Live Goals for Adaptive Service Compositions

Luciano Baresi and Liliana Pasquale
Politecnico di Milano - Dipartimento di Elettronica e Informazione
piazza L. Da Vinci, 32 – 20133 Milano, Italy
baresi|pasquale@elet.polimi.it

ABSTRACT

Service compositions represent an important family of self-adaptive systems. Though many approaches for monitoring and adapting service compositions have already been proposed, a clear connection with the motivations for using such techniques is still missing. To this aim we address self-adaptation from requirements elicitation down to execution. In this paper, we propose to enrich existing goal models with adaptive goals, responsible for the actual evolution/adaptation of the goal model at runtime. We also translate the goal model with both conventional and adaptive goals, into the actual functionality provided by the system and the adaptation policies needed to make it self-adapt.

Categories and Subject Descriptors

D.2 [**Requirements/Specifications**]: Methodologies

General Terms

Design

Keywords

Requirements, Supervision, Goals, Service compositions

1. INTRODUCTION

Service compositions represent an important family of self-adaptive systems. One of their pillars is the capability of adapting systems' behavior at runtime, but these changes must be disciplined to keep systems in acceptable states. Even if the architectural paradigm intrinsically embeds runtime adaptation, we need proper means to specify how a system should self-adapt, how these capabilities are implemented, and what supporting infrastructure oversees their application.

If we think of BPEL (Business Process Execution Language, [1]) processes, which are probably the most common service compositions in these days, there have been many proposals interested in the technical details needed to support monitoring [9, 3, 13] and adaptation [13, 7, 9]. However none of them addressed the issue of understanding and specifying the actual adaptation capabilities required by a process. Many approaches contribute to identifying *how* to enrich BPEL processes with self-adaptation capabilities, but only few address the issue of *what* capabilities are needed, and *when* they should be activated.

To fill this gap, we propose a complete approach to reason on self-adaptation capabilities from requirements elicitation down to execution. We extend the KAOS goal model [17] with the concept of *adaptive goal*s. These goals are responsible for the actual evolution/adaptation of the goal model at runtime, and specify the countermeasures taken when a *conventional* goal is violated. To cope with runtime unexpected changes, goals are conceived as *live* abstractions able to change dynamically.

Since each process instance has its own execution context, adaptive goals see it as domain variables and predicate on them. We also establish and maintain the relationships between adaptations and affected process elements (variables, partner links, activities) to predict the effects of the different adaptation strategies and also to be able to reason on the consistency of adapted processes.

Both conventional and adaptive goals are then translated semi-automatically into two models: a *functional* model, which is responsible for the actual functionality provided by the system, and a *supervision* model, which is in charge of the adaptation capabilities. The former represents the set of compositions that satisfy stated requirements, while the latter defines how to assess the requirements of interest and "force" their satisfaction.

The composition (process) that best fits the set of available services is then transformed into a complete BPEL process. A dedicated infrastructure [9], which comprises probes, monitors and actuators, augments a conventional BPEL engine with self-adaptive capabilities and executes these compositions.

The paper focuses on the upper part of the proposal and describes the requirements model and the derivation process. The main concepts are explained through a simple application for organizing dinners out with friends. Besides booking restaurants, the application must also cope with undesired events like participants that are late or do not show up.

The rest of the paper is organized as follows. Section 2 introduces the KAOS goal model, which acts as starting point for our proposal. Section 3 describes our notion of adaptive goals. Section 4 and Section 5 illustrate the (semi-)

SEAMS '10, May 2-8, 2010, Cape Town, South Africa

automatic derivation of the functional model and the supervision model, respectively. Section 6 discusses some related approaches and Section 7 concludes the paper.

2. KAOS

KAOS [17] proposes a set of models to reason about different views of a system and ease the generation of a direct mapping onto the underlying implementation (BPEL processes, in our case). In this work, we leverage the goal, object and operation model.

The goal model defines the main objectives (functional and non-functional requirements) the application should meet. Goals can be refined into conjoined subgoals (AND-refinement) or into alternative combinations of subgoals (OR-refinement). The satisfaction of a parent goal depends on the achievement of all (AND-refinement) or at least one (OR-refinement) of its subgoals. Goals are formally rendered as LTL (Linear Temporal Logic) expressions[1].Two goals are in conflict when the achievement of one obstructs the satisfaction of the other.

Goal refinement can be accomplished through formal rules [17], and terminates when the goal can be "operationalized", that is, it can be decomposed into a set of operations. Operations are input-output relationships over a set of objects (*In/Out*). They are specified through domain pre- and post-conditions, where the former (*DomPre*) characterize the state of domain variables before applying the operation, and the latter (*DomPost*) define a relationship between the state of an entity before and after applying the operation. One can also add required pre-conditions (*ReqPre*), triggering pre-conditions (*TrigPre*) and required post-conditions (*ReqPost*). Required pre-conditions define those states in which an operation is allowed to be applied. Required triggering conditions define those states in which the operation must be immediately applied, provided the domain and required pre-conditions are true. Required post-conditions define additional conditions the application of an operation must satisfy.

Figure 1 sketches a goal model for the example application whose main goal is to organize sucessful dinners[2]. This requirement implies that the system collects the preferences regarding the maximum cost of a restaurant (goal *G1.1*), finds a restaurant that matches these preferences (goal *G1.2*), book it (goal *G1.3*), and "hope" that participants are on time (goal *G1.4*).

The vocabulary used in the goal model introduces the following entities: *Preference(id, cost, time, state: {empty, filled})*, *Rest_List(id, restaurants: Set<Restaurant>, state: {empty, filled})*, *Restaurant(id, name, address, cost)*, *Sel_Restaurant(id, restaurant, arrTime, bookTime, state: {empty, selected, booked, present, not_present })*.

