# Adaptive Goals for Self-Adaptive Service Compositions

Luciano Baresi and Liliana Pasquale
*Politecnico di Milano*
*DEI - Dipartimento di Elettronica e Informazione*
*p.zza Leonardo Da Vinci, 32 - 20133, Milano, Italy*
{*baresi, pasquale*}*@elet.polimi.it*

***Abstract*—Service compositions need to continuously self-adapt to cope with unexpected failures. In this context adaptation becomes a fundamental requirement that must be elicited along with the other functional and non functional requirements. Beside modelling, effective adaptation also demands means to trigger it at runtime as soon as the actual behavior of the composition deviates from stated requirements. This paper extends traditional goal models with *adaptive* goals to support continuous adaptation. Goals become live, runtime entities whose satisfaction level is dynamically updated. Furthermore, boundary infringement triggers adaptation capabilities. The paper also provides a methodology to trace goals onto the underlying composition, assess goals satisfaction at runtime, and activate adaptation consequently. All the key elements are demonstrated on the definition of the process to control an advanced washing machine.**

## I. Introduction

Service compositions (or processes) are one of the main technical solution adopted to provide business processes. The reasons behind this success lay in their flexibility and the reduced development costs. However the main pitfall of these applications is their intrinsic unreliability due to the distributed and loosely coupled nature of their service components. For this reason a service composition may fail unexpectedly due to several faults [5]: physical faults (if the network malfunctions, or a partner service is down), development faults (e.g., incompatibility among parameters or changes in the interfaces provided by partner services), or interaction faults (QoS violations, slow response times). In this context self-adaptation is a critical feature for service-centric applications that need to cope with failures autonomously to keep their behavior aligned with the "perception" of the stakeholders. The need for adaptation may also come from new/changing business needs, that impose to update the functionality provided by the system accordingly.

A simple way to achieve this objective would be the design-time definition of all possible evolutions of the process, but this is not always feasible, since some requirements or features of the operational environment may be unknown, and it would also produce an overly tangled design of the process. In contrast, it is much more convenient if we think of adaptation as a requirement per-se that must be properly supported throughout the whole lifecycle of the process.

Requirements models must represent the "conventional" (i.e. functional and non functional) requirements of a service composition, but also its adaptation capabilities, along with the diagnosis mechanisms adopted to trigger them. Requirements must also be self-reflective, providing suitable mechanisms to assess their satisfaction at runtime and enable adaptation strategies to keep the execution on track.

Despite traditional goal models, like KAOS [3] or i* [17], have been widely adopted to represent requirements, they do not consider adaptation explicitly. Even if different approaches that try to model adaptation in goals specification also exist (e.g., [8] [11]), they only help list the different strategies that can be performed, and do not offer explicit support to the actual evolution of the system. Furthermore, to the best of our knowledge, none of these goal models directly embed the mechanisms to measure the satisfaction level of stated goals and support their modifications.

In this paper, we propose an innovative approach to represent requirements for service compositions and enforce them from the design down to the execution. Adaptation is selected according to requirements' satisfaction/violation levels. According to our view, there are: *crisp* requirements, whose satisfaction can be expressed in terms of a binary value (yes or not), and *fuzzy*, which can be satisfied at different degrees (expressed through a real number $x \in [0, 1]$).

As shown in Figure 1 our solution works at three different levels of abstraction:

1) We adopt a *live* goal model to render the process' requirements along with its adaptation capabilities. This model is able to accomodate changes at runtime and update goals' satisfaction levels dynamically. We also introduce *adaptive* goals to dynamically trigger adaptation actions, cause the system to move to a new goal model, and propagate the corresponding changes to the underlying implementation.
2) We support requirements traceability by linking the goal model to both a functional and a supervision model of the system. The former includes a set of abstract processes able to achieve the objectives stated in the goal model. The latter defines how to measure and update goals' satisfaction levels, along with the policies to safely apply adaptation at runtime.
3) We execute the functional and supervision models

through a suitable infrastructure [4]. It includes a set of execution engines that run the processes included in the functional model. It also provides data collectors, monitors, and adaptors to apply the monitoring and adaptation activities defined in the supervision model.

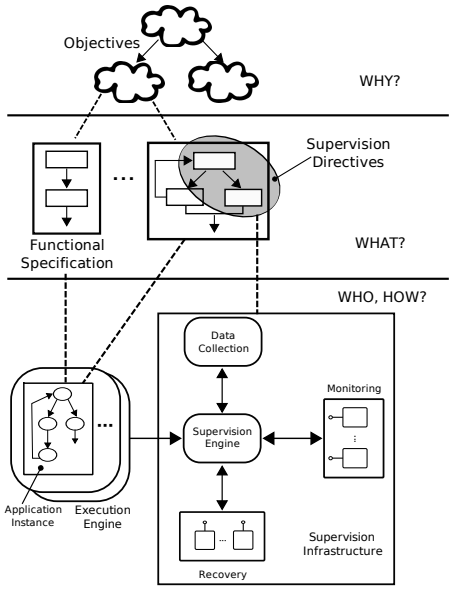



Figure 1. Overall approach.

