Towards Adaptive Systems through Requirements@Runtime[⋆]

Liliana Pasquale[1], Luciano Baresi[2], Bashar Nuseibeh[1,3]

[1] Lero - Irish Software Engineering Research Centre, Ireland
{liliana.pasquale|bashar.nuseibeh}@lero.ie
[2] Politecnico di Milano, Italy
baresi@elet.polimi.it
[3] The Open University, Milton Keynes, United Kingdom
b.nuseibeh@open.ac.uk

Abstract. Software systems must adapt their behavior in response to changes in the environment or in the requirements they are supposed to meet. Despite adaptation capabilities could be modeled with great detail at design time, anticipating all possible adaptations is not always feasible. To address this problem the requirements model of the system, which also includes the adaptation capabilities, is conceived as a runtime entity. This way, it is possible to trace requirements/adaptation changes and propagate them onto the application instances. This paper leverages the FLAGS [1] methodology, which provides a goal model to represent adaptations and a runtime infrastructure to manage requirements@runtime. First, this paper explains how the FLAGS infrastructure can support requirements@runtime, by managing the interplay between the requirements and the executing applications. Finally, it describes how this infrastructure can be used to adapt the system, and, consequently, support the evolution of requirements.

1 Introduction

Software systems must be able to adapt to continue to achieve their requirements while they are executing. The need for adaptation may be triggered by different events: the application reaches a particular execution state, the context changes, established requirements are not satisfied, or the objectives of the system change. Some of these events can be foreseen in advance, and thus the corresponding adaptation capabilities, *foreseen adaptations*, can be planned and designed carefully, while others cannot and adaptation capabilities, *unforeseen adaptations*, must be devised completely at runtime. Note that, besides predictability, there is a tradeoff between the number of different events the system is able to react to (completeness) and the cost of embedding these reactions from the beginning. Furthermore, even if all possible adaptations could be anticipated

⋆ This research has been funded by Science Foundation Ireland grant 03/CE2/I303_1 and the European Commission, Programmes: IDEAS-ERC, Project 227977 SMScom, and FP7 Network of Excellence 215483 S-Cube.

from the beginning, some of them may become useless when requirements evolve, due to new business objectives or users' needs.

In the last years, different modeling notations [2,3,1] have been proposed to engineering adaptations at the requirements level. RELAX [2] is a notation to express uncertain requirements, whose assessment is affected by the imprecision of measurements. Adaptation can be designed by relaxing non-critical requirements for guaranteeing that the critical ones are still satisfied. Awareness requirements [3] are used to represent the requirements of the activities in the feedback loop. They may trigger a set of changes in the requirements model and must be aware of the satisfaction of the other requirements of the system. However, these research contributions are mainly focused on modeling foreseen adaptations, and neglect requirements evolution. In our previous work we propose the FLAGS [1] methodology. Similarly to the other approaches, it allows the designer to elicit adaptation together with the other conventional (functional and non functional) requirements of the system. Adaptation capabilities are represented as adaptation goals that are added to the KAOS [4] goal model. However, FLAGS also conceives requirements as runtime entities [5], and provides a suitable infrastructure to dynamically support their evolution.

This paper clarifies how FLAGS supports requirements@runtime. In particular, the goal model is conceived as a runtime entity and is fed by the data coming from the running application instances. The goal model can dynamically change to accommodate new/changing requirements. We assume that applications are expressed as BPEL [6] processes. Adaptation actions modeled at requirements level can affect the underlying architecture/execution environment (e.g., by restructuring the process activities or changing some partner services). They can also modify the process definition and the goal model, thus changing the actual requirements of the system. Finally the paper exemplifies how the FLAGS methodology supports the requirements evolution. In particular, every time the requirements change, their modifications must be propagated onto the process and adaptations must evolve accordingly. Note that we do not focus on the automatic identification of new requirements/adaptations capabilities, and assume that they are manually added by the designer.

The paper is structured as follows. Section 2 describes how the FLAGS model can be used to represent adaptations. Section 3 illustrates the runtime infrastructure to support requirements@runtime. Section 4 describes how the infrastructure activates adaptations at runtime and supports the interplay between the requirements and the running application. Section 5 discusses some related approaches and concludes the paper.