For example, if we assume that the restaurants (*l.restaurants*) that match customers' preferences (p) must be retrieved within y time units, goal *G1.2* becomes:

$$l : Rest\_List(@(p.state = filled) \Rightarrow \Diamond_{t \leq y} \exists s : Sel\_Restaurant(s.state = selected \wedge \exists r : Restaurant \in l.restaurants(r.id = s.restaurant))$$

where $@P$ means $(\bullet(\neg P) \wedge \circ P)$. This goal can easily be achieved. through operation *Op2 [Show possibilities]*, which shows the set of available restaurants:

Name: *Op2*
In/Out: $p : Preference, l : Rest\_List$
DomPre: $l.state = empty$
DomPost: $l.state = filled$
TrigPre: $@(p.state = filled)$
ReqPost: $show\_possibilities \wedge \exists r : Restaurant(r \in l) \wedge \forall r \in l(r.id \neq null \wedge r.name \neq "" \wedge r.address \neq "")$

and operation *Op3 [Select restaurant]* for the actual selection:

Name: *Op3*
In/Out: $l : Rest\_List, s : Sel\_Restaurant$
DomPre: $s.state = empty$
DomPost: $s.state = selected$
TrigPre: $show\_possibilities$
ReqPost: $\exists r \in l(r.id = s.restaurant)$

Event *show_possibilities* represents the fact that the system shows the possible restaurants to its customers.

Goal *G1.2* is also associated with a non-functional requirement (goal *G1.2.1*) that says that the cost of an average dinner in any of the selected restaurants cannot be more than 15$ greater than the threshold set by the customer:

$$l : Rest\_List, p : Preference\ (@(l.state = filled) \Rightarrow \forall r : Restaurant(r \in l \wedge r.cost - p.cost \leq 15\$))$$

Furthermore the restaurant selected by the customer (goal *G1.3*) must to be booked within z time units, and the booking time must match that specified by the customer. If the restaurant is full, the dinner is canceled (goal *G1.3*):

$$s : Sel\_Restaurant, p : Preference\ (@(s.state = selected) \Rightarrow \Diamond_{t \leq z}(get\_confirm \Rightarrow (s.state = booked \wedge s.bookTime = p.time) \wedge not\_avail \Rightarrow s.state = canceled))$$

The goal is then achieved through operations *Op4 [Send*

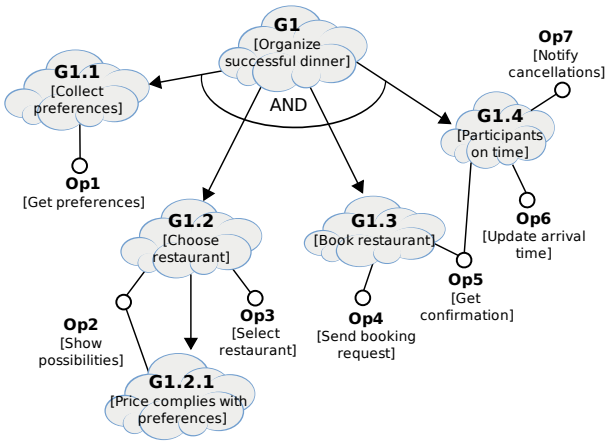

Figure 1: The goal model of the Dinner Planner.

[1]For reader's convenience, LTL operators are: *in the previous state* ($\bullet$), *in the next state* ($\circ$), *sometimes in the future* ($\Diamond$), *sometimes in the past* ($\blacklozenge$), *always in the future* ($\Box$), *always in the past* ($\blacksquare$), *always in the future unless* (W), *always in the past back to* (B), *always in the future until* (U), and *always in the past since* (S). Operators B and S are seldom adopted, hence we will not consider them in the paper.

[2]Lack of space does not allow us to present the complete formalization of our example. The entire specification is available at http://home.dei.polimi.it/pasquale/papers/TRSEAMS.pdf

booking request], which sends the booking request to the selected restaurant:

Name: *Op4*
In/Out: $s : Sel\_Restaurant$
TrigPre: $@(s.state = selected)$
ReqPre: $\blacksquare(\neg(req\_booking))$
ReqPost: $req\_booking \wedge \Diamond_{t \leq x}(get\_confirm)$

and operation *Op5 [Get confirmation]*, in charge of getting a confirmation of the reservation.

Name: *Op5*
In/Out: $s : Sel\_Restaurant$
DomPre: $s.state = selected$
DomPost: $s.state = booked$
TrigPre: $get\_confirm \vee get\_unvail \wedge$
$\neg(get\_confirm \wedge get\_unvail)$
ReqPost $(get\_confirm \Rightarrow s.bookTime = arrTime) \wedge$
$(get\_unaval \Rightarrow s.state = canceled)$

Event *req_booking* represents the request issued to the restaurant to book a table at a given time, while *get_confirm* and *get_unavail*, represent a positive and negative answer for the restaurant, respectively.

There is also another non-functional requirement that states that participants must be at the restaurant on time, or with a delay of no more than 10 minutes (goal *G1.4*):

$s : Sel\_Restaurant(@(s.state = booked) \Rightarrow$
$\Diamond(s.arrTime \neq null \wedge s.arrTime - s.bookTime \leq 10m)$

This requirement depends on operations *Op5*, already defined, and *Op6 [Update arrival time]* and *Op7 [Notify cancellations]*. As shown above, *Op5* defines the time at which the dinner will start. *Op6* registers the participants' arrival time:

Name: *Op6*
In/Out: $s : Sel\_Restaurant$
DomPre: $s.state = booked$
DomPost: $s.state = attended$
TrigPre: $part\_arrived$
ReqPost: $s.arrTime \neq null \wedge$
$s.arrTime - s.bookTime \leq 10m$

while *Op7* is used to notify that (some) participants will not come. This is approximated by using a delay greater than one hour.

Name: *Op7*
In/Out: $s : Sel\_Restaurant$
DomPre: $s.state = booked$
DomPost: $s.state = not\_attended$
TrigPre: $time - s.bookTime \geq 1h$
ReqPre: $\blacksquare(\neg(part\_arrived))$

Event *part_arrived* signals the arrival of participants to the restaurant.

Since we are interested in binding each operation with a particular set of partner services responsible for its execution, our agents are the services in charge of executing the different operations.

3. ADAPTIVE GOALS

As already said, once we have a BPEL process that satisfies stated goals at the beginning, it may violate them while the execution proceeds [5]. We can have 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 or SLA violations, delayed response times). All these problems cannot be easily foreseen at design time, as it is done in KAOS. Moreover, since the same process can be replicated in different instances run in different contexts, violations could be different from instance to instance and the same applies to the possible adaptations.

In this context, *adaptive goals* are introduced as a means to conveniently describe a set of possible ways adaptation can be carried out. They identify the conditions that may cause a goal violation (monitoring), together with the set of possible countermeasures that can eliminate/reduce/avoid it (recovery). Each adaptive goal is associated with an *obstacle* that represents the violation of the leaf goal G it is associated with, and thus it is the negation of G's definition[3].