The approach is described through a simple laundry system that controls a washing machine. It must provide assignments, select washing programs, and activate washing cycles. The expected result is to accomplish all tasks with minimum energy consumption.

The rest of the paper is structured as follows. Section II presents our goal model. Section III describes the translation of the goal model into the functional and supervision models, and explains how they can be applied at runtime, Section IV surveys related works, and Section V concludes the paper.

## II. Goal Model

This section describes the upper part of the proposal. We adapt KAOS [14] introducing fuzzy logic operators, already proposed in RELAX [16], to formalize the goals associated with fuzzy requirements (fuzzy goals), while goals representing crisp requirements (crisp goals) are still represented with traditional logic operators. We also leverage goals "operationalization" which allows us to reason about the functional view of the system, easing the generation of a direct mapping onto the underlying implementation. Finally we also embed in KAOS the possibility to express *adaptive goals* carrying on the diagnosis that may trigger some adaptation actions and the modifications introduced in the goal model by these adaptations.

### A. KAOS goal model

The main features provided by KAOS are goals refinement and formalization. Goals refinement allows us to decompose a goal into several conjoined subgoals (AND-refinement) or into alternative combinations of subgoals (OR-refinement). The satisfaction of the parent goal depends on the achievement of all (for AND-refinement) or at least one (for OR-refinement) of its underlying subgoals. Goals decomposition can also be accomplished through formal rules [14]. The refinement of a goal terminates when it can be "operationalized", i.e., it can be decomposed into a set of operations.

Goals are formalized in terms of LTL properties [1]. Each goal can follow a particular pattern depending on its temporal behavior: achieve/cease goals (specified through sometimes in the future/past operators) and maintain/avoid goals (specified through always in the future/past operators).

KAOS also distinguishes goals representing non-functional requirements (soft-goals), whose satisfaction depends on how behavioral goals (representing functional requirements) are achieved. They can influence the application execution, for example if we have different application models that provide the same functionality, and satisfy different soft-goals with a different degree. Two goals may be in conflict, if the achievement of one of them obstructs the satisfaction of the other.

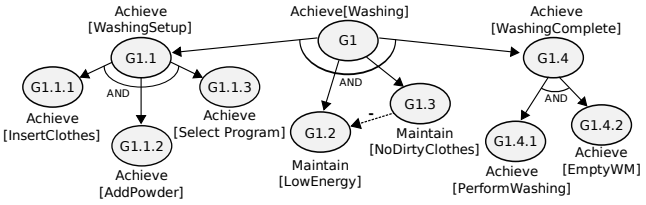



Figure 2. The KAOS goal model for the laundry system

Figure 2 sketches the simplified KAOS goal model for the laundry system. The general objective of the process is to wash clothes (goal G1), which is AND-refined into the following subgoals: setup the washing machine (goal G1.1), complete a washing cycle (goal G1.4), consume a small amount of energy (goal G1.2) and keep the number of dirty clothes equal to zero (goal G1.3). The setup of a washing machine requires that clothes to be washed be inserted in the drum (goal G1.1.1), that powder be added (goal G1.1.2), and that a washing program be selected (goal G1.1.3). Completing a washing cycle requires to start the selected program (goal G1.4.1) and empty the washing machine when it terminates the program execution (goal G1.4.2).

Goals G1.2 and G1.3 are soft goals since their satisfaction depends on "how" their parent goal is achieved. For

[1]They can be specified through operators like *sometimes in the future* ($\Diamond$), *sometimes in the past* ($\blacklozenge$), *always in the future* ($\Box$), *always in the past* ($\blacksquare$), *always in the future until* ($U$), and *always in the past since* ($S$), *in the previous state* ($\bullet$), in the next state ($\circ$).

| Goal | Formal definition |
|---|---|
| G1.1.1 | $@(wm.state = selected) \Rightarrow \Diamond_{t<x} \neg(wm.drumEmpty)$ |
| G1.1.2 | $@(wm.state = selected) \Rightarrow \Diamond_{t<y}(wm.powder)$ |
| G1.1.3 | $(wm.program = "") \wedge @(wm.state = selected) \Rightarrow$ $\Diamond_{t<z}(wm.program <> "" \wedge wm.washDuration > 0)$ |
| G1.2 | $\Box(e \leq E_{MAX})$ |
| G1.3 | $\Box(dirty\_clothes = 0)$ |
| G1.4.1 | $((wm.state = selected) \wedge (\neg wm.drumEmpty)$ $\wedge(wm.program = "") \wedge (wm.powder) \wedge$ $\wedge(\bullet(wm.drumEmpty) \vee \neg(\bullet wm.powder) \vee$ $\vee\neg(\bullet wm.program = ""))) \Rightarrow$ $\Diamond_{t \leq wm.washDuration}(wm.state = completed)$ |
| G1.4.2 | $(wm.state = completed) \Rightarrow$ $\Diamond_{t<w}(wm.program == "") \wedge (wm.state = free) \wedge$ $(\neg wm.powder) \wedge wm.drumEmpty$ |

Table I
KAOS GOALS DEFINITIONS

example goal G1.3 depends on the frequency of the washing programs, since more frequent programs may rapidly reduce the amount of dirty clothes. Moreover goal G1.3 is in conflict with goal G1.2 (see the dotted line in Figure 2) since reducing the number of dirty clothes may imply the selection of washing programs that consume more energy units.

