecture for self-adaptation inspired from robotics. The architecture includes (i) a control layer, a reactive component consisting of sensors, actuators and control loops, (ii) a sequencing layer which reacts to changes from the lower levels by modifying plans to handle the new situation and (iii) 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 [28], 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. As well as our approach, SASSY provides a uniform approach to automated adaptation software systems, however to date, goal evolution is out of the scope of their work.

Souza et al. [57] 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. [27] 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 a MDP (Markov Decision Process). It is possible to calculate the possible values for the different executions and to thus navigate the model to execute it.

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 [6]. Capabilities are atomic, self-contained and created for being composable. 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 [47]. 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 [26], actions are instantaneous and there is no provision for asserting what is true while an action is in execution. 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, 36].

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 5 research projects with heterogeneous ap-

plication contexts, from dynamic workflow [53] to a smart-travel system [54]. However, in our *in-vitro* evaluation, the same development team created both Capabilities and Goals 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 properly work.

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 over 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 in terms of states of the world, goals and capabilities. Solving the problem at the knowledge level provided us the opportunity to define a general architecture for system evolution, self-configuration and self-healing. This architecture is based on the idea that a user, at runtime, may inject a new goal model without specifying the description of how to address it. The proposed architecture is responsible for configuring and reconfiguring its business layer as the result of reasoning and deductions made at the knowledge level. System evolution is the result of a process of goals management, self-configuration is obtained through the ability of solving the proactive means-end reasoning and finally self-healing is obtained by closing the loop between self-configuration and execution. The strengths of the proposed architecture is to be domain independent and to support reusability across many application contexts.

References

[1] Dhaminda B Abeywickrama, Nicola Bicocchi, and Franco Zambonelli. Sota: Towards a general model for self-adaptive systems. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2012 IEEE 21st International Workshop on*, pages 48–53. IEEE, 2012.

[2] James F Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[3] James F Allen. Planning as temporal reasoning. *KR*, 91:3–14, 1991.

[4] Luciano Baresi and Sam Guinea. A3: self-adaptation capabilities through groups and coordination. In *Proceedings of the 4th India Software Engineering Conference*, pages 11–20. ACM, 2011.

[5] Luciano Baresi, Liliana Pasquale, and Paola Spoletini. Fuzzy goals for requirements-driven adaptation. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 125–134. IEEE, 2010.

[6] Victor R Basili. Viewing maintenance as reuse-oriented software development. *Software, IEEE*, 7(1):19–25, 1990.

[7] Amel Bennaceur, Robert France, Giordano Tamburrelli, Thomas Vogel, Pieter J Mosterman, Walter Cazzola, Fabio M Costa, Alfonso Pierantonio, Matthias Tichy, Mehmet Akşit, et al. Mechanisms for leveraging models at runtime in self-adaptive software. In *Models@ run. time*, pages 19–46. Springer, 2014.

[8] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS Press, 2009.

[9] Gordon Blair, Nelly Bencomo, and Robert B France. Models@ run. time. *Computer*, 42(10):22–27, 2009.

[10] Warren Blanchet, Eleni Stroulia, and Renée Elio. Supporting adaptive web-service orchestration with an agent conversation framework. In *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*. IEEE, 2005.

[11] R.H. Bordini, J.F. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. Wiley-Interscience, 2007.

[12] Victor Braberman, Nicolas D'Ippolito, Jeff Kramer, Daniel Sykes, and Sebastian Uchitel. Morph: A reference architecture for configuration and behaviour self-adaptation. *arXiv preprint arXiv:1504.08339*, 2015.

[13] Michael E Bratman, David J Israel, and Martha E Pollack. Plans and resource-bounded practical reasoning. *Computational intelligence*, 4(3):349–355, 1988.

[14] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[15] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. In *Software engineering for self-adaptive systems*, pages 48–70. Springer, 2009.

[16] Ned Chapin, Joanne E Hale, Khaled Md Khan, Juan F Ramil, and Wui-Gee Tan. Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, 13(1):3–30, 2001.

[17] Betty HC Cheng, Rogerio De Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, et al. Software engineering for self-adaptive systems: A research roadmap. In *Software engineering for self-adaptive systems*, pages 1–26. Springer, 2009.