Each strategy comes with an *additional condition* that describes the context in which the violation may take place and better specifies when the strategy must be triggered. These conditions cannot contradict their goals' obstacles; they can only impose further constraints to differentiate the applicability of the different strategies associated with the same goal.

Strategies have an *objective*, which can be: enforce a substitute goal, enforce the original goal, avoid a goal violation, and enforce a weaker version of the failing goal. These objectives allow one to assess the success of the strategies at runtime and evaluate whether others must be performed.

To achieve this objective a set of basic actions – strategy – have to be applied on the goal model to modify it in different ways: add or remove a goal, modify the definition of a leaf goal, add or remove operations, modify the pre- and post-conditions of operations, add or remove entities, modify the definition of objects, and change adopted agents (i.e., the partner services at architectural level).

Each strategy is also associated with a *scope*, *timeliness*, and *severity*. The scope says if the strategy is applied on a single process instance or on all the instances (i.e., on the process definition itself). The timeliness specifies if the strategy is synchronous or asynchronous with respect to the process execution. The severity can be: low, for those strategies that only try to prevent goals' failures; medium, for those strategies that try to enforce a weaker version of the failing goal; and high, for those strategies that try to enforce the failing goal. A strategy with a low severity makes sense for goals with low criticality or for goals that are supposed to be violated soon. A strategy with high severity is suitable for critical goals or for those goals that largely deviate from the desired objectives. A strategy with medium severity is used if a goal cannot be completely satisfied after it is violated, or to avoid conflicts with higher critical goals.

Section 5 shows how these strategies are translated into adaptations on the BPEL process and its supervision rules. At runtime, the actual selection of the strategy is based on the conditions associated with the different alternatives and on their severity level.

Figure 2 introduces some adaptive goals for the example goal model. Goal $G1.2.1$ is associated with adaptive goal $AG1.2.1$ that suggests two strategies ($S1$ and $S2$). The additional conditions, only partially presented here for lack of space, says that strategy 1 can only be applied when the

[3]Note that in this paper we only consider violations of leaf goals since violations of higher level goals can always be reduced to violations of leaf ones [17].

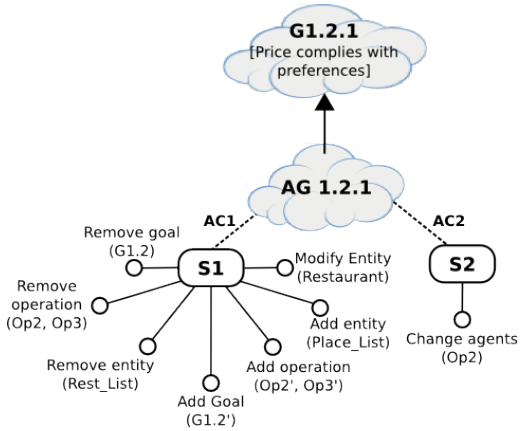

Figure 2: An example adaptive goal.

cost required by the customer is lower than 20$ ($AC1$):

$p : Preference(p.price \leq 20)$

The operations in $S1$ remove goal $G1.2$ and add goal *G1.2' [Choose Takeaway]*, which finds takeaways and, we assume, these are always the cheapest solutions. Removing goal $G1.2$ also implies the removal of entity *Rest_List* and attribute *cost* from entity *Restaurant*, since they would not be used anymore in the goal model. The addition of goal $G1.2'$ requires that attribute *type* be added to entity *Restaurant* and entity *Place_List(id, restaurants: Set<Restaurant>, state: {empty, filled})* be added to the goal model.
Goal $G1.2'$ is defined as follows:

$l : Place\_List, p : Preference\ (@(l.state = filled) \Rightarrow$
$\forall r : Restaurant(r \in l \wedge r.type = take\_away))$

Goal $G1.2'$ is operationalized by operations *Op2'* and *Op3'*. The former shows available takeaways, while the latter gets the one selected by the customer. Operations *Op2'* and *Op3'* are defined as follows:

Name: *Op2'*
In/Out: $p : Preference, l : Place\_List$
DomPre: $l.state = empty$
DomPost: $l.state = filled$
TrigPre: $@(p.state = filled)$
ReqPost: $show\_possibilities \wedge \exists r : Restaurant(r \in l) \wedge$
$\forall r \in l(r.id \neq null \wedge r.name \neq "" \wedge r.address \neq "" \wedge$
$r.type = take\_away)$

Name: *Op3'*
In/Out: $l : Place\_List, s : Sel\_Restaurant$
DomPre: $s.state = empty$
DomPost: $s.state = selected$
TrigPre: $show\_possibilities$
ReqPost: $\exists r \in l(r.id = s.restaurant)$

Strategy $S1$ aims to enforce a modified version of the original goal, and for this reason its severity level is medium. Strategy $S2$ has no additional conditions associated with it (i.e., its additional conditions will always hold true), and thus it can be taken when the obstacle associated with goal $G1.2.1$ is true. It changes the agents in charge of collecting the selected restaurant (i.e., the one that performs *Op2*). Furthermore the severity level of this strategy is high, since it enforces the original version of goal $G1.2.1$.

4. FUNCTIONAL MODEL

The functional model comprises the skeleton of the BPEL process and the contracts its partner services must comply with. The pre- and post-conditions of the operations used to refine goals guide the definition of contracts. The actual identification of the concrete services is based on available services. In some cases, available services are enough to cover all operations. In other cases, one or more operations are not matched properly, and new services must be created.

Since the implementation of new services should be nothing new, we only consider the problem of matching operations and services, with the assumption that available services cover all operations. More precisely, the matching process works as follows:

1. We define the lifecycle of each entity by representing its state transitions, and identify the operations that cause such transitions and the precedence rules among these operations. This step is fully automated.

2. We define a possible sequence of operations by exploiting the precedences detected above. This operation is fully automated.

3. We bind events and operations to concrete BPEL activities: invoke, receive, reply, pick, assign, and loop/conditional blocks. Other actions [10] (e.g., activation of human tasks or Java snippets) will be considered in the future. The selection of a particular sequence of operations and its binding with a set of BPEL activities may require the human intervention.