The formal definition of goals is reported in Table II-A[2]. For example, goal G1.1.1 states that if the washing machine's drum is empty and the washing machine is selected to perform a wash cycle ($@(wm.state = selected)$), its drum must be filled with the clothes that have to be washed (condition $\neg wm.drumEmpty$) within $x$ time units.

### *B. Modified KAOS goal model*

Service compositions need to continuously assess their requirements and activate adaptation when needed, to accomodate changes that may happen at the requirement or environmental level. For this reason, we maintain the goal model live at runtime, associating each goal with a *satisfaction* value ($s \in [0, 1]$) that is updated dynamically. Each goal is also associated with a *priority* and a set of *stakeholders* who deem that goal important. The former allows us to resolve conflicts among different adaptation actions. Two adaptations are in conflict if their concurrent application generates incoherent goal models. This way the adaptation associated with the goal having the highest priority will be executed, disabling the others. While *stakeholders* information is adopted to detect the process instances on which an adaptation should be applied (i.e., only those run by the stakeholders associated with the failing goal).

Another drawback of KAOS is the absence of an expressive means to formalize goals. In fact, since LTL formulae are evaluated through a binary value (yes or not), they can only express if a goal is satisfied or not, leaving out the corresponding satisfaction/violation level. As already explained in Section I, this can be particularly important for fuzzy goals that rely on temporal values, quantities or frequencies. In our example, we have two fuzzy goals: G1.2 and G1.3. The definition of goal G1.2 given in Table II-A can only assess whether the energy consumed does not exceed the maximum amount permitted ($E_{MAX}$), and it omits information regarding its distance from $E_{MAX}$. Furthermore the definition of goal G1.3 can only assess whether there are no dirty clothes, and omits a measurement of how many dirty clothes are left.

To express fuzzy goals, we adopt a set of operators, already introduced in RELAX [16] to express non-critical requirements. This notation proposes the following operators: $AS\ EARLY/LATE\ AS\ POSSIBLE\ \phi$ , for temporal quantities; $AS\ CLOSE\ AS\ POSSIBLE\ TO\ q\ \phi$, to assess the proximity of quantities or frequencies ($\phi$) to a certain value ($q$); $AS\ MANY/FEW\ AS\ POSSIBLE\ \phi$ , for quantities ($\phi$). This way goals G1.2 and G1.3 can be redefined in terms of RELAX operators as follows:

$\mathbf{G1.2}$ : $AS\,CLOSE\,AS\,POSSIBLE\,TO\,E_{MAX}\ (E_{MAX} - e)$

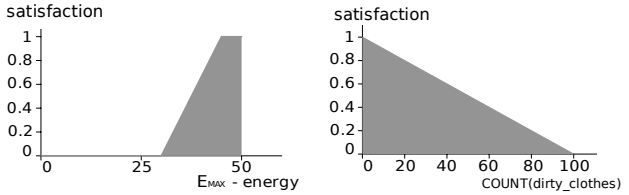
$\mathbf{G1.3}$ : $AS\,FEW\,AS\,POSSIBLE\,COUNT(dirty\_clothes)$

G1.2 expresses that the difference between the maximum amount of energy that can be consumed ($E_{MAX}$) and the energy spent ($e$) must be as close as possible to $E_{MAX}$. In other words, the energy consumed must be as little as possible. While G1.3 asserts that the amount of dirty clothes must be as few as possible.

Assessment of goals G1.2 and G1.3 is established from the result of the membership functions shown below, that assign a satisfaction value between 0 and 1, depending respectively on the amount of energy consumed (Figure 3(a)) or the number of dirty clothes (Figure 3(b)). For example, for goal G1.2 the membership function on the left assigns a satisfaction value depending on the difference between $E_{MAX}$ and the energy consumed . This goal is not satisfied when this difference is less than 30, its satisfaction is comprised between 0 and 1 if the difference is between 30 and 45 and is fully satisfied if the difference is greater than 45. The other membership function is stricter since the requirement is fully satisfied only if the number of dirty clothes is exactly equal to 0, and is less satisfied for all other values. Note that these functions are limited, e.g. the function in Figure 3(b) cannot decrease to $\infty$. The shape of the membership function can be tuned depending on the priority assigned to a goal: the higher the priority is the higher the severity of the membership function will be. The shape of the membership function can also be inserted manually by the requirements engineers.