2 Modeling Adaptations with FLAGS

This section classifies different kinds of adaptation, and demonstrates how they can be elicited through FLAGS. We use an example of a web portal, *Click&Eat*, which collects information regarding different restaurants that deliver food at home and allows customers to order food from one of them.

2.1 Adaptation Classification

Adaptations can be classified along several dimensions [7]. For the objectives of this paper we just consider their *effect* and *anticipation*.

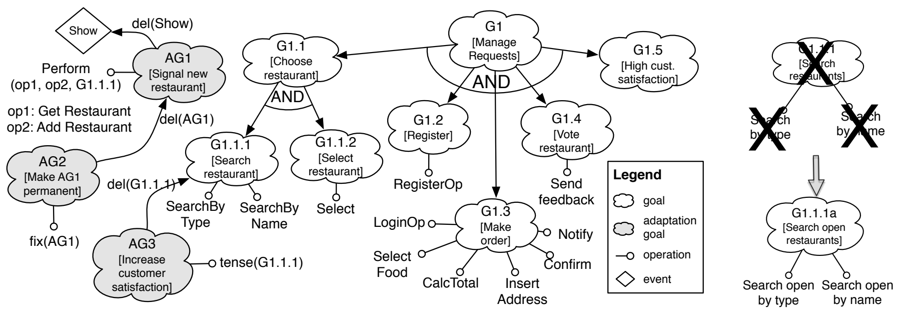
- **Effect.** Adaptations may just change the underlying implementation or may also modify the actual requirements of the system. In the first case, they can be performed automatically and may affect one or all application instances.
  - **Single Instance.** These adaptations are suitable to cope with transient events and can modify the execution flow of a process instance or one of its partner links. Hence, the effects of these adaptations are temporary and do not have an impact on the process definition.
  - **All Instances.** They restructure the process by changing the definition of all (executing and future) process instances. Hence, the effects of these adaptations are permanent.

  Despite it can be feasible to apply temporary actions to all process instances or permanent actions on a single instance, we did not find useful to introduce this further distinction for our purposes.
  In case some requirements cannot be satisfied, due to, for example, lack of resources or premature design choices, or to accommodate context changes, adaptation actions modify the requirements model —including the adaptation capabilities— and propagate their effect on all application instances.
- **Anticipation.** Adaptations can be planned ahead of time or not. Foreseen adaptations are modeled by the designer along with the other conventional requirements of the system. This way the underlying process can adapt autonomously when a specific scenario takes place (e.g., a goal is violated, a specific event happens). Unforeseen adaptations are determined by changes in the requirements that may take place after an application is already on the market, due to new users needs, or mutations in the organization, laws and business opportunities. Requirements changes may also be due to the execution of an adaptation that modifies the requirements model. In all these cases, new adaptations must be identified, or some of the existing adaptations may no longer be useful or may need to be modified. Despite requirements changes cannot be identified automatically, the way new requirements impact on the underlying application can be detected semi-automatically through requirements traceability.

2.2 Adaptation Elicitation

Adaptation goals must be specified over the conventional requirements of the system. Figure 1(a) shows a FLAGS model of *Click&Eat*. The general objective of the system is to manage the customers' requests (goal G1). Customers want to browse available restaurants to view offered food items and their cost (goal G1.1). To this aim they must search a set of potential restaurants (goal G1.1.1) and select one among them (goal G1.1.2). The search can be performed by name (operation *SearchByName*) or by kind of provided food (operation *SearchByType*). All customers must register (goal G1.2) to be able to make an order (goal

G1.3). This application also aims to collect the customers' feedbacks regarding a selected restaurants (goal G1.4). Finally the average satisfaction of all customers must be high (goal G1.5).



(a) (b)

Fig. 1. The FLAGS model (a) and its modifications (b) for *Click&Eat* web portal.