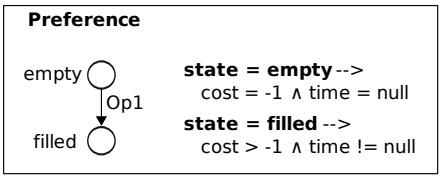

Figure 3: Step 1 applied to object Preference.

To exemplify these operations on our running example, we must start inspecting the domain pre- and post-conditions of each operation. This is to detect possible state changes on domain variables. The analysis of triggering pre-conditions, required pre-conditions and required post-conditions is then used to define the order among operations. For example, if a pre-condition of operation A is implied by the post-condition of operation B, we can say that B precedes A (B < A).

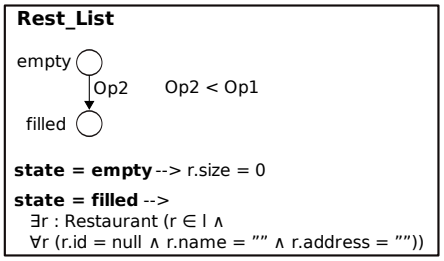

Figure 4: Step 1 applied to object Rest_List.

For example, an object *Preference* can only transit from state *empty* to state *filled* through operation *Op1*. Furthermore, when in state *empty*, its attributes *cost* and *time* are set to default values, since this is the initial state. In state *filled*, attribute *cost* must be greater than -1 and *time* must have a valid value (Figure 3).

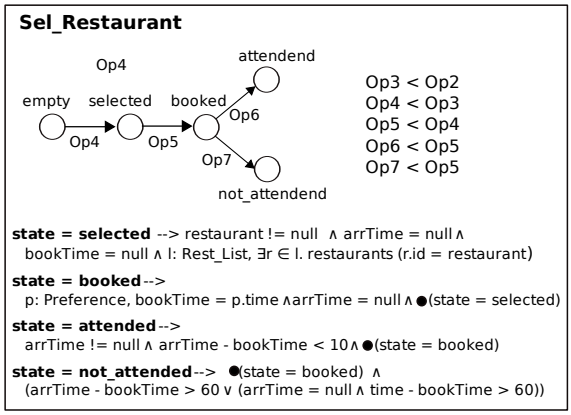

Figure 5: Step 2 applied to object Sel_Restaurant.

Object *Rest_List* can move from state *empty* to state *filled* through operation *Op2*. In state *empty*, attribute *restaurants* is not set to any value. While in state *filled*, it must be associated with at least one valid restaurant with a cost compatible with what set by the user (Figure 4). *Op2* also reads attribute *cost* of entity *Preference*, which must be set to a valid value. This implies that the object *Preference* must be in state *filled* before *Op2* can be executed. For this reason, we can also infer that *Op1* must be executed before *Op2*.

Entity *Sel_Restaurant* can move from state *empty* to state *selected* through operation *Op3*, from state *selected* to state *booked* through operation *Op5*, from state *booked* to state *attended* through operation *Op6*, and from state *booked* to state *not_attended* through operation *Op7* (Figure 5). *Sel_Restaurant* can be in state *selected* only if all its fields, but *bookTime*, are not set to the default value. It can be in state *booked* only if its previous state was *selected* and attribute *bookTime* has a value different from the default one.

All these considerations say that *Op2* must be executed before *Op3* since it is triggered by event *show_possibilities*, which is in the required post-conditions of *Op2*. Furthermore *Op4* can be performed only if entity *Sel_Restaurant* is in state *selected* (hence after *Op3* has been executed). Operation *Op4* must precede *Op5*. In fact *Op5* is triggered by event *get_confirmation*, which is among the required post-conditions of *Op5*. Finally operations *Op6* and *Op7* can only be executed if *Sel_Restaurant* is in state *booked*, that is, after operation *Op5*.

The aforementioned precedences between operations lead to the sequence of operations of Figure 6, which shows a possible operation flow and the generated events. Note this is an optimistic case; in more realistic situations, the result is only a partial order among operations, and we suppose that the user is in charge of selecting the actual order and thus of defining the actual flow.

The next step is the association of entities with process variables, while operations and events are translated into

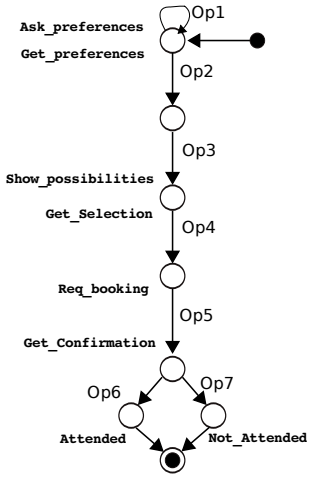

Figure 6: A possible flow.

concrete process activities. The first step is straightforward, while the second needs some clarifications:

1. If an operation only generates an event without modifying any entity, it can be associated with an invoke, invoke/receive, or a reply activity. This kind of operations cannot guide the selection of partner services since no effect is specified on the domain variables.
2. If an operation is not triggered by any event and modifies the state of some entities, it is associated with a set of assign activities. The parameters of these activities will depend on the variables' attributes that have to be modified. These assign activities can also be preceded by a set of invoke/reply activities followed by other assign activities when indicated by the user.
3. If an operation is triggered by an event, that event can be associated with a receive or pick activity. While if an operation is triggered by a condition on a set of entities, it is translated into an if block. If an operation is triggered by a condition on a set of entities and can be executed more than once in a row (i.e., it is a self-loop), it can be translated into a loop activity. If an operation is triggered by a condition evaluated at a given time, it is translated into a pick activity.
4. If an operation is triggered by an event and also modifies some entities' attributes, the receive and pick activities identified at step 1 are followed by a set of assign activities. Their output parameters must be assigned to the attributes of some process' variables. These operations guide the selection of the partner services and the selection process must ensure that the pre- and post-conditions stated in operations' definitions be satisfied.

Eventually, Figure 7 shows a consistent process for the Dinner Planner example. *Op1* is triggered by event *get_preference* and it causes a state change in object *Preference*. Hence it satisfies rules 2 and 4, and it is translated into a sequence of 2 activities: receive *get_preference* (associated with event *get_preference*) and assign *preference* (to set attributes *cost*, and *time* of variable *$p*). The parameters of the first activity can be used in the assignment.