To propagate the satisfaction value of leaf goals (that can be monitored directly) to their parent goal we adopt the following method:

[2]Notice that operator '@' has the following meaning: $@P \equiv \bullet\neg P \wedge \circ P$




(a) (b)

Figure 3. Membership functions for goals G1.2 (a) and G1.3 (b)

- If a goal $G$ is AND-refined in $k$ crisp goals and $n$ fuzzy goals its satisfaction value is given by:

  $s(G) = (s(C_1) \wedge ... \wedge s(C_k)) * \min(s(F_1), ..., s(F_n))$
  $s(C_i) = 0, 1\ i = 1, ..., k$ (satisfaction value of crisp goals)
  $s(F_i) \in [0, 1]\ i = 1, ..., n$ (satisfaction value of fuzzy goals)

  If at least one of the crisp subgoals is not satisfied, the parent goal is not satisfied. Otherwise the satisfaction of goal $s(G) \in [0, 1]$ depends on the minimum satisfaction value of the fuzzy subgoals ($s(F_i)$). For example, if goal G1.1 and G1.4 are satisfied and the satisfaction value of G1.2 and G1.3 is respectively 0.4 and 0.8, the satisfaction of G1 will be 0.4
- If a goal $G$ is OR-refined in $k$ crisp goals and $n$ fuzzy goals, its satisfaction value will be given by:

  $s(G) = (s(C_1) \vee ... \vee s(C_k)) * \max(s(F_1), ..., s(F_n))$

  If all crisp subgoals are not satisfied, the parent goal is not satisfied. Otherwise the satisfaction $s(G) \in [0, 1]$ depending on the maximum satisfaction value of the fuzzy subgoals.

Other problems may arise if a leaf goal is not directly monitored, e.g. for performance reasons, or probes malfunctioning. To solve this issue we associate each contribution link with a weight. If goal $G_i$ hurts/breaks goal $G$ the weight assigned to the contribution link is negative (comprised between -1 and 0), while since $G_i$ helps/makes goal $G$ the weight assigned to the contribution link is positive (comprised between 0 and 1). Hence if a goal ($G$) is associated with $n$ goals through contribution links, its satisfaction value is calculated as follows:

$s(G) = \sum_{i=1}^{N}(\alpha_i * s(G_i)), \quad -1 \leq \sum_{i=1}^{N} \alpha_i \leq 1$
$\alpha_i$: *weight of the contribution link exiting from goal* $G_i$ *to goal* $G$
$\alpha_i in [-1, 0]$, *if* $G_i$ *hurts* $G$
$\alpha_i in [0, 1]$, *if* $G_i$ *helps* $G$

Users can select the goals they are interested in monitoring to improve system performance. Otherwise monitored goals can be selected depending on the data collected at runtime.

### C. Operationalization

Operationalization [1], [14] is the process that allows us to (semi-automatically) infer a set of operations from the formal definition of one or more goals. It improves requirements traceability, easing the generation of the functional and supervision models that must be applied at runtime.

An operation is an input-output relationship over a set of objects. Operations are specified depending on their effects on the domain: domain preconditions (`DomPre`) and post-conditions (`DomPost`). Operations are also specified through required pre-conditions (`ReqPre`), triggering pre-conditions (`TrigPre`), and required post-conditions (`ReqPost`). Required preconditions define the application states in which an operation is allowed to be performed. Triggering conditions define the application states that activate the operation execution. Required post-conditions define additional conditions that must be true after the execution of an operation.

Figure 4 shows the result of the operationalization applied to our case study. For example operation $Select\ Program$ selects a washing program if the washing program has not been selected yet ($ReqPre :\ wm.program = ""$). This operation is activated if in the previous states the washing program is not selected and in the current state the washing machine is selected to perform a washing cycle ($TrigPre :\ wm.state = selected$). Furthermore the operation is activated by event $Sel\_prog(p)$ which signals the invocation of an external partner service which perform the program selection. To achieve the satisfaction of goal G1.1.3 the name and the duration of the selected washing program $p$ have to be assigned to the washing machine ($ReqPost :\ wm.program = p.name \wedge wm.washDuration = p.duration$).

Goals G1.2 and G1.3 rely on all the operations shown in Figure 4. Hence, to enforce their satisfaction we need to add to each operation the following required post-conditions:

$\mathbf{ReqPost_{G1.2}}$ :
$\Box(AS\,CLOSE\,AS\,POSSIBLE\,TO\,E_{MAX}(E_{MAX} - e))$

$\mathbf{ReqPost_{G1.3}}$ :
$\Box(AS\,FEW\,AS\,POSSIBLE\,COUNT(dirty\_clothes))$

This operationalization does not claim to be complete. Instead its purpose is to highlight the links between the goals and the operations of the real application.

### D. Adaptive goals

Our goal model represents adaptation strategies through *adaptive goals* that are able to trigger suitable adaptation actions depending on the satisfaction of crisp/fuzzy goals. Adaptation actions can change the system's goal model in different ways: add/remove goals, modify goals' definitions (e.g, changing the membership functions of fuzzy goals), add/remove/modify operations, and add/remove agents in charge to perform some operations.