[18] O. Corcho and A. Gómez-Pérez. A roadmap to ontology specification languages. *Knowledge Engineering and Knowledge Management Methods, Models, and Tools*, pages 80–96, 2000.

[19] Massimo Cossentino, Daniele Dalle Nogare, Raffaele Giancarlo, Carmelo Lodato, Salvatore Lopes, Patrizia Ribino, Luca Sabatucci, and Valeria Seidita. Gimt: A tool for ontology and goal modeling in bdi multi-agent design. In *Workshop" Dagli Oggetti agli Agenti"*, 2014.

[20] Massimo Cossentino, Nicolas Gaud, Vincent Hilaire, Stéphane Galland, and Abderrafiâa Koukam. Aspecs: an agent-oriented software process for engineering complex systems. *Autonomous Agents and Multi-Agent Systems*, 20(2):260–304, 2010.

[21] Massimo Cossentino, Carmelo Lodato, Salvatore Lopes, and Luca Sabatucci. Musa: a middleware for user-driven service adaptation. In *Proceedings of the 16th Workshop "From Objects to Agents"*, Naples, Italy, June 17-19, 2015.

[22] Fabiano Dalpiaz, Alexander Borgida, Jennifer Horkoff, and John Mylopoulos. Runtime goal models: Keynote. In *Research Challenges in Information Science (RCIS), 2013 IEEE Seventh International Conference on*, pages 1–11. IEEE, 2013.

[23] Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos. Adaptive sociotechnical systems: a requirements-based approach. *Requirements engineering*, 18(1):1–24, 2013.

[24] Rogério De Lemos, Holger Giese, Hausi A Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[25] Thomas L Dean and Subbarao Kambhampati. Planning and scheduling. In *CRC Handbook of Computer Science and Engineering*, pages 614–636. CRC Press, 1997.

[26] Richard E Fikes and Nils J Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3):189–208, 1972.

[27] Carlo Ghezzi, Leandro Sales Pinto, Paola Spoletini, and Giordano Tamburrelli. Managing non-functional uncertainty via model-driven adaptivity. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 33–42. IEEE Press, 2013.

[28] H. Gomaa and K. Hashimoto. Dynamic self-adaptation for distributed service-oriented transactions. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, pages 11–20, 2012.

[29] Michael M Gorlick and Rami R Razouk. Using weaves for software construction and analysis. In *Software Engineering, 1991. Proceedings., 13th International Conference on*, pages 23–34. IEEE, 1991.

[30] Nicola Guarino, Massimiliano Carrara, and Pierdaniele Giaretta. Formalizing ontological commitment. In *AAAI*, volume 94, pages 560–567, 1994.

[31] Renata Guizzardi, Xavier Franch, and Giancarlo Guizzardi. Applying a foundational ontology to analyze means-end links in the i framework. In *Research Challenges in Information Science (RCIS), 2012 Sixth International Conference on*, pages 1–11. IEEE, 2012.

[32] Susan DP Harker, Ken D Eason, and John E Dobson. The change and evolution of requirements as a challenge to the practice of software engineering. In *Requirements Engineering, 1993., Proceedings of IEEE International Symposium on*, pages 266–272. IEEE, 1993.

[33] Ivan J Jureta, Alexander Borgida, Neil A Ernst, and John Mylopoulos. The requirements problem for adaptive systems. *ACM Transactions on Management Information Systems (TMIS)*, 5(3):17, 2014.

[34] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[35] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering, 2007. FOSE'07*, pages 259–268. IEEE, 2007.

[36] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[37] Yves Lesperance. A formal account of self-knowledge and action. In *IJCAI*, pages 868–874. Citeseer, 1989.

[38] E.J. Lowe. *A survey of metaphysics*. Oxford University Press Oxford, 2002.

[39] Robert C Moore. *Reasoning about knowledge and action*. PhD thesis, Massachusetts Institute of Technology, 1979.

[40] Mirko Morandini, Loris Penserini, and Anna Perini. Towards goal-oriented development of self-adaptive systems. In *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 9–16. ACM, 2008.