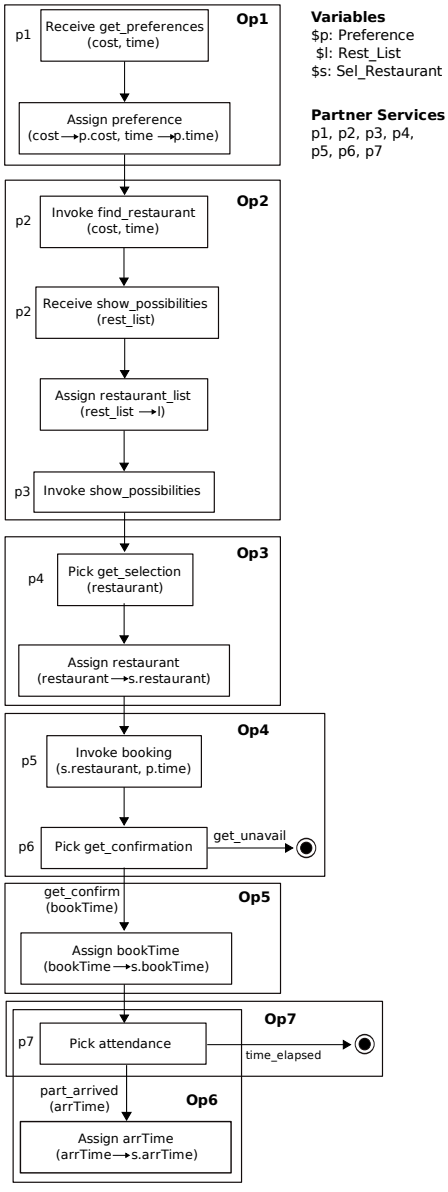

Figure 7: A possible process.

Op2 causes a state change in *Restaurant_List* and generates event *show_possibilities*. Hence it satisfies rules 1 and 2, and it is translated into the following activities: invoke *find restaurant*, receive *show possibilities*, assign *restaurant list* (to set attribute *restaurants* of variable *$l*), and invoke *show possibilities*.

The first activity uses partner service *p2* to get a list of restaurants that match user preferences. Its input parameters must contain variable *$p* since it is also among the input variables of *Op3*. The parameters of the second activity must include the list of valid restaurants to be assigned to variable *$l*. The contract specified for object *Rest_List* (Figure 4) is used for the selection of the partner service *p2*. Finally an invoke operation is associated with event *show_possibilities*.

Op3 is triggered by event *get_selection* and it causes a state change in object *Sel_Restaurant*. Hence it satisfies rule 2 and 4 and is translated into a sequence of two activities: pick *get_selection*, associated with event *get_selection*, and assign *restaurant* (to set attribute *restaurant* of variable *$s*). The parameters of the first activity must be enough to set attribute *restaurant* to a valid value.

Op4 generates event *req_booking* and event *get_confirm* after a certain time. It satisfies rule 1 and is translated into activities: invoke *booking* (associated with event *req_booking*) and pick *get_confirmation* (associated with event *get_confirm*). The input parameters of the first activity must allow one to set attribute *restaurant* of variable *$s* to a valid value.

Op5 is triggered by events *get_confirm* and *get_unavail*. It is translated into a pick activity that can process the messages associated with the previous events. In case of event *get_confirm*, the process performs activity assign *bookTime* to set attribute *bookTime* of variable *$s* to the output parameter of the corresponding pick activity. In case of event *get_unavail* the process terminates.

Op6 and *Op7* are mutually exclusive, and they both are triggered by an event (*part_arrived* and *time_elapsed*, respectively). These operations are translated into activity pick *get_attendance* to process the messages associated with the previous events. In case of *part_arrived*, we add the assign *arrTime* that sets attribute *arrTime* of variable *$s* to the output parameter of the corresponding pick activity.

5. SUPERVISION MODEL

Adaptive goals, together with their obstacles and additional conditions, guide the definition of the supervision directives that have to be embed in the process. These directives comprise both monitoring, to decide when adaptation must be carried out, and recovery, to change the running instance or the process itself. For each directive, we must collect data (from the process' state or external sources) needed to evaluate whether it must be applied. Note that the process' variables required to evaluate a goal are those associated with the entities used in the goal's definition. The supervision infrastructure provides probes to stop and resume the process execution when needed (for monitoring and adaptation purposes), monitoring components to analyze retrieved data, and adaptors to apply recovery actions.

5.1 Monitoring directives

In this paper, we instantiate the "general" approach by translating obstacles and additional conditions into the languages of two different supervision engines: ALBERT [3], which evaluates LTL properties, and Dynamo [9], which ex-

ploits punctual FOL (First Order Logic) properties. This was a convenience choice, since these engines were developed by one of the authors. However we provided an example and other monitors can be adopted in the same way. Given the "duration" of ALBERT properties, this engine is mainly used to evaluate goals asynchronously with respect to the process execution. In contrast, Dynamo, since its properties are punctual, is usually employed synchronously and the actual process execution does not continue before completing the evaluation of stated conditions.

The translation is defined by taking into account the following specification patterns for goals [17]:

1. $CurrCondition \Rightarrow \Diamond TargetCondition$
2. $CurrCondition \Rightarrow \Diamond_{\leq X} TargetCondition$
3. $CurrCondition \Rightarrow \circ TargetCondition$
4. $CurrCondition \Rightarrow GoodCondition$
5. $CurrCondition \Rightarrow \Box GoodCondition$
6. $CurrCondition \Rightarrow GoodCondition\ W\ NewCondition$[4]
7. $GoodCondition \Rightarrow \bullet Precondition$
8. $\Box_{\leq X}(CurrCondition \rightarrow GoodCondition)$
9. $CurrCondition \Rightarrow \Box_{\leq X} GoodCondition$
10. $CurrCondition \Rightarrow GoodCondition\ W_{\leq X} NewCondition$[5]

Expressions *CurrCondition*, *GoodCondition*, and *NewCondition* are LTL formulae. When they are used in the antecedent of an expression, they detect a particular process activity in which a goal must be evaluated.