Adaptations make the process move to a new goal model that also includes adaptive goals. This way, in every stage of its evolution the process is able to perform adaptation.

**Operation: Insert Clothes**
**Input:** wm: WashingMachine
**Output:** wm: Washingmachine
**DomPre:** wm.state=selected
**TrigPre:** (wm.drumEmpty)
**ReqPost:** ¬wm.drumEmpty
**(FOR G1.1.1)**

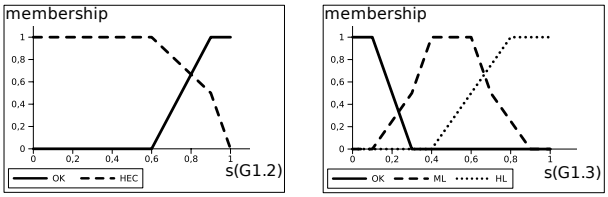
**Operation: Insert Powder**
**Input:** wm: WashingMachine
**Output:** wm: Washingmachine
**DomPre:** wm.state = selected
**TrigPre:** ¬(wm.powder)
**ReqPost:** wm.powder
**(FOR G1.1.2)**

**Operation: Select Program**
**Input:** wm: WashingMachine, p: WashingProgram
**Output:** wm: WashingMachine
**DomPre:** wm.state = selected
**ReqPre:** wm.program = ""
**TrigPre:** Sel_Prog(p)
**ReqPost:** wm.program= p.name ∧
**(FOR G1.1.3)** wm.washDuration = p.duration

**Operation: Wash**
**Input:** wm: WashingMachine
**Output:** wm: WashingMachine
**DomPre:** wm.state = selected
**DomPost:** wm.state = completed
**TrigPre:** start_wash(t)
**ReqPre:** wm.powder ∧ ¬wm.drumEmpty ∧
¬wm.program = ""
**ReqPost:** wm(t').sens="green" ∧
**(FOR G1.4.1)** t' -t < wm.washDuration

**Operation: Empty WM**
**Input:** wm: WashingMachine
**Output:** wm: WashingMachine
**DomPre:** wm.state = completed
**DomPost:** wm.state = free
**ReqPre:** wm.sens = "green"
**ReqPost:** ¬(wm.powder) ∧ wm.drumEmpty ∧ wm.program = ""
**(FOR G1.4.2)**

Figure 4. Operations each requirement relies on




(a) (b)

Figure 5. Membership functions of adaptive goals A1 (a) and A2 (b)

Adaptive goals are associated with a set of membership functions identifying different process' operative levels. In our example, we have adaptive goal A1 associated with two membership functions, shown in Figure 5(a). These membership functions devise operative level normal (OK) and high energy consumption (HEC), depending on the satisfaction of goal G1.2. In fact if this satisfaction level is higher than 0.8, the process is working correctly and no adaptation has to be performed[3]. Otherwise the process is in operative level HEC, associated with an adaptation action (A(HEC)). This adaptation modifies operation $Select\ Program$ changing its required post-condition to select economic programs having a duration shorter than $2 - HEC(s(G1.2))$ hours.

$$\mathbf{ReqPost_{G1.1.3}} : wm.program = p.name) \wedge$$
$$wm.washDuration = p.duration \wedge$$
$$\wedge p.duration \leq 2 - HEC(s(G1.2))$$

This example also highlights how adaptation strategies can be tuned depending on the satisfaction level of stated requirements. In fact in our case the duration of a selected washing program will depend on the satisfaction level of goal G1.2. Obviously, if the violation is too sever it may happen that no suitable washing programs can be found.

Our model also includes adaptive goal A2, associated with three operative levels (see Figure 5(b)), depending on the satisfaction of goal G1.3: normal (OK), medium load (ML) and high load (HL). If the satisfaction of goal G1.3 is higher than 0.7, the system is working properly and no adaptation has to be performed[4]. If the satisfaction level of goal G1.3 is between 0.2 and 0.7, the system is in operative level ML. While since the satisfaction level of goal G1.3 is lower than 0.2 the system is in operative level HL.

Adaptation associated with operative level ML (A(ML)) relaxes goal G1.2, making its membership function less strict. This way we reduce the number of times in which adaptation A(HEC) has to be executed, avoiding the selection of time consuming washing programs. While adaptation associated with operative level HL (A(HL)) adds a new goal to the system (G1.5) that asserts to adopt another washing machine, if the current one is already selected and there are other clothes to wash. Goal G1.5 is defined as follows:

$$\mathbf{G1.5} : wm, wm' : WashingMachine$$
$$(dirty\_clothes > 0 \wedge \neg(wm.state = free) \wedge \exists wm'$$
$$(\neg(wm.id = wm'.id) \wedge wm'.state = free)) \Rightarrow$$
$$\Diamond(wm'.state = selected)$$

Note also that since goal G1.2 and G1.3 are in conflict, they must have different priorities. This way only adaptations triggered by one of them will be performed.