[41] Mirko Morandini, Loris Penserini, and Anna Perini. Operational semantics of goal models in adaptive agents. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 129–136. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[42] Allen Newell. The knowledge level. *Artificial intelligence*, 18(1):87–127, 1982.

[43] Patrick D O'Brien and Richard C Nicol. Fipa—towards a standard for software agents. *BT Technology Journal*, 16(3):51–59, 1998.

[44] Peyman Oreizy, Michael M Gorlick, Richard N Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S Rosenblum, and Alexander L Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent systems*, (3):54–62, 1999.

[45] Peyman Oreizy, Nenad Medvidovic, and Richard N Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th international conference on Software engineering*, pages 177–186. IEEE Computer Society, 1998.

[46] T. Patikirikorala, A. Colman, J. Han, and Liuping Wang. 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 on*, pages 33–42, 2012.

[47] Marco Pistore, Annapaola Marconi, Piergiorgio Bertoli, and Paolo Traverso. Automated composition of web services by planning at the knowledge level. In *IJCAI*, pages 1252–1259, 2005.

[48] Nauman A Qureshi and Anna Perini. Engineering adaptive requirements. In *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS'09. ICSE Workshop on*, pages 126–131. IEEE, 2009.

[49] Anand S Rao. Agentspeak (l): Bdi agents speak out in a logical computable language. In *Agents Breaking Away*, pages 42–55. Springer, 1996.

[50] Patrizia Ribino, Massimo Cossentino, Carmelo Lodato, Salvatore Lopes, Luca Sabatucci, and Valeria Seidita. Ontology and goal model in designing bdi multi-agent systems. *WOA@ AI* IA*, 1099:66–72, 2013.

[51] Luca Sabatucci and Massimo Cossentino. 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, Italy*, May 18-19 2015.

[52] Luca Sabatucci, Massimo Cossentino, Carmelo Lodato, Salvatore Lopes, and Valeria Seidita. A possible approach for implementing self-awareness in jason. In *EUMAS*, pages 68–81. Citeseer, 2013.

[53] Luca Sabatucci, Carmelo Lodato, Salvatore Lopes, and Massimo Cossentino. Towards self-adaptation and evolution in business process. In *AIBP@ AI* IA*, pages 1–10. Citeseer, 2013.

[54] Luca Sabatucci, Carmelo Lodato, Salvatore Lopes, and Massimo Cossentino. Highly customizable service composition and orchestration. In Schahram Dustdar, Frank Leymann, and Massimo Villari, editors, *Service Oriented and Cloud Computing*, volume 9306 of *Lecture Notes*

in Computer Science, pages 156–170. Springer International Publishing, 2015.

[55] Luca Sabatucci, Patrizia Ribino, Carmelo Lodato, Salvatore Lopes, and Massimo Cossentino. Goalspec: A goal specification language supporting adaptivity and evolution. In *Engineering Multi-Agent Systems*, pages 235–254. Springer, 2013.

[56] Motoshi Saeki. Semantic requirements engineering. In *Intentional Perspectives on Information Systems Engineering*, pages 67–82. Springer, 2010.

[57] V.E.S. Souza, A. Lapouchnian, and J. Mylopoulos. (requirement) evolution requirements for adaptive systems. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, pages 155–164, 2012.

[58] Vítor E Silva Souza, Alexei Lapouchnian, Konstantinos Angelopoulos, and John Mylopoulos. Requirements-driven software evolution. *Computer Science-Research and Development*, 28(4):311–329, 2013.

[59] Katia Sycara, Seth Widoff, Matthias Klusch, and Jianguo Lu. Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. *Autonomous agents and multi-agent systems*, 5(2):173–203, 2002.

[60] Jon Whittle, Peter Sawyer, Nelly Bencomo, Betty HC Cheng, and Jean-Michel Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, pages 79–88. IEEE, 2009.

[61] Michael J Wooldridge. *Reasoning about rational agents*. MIT press, 2000.

[62] Eric Yu. Modelling strategic relationships for process reengineering. *Social Modeling for Requirements Engineering*, 11:2011, 2011.

[63] Franco Zambonelli, Gabriella Castelli, Marco Mamei, and Alberto Rosi. Programming self-organizing pervasive applications with sapere. In *Intelligent Distributed Computing VII*, pages 93–102. Springer, 2014.