5.1.1 Translation in ALBERT

The first three patterns are translated as follows:

1. $onEvent(CurrCondition) \rightarrow Until(TargetCondition, true)$
2. $onEvent(CurrCondition) \rightarrow Until(TargetCondition, elapsed(onEvent(CurrCondition)) \geq X)$
3. $(onEvent(\circ(CurrCondition)) \rightarrow TargetCondition)$

The pattern antecedent *CurrCondition* $\Rightarrow$ is translated into *onEvent(CurrCondition)*$\rightarrow$. The statement *onEvent(CurrCondition)* $\rightarrow$ *Expression* represents the fact that *Expression* must hold after the execution of activity *act*, detected by *CurrCondition*. In fact *onEvent(CurrCondition)* indicates the set of process activities after which *CurrCondition* is true. These activities must belong to the goal's scope (i.e., process activities that implement the goal's operations).

Expression $onEvent(\circ(CurrCondition))$ represents the end of the activity executed after that detected by $onEvent(CurrCondition)$. Expression $elapsed(onEvent(CurrCondition))$ detects the time instants elapsed since activity detected by *onEvent* $(CurrCondition)$ was executed.

The other patterns are translated as follows:

4. $onEvent(CurrCondition) \rightarrow GoodCondition$
5. $onEvent(CurrCondition) \rightarrow Until(GoodCondition, true)$
6. $onEvent(CurrCondition) \rightarrow Until(GoodCondition, false) \vee Until(GoodCondition, NewCondition)$
7. $onEvent(GoodCondition) \rightarrow past(Precondition, onEvent(\bullet(GoodCondition)), 1)$
8. $onEvent(CurrCondition) \rightarrow Until(GoodCondition, elapsed(onEvent(0)) \geq X)$
9. $onEvent(CurrCondition) \rightarrow Until(GoodCondition, elapsed(onEvent(CurrCondition)) \leq X)$
10. $onEvent(CurrCondition) \rightarrow (Until(Until(GoodCondition, NewCondition), elapsed(onEvent(CurrCondition)) \geq X) \vee Until(GoodCondition, elapsed(onEvent(CurrCondition)) \leq X))$

In this case expression $onEvent(\bullet(GoodCondition))$ represents the activity executed before that detected by $onEvent(GoodCondition)$. Expression $onEvent(0)$ represents the starting activity of the process.

For example goal $G1.3$ follows pattern 2 and is specified as follows:

$onEvent(invoke\ show\_possibilities) \rightarrow$
$Until(\$s/bookTime = \$p/time,$
$elapsed(onEvent(invokeBooking)) \leq X)$

The antecedent of goal $G1.3$ detects activity invoke *show_possibilities*, after which the goal can be evaluated. The satisfaction of goal G1.3 can be assessed after the last activity in its scope, assign *bookTime*.

Goal $G1.2.1$ follows pattern 4 and is translated as follows:

$onEvent(invoke\ show\_possibilities) \rightarrow$
$(\forall restaurant\ in\ \$l.restaurants;$
$\$restaurant/price - \$p/price \leq 15)$

The antecedent of goal $G1.2.1$ detects activity invoke *show possibilities*, after which the goal can be evaluated. Its satisfaction is punctual and can be assessed for the first state in which the monitoring rule is evaluated.

Goal $G1.4$ follows pattern 1 and is translated as follows:

$onEvent(Assign\ bookTime) \rightarrow$
$Until(\$s/arrTime \neq null \wedge$
$\$s/arrTime$ - $\$s/bookTime, true) \leq 10)$

5.1.2 Translation in Dynamo

Dynamo monitoring rules can be applied before and after process activities. An activity is identified by the rule's attribute location; while attribute isPrecondition signals whether the rule must be evaluated before or after the activity a rule is associated with. Dynamo is only able to monitor goals specified through Pattern 3, 4, 5, and 7.
Pattern 3 can be translated into the following rule:

location: next[activity(CurrCondition)]
isPrecondition: false
MonitoringRule: TargetCondition

where we assume that *TargetCondition* is a FOL expression. Furthermore the rule is applied after the activity that follows the one detected by *CurrCondition*. For example, if *CurrCondition* detects activity pick *get selection*, the rule expressed by *TargetCondition* is applied after activity assign *restaurant*.
Pattern 4 is translated as follows:

location: activity(CurrCondition)

[4] $GoodCondition\ W\ NewCondition \equiv (GoodCondition\ U\ NewCondition) \vee \Box(GoodCondition)$
[5] $P\ W_{\leq X}\ Q \equiv PU_{\leq d}Q \vee \Box_{\leq d}P$

isPrecondition: false
MonitoringRule: GoodCondition

and again *GoodCondition* is a FOL expression. It is applied after the activity detected by *CurrCondition*.
Pattern 5 is translated as follows:

location: next*[activity(CurCondition)]
isPrecondition: false
MonitoringRule: GoodCondition

and is applied after the activity detected by *CurrCondition* and after all its subsequent activities.
Pattern 7 is translated in two rules:

location: prev[activity(CurrCondition)]
isPrecondition: false
MonitoringRule: store(alias, data adopted in *Precondition*)

location: activity(CurrCondition)
isPrecondition: false
MonitoringRule: Precondition

The first rule stores the variables used in expression *Precondition* through the Dynamo primitive store. While the second rule evaluates *Precondition* on stored data.

For example Goal $G1.2.1$ matches pattern 2 and is translated as follows:

location: */start/.../Invoke show possibilities*
isPrecondition: false
MonitoringRule: $(forall\ \$r \in \$l.restaurants;$
$\$r.price\ \ \$p.price \leq 15; true)$

While Goal $G1.3$ can be translated as follows:

location: */start/.../Pick get selection*
isPrecondition: false
MonitoringRule: $\$Resp\_time \leq z$

The primitive *Resp_time* carries the time taken to perform activity pick *get selection*.

5.2 Recovery directives

The generation of recovery directives requires the inspection of the strategies associated with the adaptive goals. For each strategy, we must consider its scope, timeliness, and the other strategies it can have conflicts with.

Note that only punctual properties can be evaluated synchronously, and changes applied on both a single instance and all process' instances. If an obstacle and the additional conditions associated with a strategy need to be evaluated over a set of states (e.g. temporal properties), evaluation must be asynchronous and adaptation can only be applied on the instances that will be created afterwords.

Furthermore strategies can conflict as follows:

1. Conflicts among strategies that can be applied on the same goal at the same time due to the overlapping of their additional conditions;
2. Conflicts among strategies applied for the benefit of conflicting goals (where the conflict is specified in the goal model);
3. Conflicts among strategies that, if applied at the same time, may generate incoherent processes.