So far we described permanent adaptations, which change the goal model permanently. We also provide transient adaptations that temporarily substitute a failing goal with other ones. Operations corresponding to these temporary

[3]The function associated with operative level OK has an higher membership ( $\geq 0.6$) than the membership of the function associated with operative level HEC ($\leq 0.6$).

[4]The function associated with operative level OK has an higher membership ($> 0.6$) than that of the other functions

goals have to be executed right after the violation is detected, requiring to block the system when monitoring and diagnosis are performed. For example, if goal G1.4.1 is violated because the washing machine turns off suddenly during the washing cycle, the system switch to a new transient goal (G1.6) that asserts to turn on the washing machine.

$$\mathbf{G1.6}: wm.state = off \wedge \blacksquare_{t-k} \neg(wm.sens = green) \Rightarrow \Diamond(wm.state = selected \wedge \neg(wm.powder) \wedge \wedge \neg(wm.drumEmpty) \wedge wm.program = \text{""})$$

This goal can be operationalized as follows:

**Operation:** Turn On
**Input:** wm: WashingMachine
**Output:** wm: WashingMachine
**DomPre:** $wm.state = off$
**DomPost:** $wm.state = selected$
**TrigPre:** $start\_wash(t-k) \wedge \blacksquare_{t-k}(wm.sens = \text{"}green\text{"}) \wedge @(wm.state = off)$
**ReqPost:** $\Diamond_{t<v} wm.program = \text{""} \wedge \neg(wm.powder)$

A transient goal is also associated with a return state. As we will see in Section III-1, in this case the execution is resumed to the state in which the drum is full, there is no powder and a program has to be selected. Transient goals can be monitored directly and in case they are not satisfied may trigger the enactment of other transient goals. Note that in this case the process does not move to a new goal model but resumes the old one right after the adaptation completes.

## III. Runtime Elements

This section illustrates the key elements of the runtime infrastructure and provides some hints on the functional and supervision models.

*1) Functional Model:* A functional model is an abstract process composed of variables, activities (to perform) and messages (exchanged with partner services). These elements have a direct mapping onto objects, events, and operations in the goal model, respectively. In our example, the process variables are `WashingMachine` and `WashingProgam`, shown below.

***WashingMachine*** *(id: ID, state: {off, free, selected, completed}, powder:true, false, drumEmpty: true, false, program: String, washDuration: int)*

***WashingProgram*** *(id: ID, duration: int, name: String)*

The former is described by the following attributes: `id` is an identifier, `state` indicates if it is off, free, selected for a washing cycle, or if a washing cycle completes, `powder` shows if the powder container is not empty, `drumEmpty` says whether there are clothes in the drum, and `program` communicates the current washing program. The latter is described by the following attributes: `id` is an identifier, `duration` stores the duration of the current washing program, `name` contains its name.

Activities imply an interaction of the software system with the surrounding environment (sensors and actuators) and interactions with external components. These interactions may be highlighted by the presence of events representing the exchange of messages with the partner services. For example, event $Sel\_Prog(p)$ of operation $Select\ Washing\ Program$ and event $Start\_Wash$ of operation $Wash$ are mapped onto messages sent and received to/from an external agent in charge of selecting the program.

The workflow also defines the order in which groups of operations must be executed (sequentially or in any order) depending on their pre- and post-conditions. An operation $Op1$ must be executed after operation $Op2$ if and only if its preconditions are implied by the post-conditions of $Op2$:

$$DomPost(Op1) \wedge ReqPost(Op1) \Rightarrow DomPre(Op2) \wedge ReqPre(Op2) \wedge TrigPre(Op2)$$

This means that, for example, operation $Empty\ Washing\ Machine$ must be executed after operation $Wash$.
While two operations (e.g., $Op1$ and $Op2$) can be executed in any order if the following conditions

- $(ReqPost(Op1) \wedge DomPost(Op1) \implies ReqPre(Op2) \wedge TrigPre(Op2) \wedge DomPre(Op2)$
- $(ReqPost(Op2) \wedge DomPost(Op2) \Rightarrow ReqPre(Op1) \wedge TrigPre(Op1) \wedge DomPre(Op1))$

Again, this means that operations $Insert\ \ Powder$, $Insert\ Clothes$, and $Select\ Program$ can be executed in any order. The same checks must be performed for those operations derived from goals added after adaptation. For example, in our case we have to check whether operation $Turn\ On$ can be executed after operation $Select\ Program$ and before operation $Insert\ Clothes$. These rules allow us to build an executable process, in which the operations are put in the proper order (Figure 6).

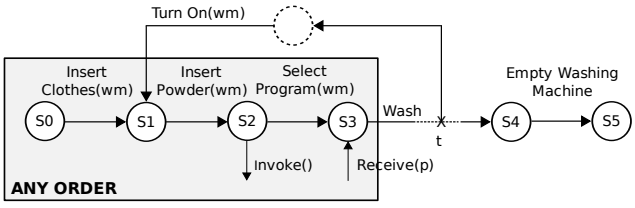



Figure 6. Workflow of the laundry system.

*2) Supervision model:* A supervision model comprehends directives for data collectors, monitors, and adaptors. They retrieve information about the actual behavior of the system, evaluate the degree of goals' satisfaction, and apply adaptation, if needed.

**Data Collection.** A data collector provides monitoring components with the necessary information to evaluate goals'

satisfaction. Data may come from different kinds of probes. We can have data acquired from external sensors (e.g., washing machine parameters), process internal variables, messages exchanged with partner services, and historical data collected with previous process executions. Moreover, data collectors are able to mask the heterogeneity and protocols adopted by different probes, and provide monitors with homogeneous information. Data collectors must be properly configured to collect the information of interest. Then, they can provide monitors with retrieved data in the following ways: (a) as soon as they are collected (push), every specified time frame (periodic push), and when required by monitoring components (pull). In our example, to evaluate the satisfaction of goal G1.1.3, we need to collect data from the surrounding environment (`wm.state`, `wm.program`) in a push way. Furthermore these data must be collected for the states in which the process executes the operations associated with the monitored goal (state `S2`).

**Monitoring.** The runtime infrastructure comprises a set of monitoring plug-ins able to evaluate LTL expressions or membership functions over a set of states. Each monitoring plug-in works with some data collectors and is configured to check the requirement's satisfaction in particular states of the process. Requirements evaluation is performed by translating the formal definition of a goal into a directive that can be processed by the monitoring component. For example, to verify goal G1.1.3, we need an LTL checker that works with data `wm.state` and `wm.program`. While since fuzzy goal G1.2 could be ideally verified in each state of the application, we must manually select in which states we want to collect data about energy consumption to trigger the evaluation of the corresponding requirement. The evaluation of a monitoring constraint may require that the process be blocked when a transient adaptation must be executed since a monitored goal is violated. This happens for goal G1.4.1: when washing completes the process is kept blocked to evaluate its satisfaction. If the goal is not verified, operation `Turn on(wm)` is triggered.

**Adaptation.** Adaptation is triggered by the evaluation of the membership functions associated with adaptive goals. This happens when goal satisfaction levels are updated at runtime. At this level, adaptation action specified in the adaptive goals are translated into the following lower level actions: `identify_ agent` detects a concrete partner service (endpoint) that must perform an operation, `change_state` modifies the process state, `change_protocol` modifies the order of the process activities, `add_op`/`remove_op` adds/removes the process activities corresponding to specific operations detected in the goal model, and `change_directives` changes the adaptation directives given to data collectors, monitors, and adaptors. Adaptation actions defined in the adaptive goals can have different impacts on the workflow and supervision components. For example, action `change_agent` is translated into the lower level operation `identify_agent`, and it may also generate changes in the functional model (actions `change_protocol`, `add_op`, `remove_op`), or in the way supervision is performed (action `change_directives`). Each adaptation can be safely performed in specific execution points, called *quiescent states* [18]. For this reason, we must control the execution of the system and keep it blocked when an adaptation needs to be executed and the application reaches one of the quiescent states. For example, suppose we want to perform adaptation associated with operative level ML of adaptive goal A2. This action will trigger lower level action `change_directives` that modify the way in which monitoring components evaluate the membership function that assesses the satisfaction of goal G1.2. In this case, all execution points are quiescent states, since this adaptation does not impact on the business logic and the evaluation of the membership function must not be done on a temporal window. While since our adaptation changes the order in which operations `Insert Powder` and `Select Program` are executed, S0 and S1 (Figure 6) become quiescent states.

## IV. Related Work

Different works have already tried to bridge the gap between requirements specifications and the underlining service compositions. For example, Kazhamiakin et al. [10] adopt Tropos [7] to specify the objectives of the different actors involved in a choreography. Tropos tasks are refined into messages exchanges, suitable annotations are added to express conditions on goal creation and fulfillment, and assume/guarantee conditions are added to the tasks delegated to partner services. These elements enable the generation of an annotated BPEL process that can be statically verified, applying model checking. Instead, we adopt requirements to derive suitable supervision directives to monitor and adapt service compositions at runtime.

We adopt fuzzy goals [12] to tune the adaptation activities according with the satisfaction levels of the stated goals. Fuzzy goals have been used for several objectives, e.g., the selection of COTS components during requirements elicitation [2]. In fuzziness to express and assess the satisfaction degree of requirements with the idea of introducing a bit of vagueness, the possibility of preventing some violations, and the ability to tolerate small/transient deviations. This way monitoring requirements satisfaction level at runtime, allows one to effectively select and tune the adaptation options accordingly.