For conflicts of type 1, our policy is to allow the strategy with higher severity to be triggered before the others. If there is more than one strategy with the same severity, all of them can be triggered at the same time in case they do not generate a conflict of type 3. Another possibility can be to suggest the user to define these strategies in a hierarchical way, were a strategy is performed if the selected strategy does not achieve its objective.

Conflicts of type 2 are resolved through the goal model. Conflicting goals always have different priorities (the more critical ones have the higher priorities), and only the strategy of the goal with higher priority is triggered. Otherwise, if strategies have no conflicts of type 3, they all can be applied, with the hypothesis that the less critical goal has to adopt a strategy with severity level medium or low.

For example, if we consider goal $G1.2.1$, Table 1 summarizes the results of these considerations.

AG1.2.1			
Strategy	**Scope**	**Timeliness**	**Conflicts**
S1	instance	synchronous	S2
	process	synchronous asynchronous	
S2	instance	synchronous	S1

Table 1: Adaptation policy table for AG1.2.1.

Goal $G1.2.1$ is punctual, hence adaptation can be applied either synchronously or asynchronously. Since its additional conditions depend on the user's preferences, the scope is limited to the actual instance and adaptation can only be applied synchronously because we must know where to apply changes. The scope of strategy $S1$ is set to process since it addresses failures of a partner service, and thus the problem may be shared among different process instances. Strategy $S1$ and $S2$ may be in conflict since their additional conditions overlap. In this case $S1$ will be triggered, since it has a higher severity. The user can also specify that $S2$ substitutes $S1$, in case this fails in achieving its objective. The user can also choose to disable some strategies explicitly.

As said above, we also need to translate strategies into sets of basic actions that are to be applied on the original goal model, and its underlying process, to generate the adapted versions. For example, if we apply $S1$ of $AG1.2$ (see Figure 2), we must replace the portion of the goal model shown on the left-hand of Figure 8 with the goals shown on the right-hand of Figure 8. Action *remove goal(G1.2)* removes the goals, its subgoals, and its adaptive goals. Furthermore it also causes the removal of entity *Rest_List*, which is not needed anymore in the goal model, and of attribute *cost* from entity *Restaurant*. Action *add goal(G1.2')* causes the addition of operations *Op2'* and *Op3'*. It also causes adding attribute *type* to entity *Restaurant* and adding entity *Place_List(id, restaurants: Set<Restaurant>, state: {empty, filled})* to the goal model.

After performing this step, the modified goal model is transformed into a new functional model as explained in Section 4. In this case, the activities in the scope of goal $G1.2$ are substituted with other ones (see Figure 9). Furthermore, it also changes the contract of the adopted partner service $p2'$ since now has to find takeaways. The differences between the previous functional model and the new one can be represented through basic actions, as those presented by Casati et al. [4]. The authors propose two kinds of changes: on the process schema (add/remove activities, add/remove

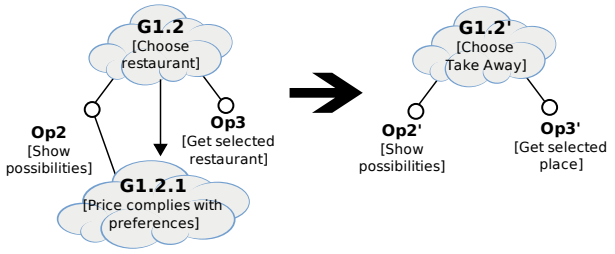

Figure 8: Modified goal model after applying S1.

variables, and add/remove partner services) and modifications on the process state (change variable values, initiate the rollback of a process region or of the entire process, terminate the process, and reassign an activity to a different partner service).

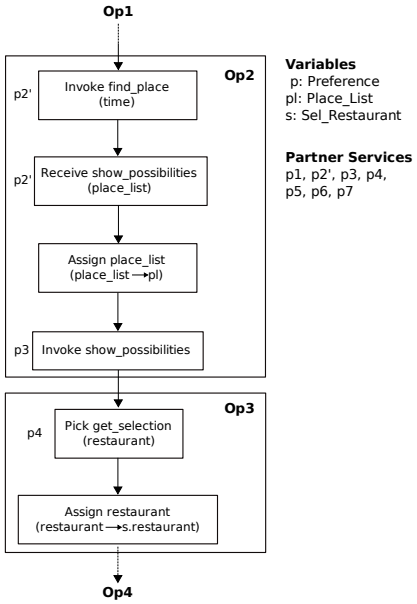

Figure 9: Modified functional model after applying S1.

For example, the modifications applied by $S1$ on the process definition are: the removal of the activities operations *Op2* and *Op3* rely on; and the addition of the activities operations *Op2'* and *Op3'* rely on. Partner service $p2$ is substituted by $p2'$. Variable *$l* is removed and a new variable is introduced (*$pl*).

To apply these changes, we must specify the process' execution points compliant with behavioral consistency rules [4][6]. For example:

- Each activity that is active when the old process instance is suspended must be present in the new one.
- A variable present both in the new and old process definition must keep the same type.

Furthermore migration consistency rules [4] have to be verified when a new execution state is built for the modified process instance. For example:

- Variables in the new process definition that are not present in the old one are initialized with their default value.
- Activities present both the new and the old process definition are initialized with the same execution state they had in the old process instance (e.g., not started, active, completed, failed, cancelled, etc.). While activities in the old process definition that are not present in the new one are set to the not started state.

For our example the safe points are all the execution points before operation *Op2'*. If a process instance has already passed the execution points in which a violation can happen (activities in the scope of *Op1*), modifications are not performed. Otherwise, at runtime the system stops the process instances when they reach the first safe point and applies modifications accordingly.

6. RELATED WORK

Our proposal aims at providing a goal-based methodology to model the requirements of service compositions. Cheng et al. [6] proposed a similar approach for self-adaptive systems in general. The authors detect the reasons (threats) that may cause uncertainty in the satisfaction of goals, and propose 3 kinds of strategy for their mitigation: add new functionality, tolerate uncertainty, or switch to a new goal model that have to repair the violation. Instead our strategies do not constraint the ways a goal model can be modified but can have different objectives and severity. These features allow us to solve conflicts among strategies and provide ways to apply them at runtime.