The concept of requirements monitoring was originally proposed by Fickas et al. [6]. The authors adopt a manual approach to derive monitors able to verify requirements' assumptions at runtime. Mylopolus et al. [15] use the generation of log data to infer the denial of requirements

and detect problematic components. Diagnosis is inferred automatically given assumptions on which requirements can fail. Robinson [13] distinguishes between the design-time model, where business goals and their possible obstacles are defined, and the runtime model, where logical monitors are automatically derived from the obstacles and are applied onto the running system. This approach requires diagnostic formulae to be generated manually from obstacle analysis. Despite a lot of work focused on requirements monitoring only a few of them provide reconciliation mechanisms when requirements are violated. Wang et al. [15] generate system reconfiguration guided by OR-refinements of goals, after a goal violation is diagnosed. They choose the configuration that contributes most positively to the non-functional requirements of the system and also has the lowest impact on its current configuration. Our proposal builds on existing monitoring solutions and emphasizes the separation among probing, analysis, and reaction to provide a single homogeneous framework in which the different approaches can be seamlessly integrated. This way, monitoring and adaptation capabilities are not hard-coded in the infrastructure, but can be selected and customized according to the actual needs. We exploit the operation model to trace requirements onto the underlying implementation and detect the variables that must be collected and the constraints that must be evaluated.

## V. Conclusions

This paper presented an innovative goal-based approach for specifying the requirements and adaptation capabilities of service compositions and map them onto the underlining implementation. Our proposal allow to specify some adaptation features at design time (e.g., the strategy to apply) and customize adaptation details at runtime depending on the specific execution context and an experimented violations.

In the future work we plan to explicitly embed the goal satisfaction levels in the goals model and detect different adaptation strategies depending on the satisfaction levels. We also think to include other actions [9] (e.g. send an email message, log variables' state, activate a human task or other activities coded in java).

## Acknowledgements

This research has been funded by the European Commission, Programmes: IDEAS-ERC, Project 227977 SMScom, and FP7/2007- 2013, Projects 215483 S-Cube (Network of Excellence).

## References

[1] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Learning Operational Requirements from Goal Models. In *Proc. of the 31st Int. Conf. on Softw. Eng.*, pages 265–275, 2009.

[2] C. Alves, X. Franch, J. P. Carvallo, and A. Finkelstein. Using Goals and Quality Models to Support the Matching Analysis During COTS Selection. In *Proc. of the 4th Int. Conf. on COTS-Based Softw Sys.*, pages 146–156, 2005.

[3] Anne Dardenne and Axel van Lamsweerde and Stephen Fickas. Goal-Directed Requirements Acquisition. *Sci. Comput. Program.*, 20(1-2):3–50, 1993.

[4] L. Baresi, S. Guinea, and L. Pasquale. Integrated and Composable Supervision of BPEL Processes. In *Proc. of the 6th Int. Conf. of Serv. Orien. Comp.*, pages 614–619, 2008.

[5] K. S. Chan, J. Bishop, J. Steyn, L. Baresi, and S. Guinea. A Fault Taxonomy for Web Service Composition. In *Proc. of the 3rd Int. Work. on Eng. Serv. Orien. Appl.*, pages 363–375, 2007.

[6] S. Fickas and M. S. Feather. Requirements Monitoring in Dynamic Environments. In *Proc. of the 2nd Int. Symp. on Req. Engineering*, page 140, 1995.

[7] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. Specifying and analyzing early requirements in Tropos. *Req. Eng.*, 9(2):132–150, 2004.

[8] H. J. Goldsby, P. Sawyer, N. Bencomo, B. H. C. Cheng, and D. Hughes. Goal-Based Modeling of Dynamically Adaptive System Requirements. In *Proc. of the 15th Int. Conf. on Engineering of Computer-Based Systems*, pages 36–45, 2008.

[9] JBoss Community. Drools Flow. http://www.jboss.org/drools/drools-flow.html.

[10] R. Kazhamiakin, M. Pistore, and M. Roveri. A Framework for Integrating Business Processes and Business Requirements. In *Proc. of the 8th Int. Conf. on Ent. Distr. Obj. Comp.*, pages 9–20.

[11] A. Lapouchnian, Y. Yu, S. Liaskos, and J. Mylopoulos. Requirements-driven design of autonomic application software. In *Proc. of the 2006 Conf. of the Center for Adv. Studies on Coll. Research*, page 7.

[12] X. F. Liu. Fuzzy Requirements. *IEEE Potentials*, pages 24–26, 1998.

[13] W. N. Robinson. Monitoring Web Service Requirements. In *Proc. of the 11th Int. Req. Eng. Conf.*, pages 65–74, 2003.

[14] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[15] Y. Wang, S. A. Mcilraith, Y. Yu, and J. Mylopoulos. Monitoring and Diagnosing Software Requirements. *Automated Software Engg.*, 16(1):3–35, 2009.

[16] J. Whittle, P. Sawyer, N. Bencomo, and B. H. C. Cheng. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *Proc. of the 17th Int. Req. Eng. Conf.*, 2009.

[17] E. S.-K. Yu. *Modelling strategic relationships for process reengineering*. PhD thesis, Toronto, Ont., Canada, 1996.

[18] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *Proc. of the 28th Int. Conf. on Soft. Eng.*, pages 371–380, 2006.