Different works have already tried to link service compositions with the business objectives they have to achieve. For example, Kazhamiakin et al. [11] adopt Tropos to specify the objectives of the different actors involved in a choreography. Tropos tasks are refined into message 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 annotated BPEL processes. Our approach, instead, is based on KAOS and focuses on software agents [19]. Moreover, while Kazhamiakin's processes can be verified statically through model checking, ours also embed self-adaptation capabilities.

Another similar approach is the one proposed by Mahfouz et al. [12], which models the goals of each actor and also the dependences among them. Actor dependencies take place when a task performed by an actor depends on another task performed by a different actor. Dependencies are then translated into message sequences exchanged between actors, and objectives into a set of local activities performed in each actor's domain. The authors also propose a methodology to modify a choreography according to changes in the business needs (dependencies between actors and local objectives). Although this approach traces changes at requirements level in the underlining composition, it does not provide explicit policies to apply these changes at runtime.

The idea of monitoring requirements was originally proposed by Fickas et al. [8]. The authors adopt a manual approach to derive monitors able to verify requirements' satisfaction at runtime. Wang et al. [18] use the generation of log

[6]They assure that neither runtime errors or non-deterministic behaviors may take place.

data to infer the denial of requirements and detect problematic components. Diagnosis is inferred automatically after stating explicitly what requirements can fail. Robinson [15] 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 that diagnostic formulae be generated manually from obstacle analysis.

Despite a lot of work focused on monitoring requirements, only few of them provide reconciliation mechanisms when requirements are violated. Wang et al. [18] generate system reconfigurations guided by OR-refinements of goals. They choose the configuration that contributes most positively to the non-functional requirements of the system and also has the lowest impact on the current configuration.

To ensure the continuous satisfaction of requirements, one needs to adapt the specification of the system-to-be according to changes in the context. This idea was originally proposed by Salifu et al. [16] and was extensively exploited in different works [14, 2] that handled context variability through the explicit modeling of alternatives. Penserini et al. [14] model the availability of execution plans to achieve a goal (called ability), and the set of pre-conditions and context-conditions that can trigger those plans (called opportunities). Dalpiaz et al. [2] explicitly detect the parameters coming from the external environment (context) that stimulate the need for changing the system's behavior. These changes are represented in terms of alternative execution plans. Moreover the authors also provide precise mechanisms to monitor the context. All these works are interesting for their capability of addressing adaptation at requirements level, but they mainly target context-aware applications and adaptation. They do not consider adaptations that may be required by the system itself because some goals cannot be satisfied anymore. We also foresee a wider set of adaptation strategies and provide smarter mechanisms to solve conflicts among different strategies.

7. CONCLUSIONS

The paper presents a goal-based approach for dealing with self-adaptive BPEL processes from requirements elicitation down to execution. It proposes the concept of *adaptive goal* and demonstrates how to address adaptation and changes at goal level. Adaptive goals identify countermeasures offered to the different instances of the system-to-be to cope with problems and changes. Conventional goals govern the semi-automatic definition of the process, while adaptive goals are used to define the supervision model, that is, how the system can adapt itself in case of anomalies. The selection of the actual strategy depends on the execution context (experienced violations and execution state) of each instance of the BPEL process. In the future, we would like to have an explicit notion of satisfaction level for requirements. This would enable more sophisticated adaptation strategies linked to these levels rather than to boolean values (i.e., to the satisfaction or insatisfaction of requirements). We also plan to augment our methodology with suitable design tools and use it to model more complex and realistic service compositions.

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).

8. REFERENCES

[1] Web Services Business Process Execution Language Version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[2] R. Ali, F. Dalpiaz, and P. Giorgini. A Goal Modeling Framework for Self-Contextualizable Software. In *Proc. of the 10th Int. Work. on Enterprise, Business-Process and Information Systems Modeling*, pages 326–338, 2009.

[3] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of Web Service Compositions. *IET Software*, pages 219–232, 2007.

[4] F. Casati, S. Ilnicki, L. jie Jin, V. Krishnamoorthy, and M.-C. Shan. Adaptive and Dynamic Service Composition in *Flow*. In *Proc. of the 12th Int. Conf. of Advanced Information Systems Engineering*, pages 13–31, 2000.

[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 Engineering Service-Oriented Applications*, pages 363–375, 2007.

[6] B. H. C. Cheng, P. Sawyer, N. Bencomo, and J. Whittle. A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. In *Proc. of the 12th Int. Conf. on Model Driven Engineering Languages and Systems*, pages 468–483, 2009.

[7] M. Colombo, E. D. Nitto, and M. Mauri. SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules. In *Proc of the 4th Int. Conf. of Service Oriented Computing*, pages 191–202, 2006.

[8] S. Fickas and M. S. Feather. Requirements Monitoring in Dynamic Environments. In *Proc. of the 2nd Int. Symp. on Requirements Engineering*, pages 140–147, 1995.

[9] S. Guinea. *Dynamo: A Framework for the Supervision of Web Service Compositions*. PhD thesis, Politecnico di Milano, 2007.

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

[11] R. Kazhamiakin, M. Pistore, and M. Roveri. A Framework for Integrating Business Processes and Business Requirements. In *Proc. of the 8th Int. Conf. on Enterprise Distrib. Object Computing*, pages 9–20.

[12] A. Mahfouz, L. Barroca, R. C. Laney, and B. Nuseibeh. Requirements-Driven Collaborative Choreography Customization. In *Proc. of the 7th Int. Joint Conf. ICSOC-ServiceWave*, pages 144–158, 2009.

[13] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive Monitoring and Service Adaptation for WS-BPEL. In *Proc. of the 17th Int. Conf. on World Wide Web*, pages 815–824, 2008.

[14] L. Penserini, A. Perini, A. Susi, and J. Mylopoulos.

High Variability Design for Software Agents: Extending Tropos. *ACM Trans. Autonomous Adaptive Systems*, 2(4):75–102, 2007.

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

[16] M. Salifu, Y. Yu, and B. Nuseibeh. Specifying Monitoring and Switching Problems in Context. In *Proc. of the 15th Int. Conf. on Requirements Engineering*, pages 211–220, 2007.

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

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

[19] E. S. K. Yu. Models for Supporting the Redesign of Organizational Work. In *Proc. of the 5th Int. Conf. on Organizational Computing Systems*, pages 226–236, 1995.