Adaptation goals define the adaptation capabilities embedded in the system at requirements level. The operationalization of these goals defines the actions to be carried out when adaptation is required. Each adaptation goal is associated with a trigger and a set of conditions. The trigger states when the adaptation goal must be activated. Conditions specify further necessary constraints that must be satisfied to allow the corresponding goal to be executed. Conditions may refer to properties of the system (e.g., satisfaction levels of goals, or adaptation goals already performed) or domain assumptions.

Adaptation goals can embed process-level actions that simply change the way goals are achieved (process activities, partner services), or goal-level actions that modify the goal model. Process level adaptations have conditions that do not depend on the satisfaction of goals, but are just expressed through untimed formulas over runtime data. Furthermore, they can be performed any number of times, while goal level adaptations must be performed at most once. Process-level actions are: *perform([*o_1*,..., *o_n*], g/o)*, may optionally execute a sequence of operations (o_1, ..., o_n) and resumes the execution before operation o or goal g is executed; *substitute(*a_1*, *a_2* [, o])*, substitutes agent a_1 with a_2 for all operations performed by a_1 (in case the third parameter is not specified) or only when a_1 executes operation o (otherwise); *fix(ag)*, includes the effects of an adaptation goal (ag) in the process definition[4]. Goal level actions are: *add/remove(g)*, adds or removes (conventional/adaptation) goal g; *add(o, g)*, adds operation o to goal g; *remove(o)*, removes operation o; *add/remove(e/ev/a)*, adds or remove an entity (e), or an event (ev), or an agent (a); *relax(g, constr)*, modifies the definition

[4] Action $fix(ag)$ can be applied in case ag temporarily modifies the process execution flow, without changing the process definition or the goal model.

of a leaf goal (g) making it less strict (e.g., by adding a new disjunct constraint or relaxing the corresponding membership function); *tense(g, constr)*, modifies the definition of a leaf goal (g) by making it stricter (e.g., by adding a new conjunct constraint or tensing the corresponding membership function). Adaptation goals are associated with other elements of the goal model (dependencies). In case at least one element in the dependencies is removed or modified, the adaptation goal must be consequently removed. This way adaptations can be automatically adjusted to comply with the modifications of the goal model.

To handle those cases in which a desired restaurant cannot be found, a temporary adaptation goal (AG1) is defined. It aids goal G1.1.1, is triggered when event *Show* takes place (i.e., after operation *SearchByName*), under the condition that attribute *list* of event *Show* does not contain any restaurant. As actions, AG1 performs operations *Get Restaurant* and *Add Restaurant*, which respectively retrieve the information of a restaurant from a user and add it to the list of available restaurants. This adaptation depends on event *Show*: in case this event is removed from the the model, AG1 must be consequently removed. In case AG1 is triggered too many times, a permanent adaptation goal (AG2) is applied. It performs action *fix(AG1)*, which removes AG1 and adds the actions performed by AG1 to the process definition. It is triggered when AG1 is performed and under the condition that AG1 has been triggered more than 10 times. When some goals (G1.5) are not satisfied enough, due to wrong design choices, the requirements of the system should change. For example, adaptation goal AG3 may be applied to tense goal G1.1.1 by adding a conjuntive constraint. This new constraint asserts that only those restaurants with a feedback greater than 40% must be shown to the customer.

The customers may change their requirements while the system is executing. For example, they may want to visualize only those restaurants that are open at the moment when the search is performed. To address these changes it is necessary to modify the goal model manually, by substituting goal G1.1.1 with G1.1.1a, as shown in Figure 1(b). Consequently, new unforeseen adaptations must be automatically applied at the applications level to propagate these modifications to all instances.

3 The Runtime Infrastructure

As shown in Figure 2, FLAGS provides a conceptual infrastructure to enact requirements at runtime. Its components operate respectively at the process and the goal level, whereas the *Process Reasoner* handles the interplay between them. To support requirements evolution the infrastructure manages two models at runtime: the *FLAGS model* and the *implementation model*. The former includes the requirements and the adaptation capabilities performed at the goal level, while the latter includes the definition of the process together with the data collection, monitoring and adaptation capabilities necessary to support the adaptations at the process level. These models are managed respectively by the components at the goal and process level.

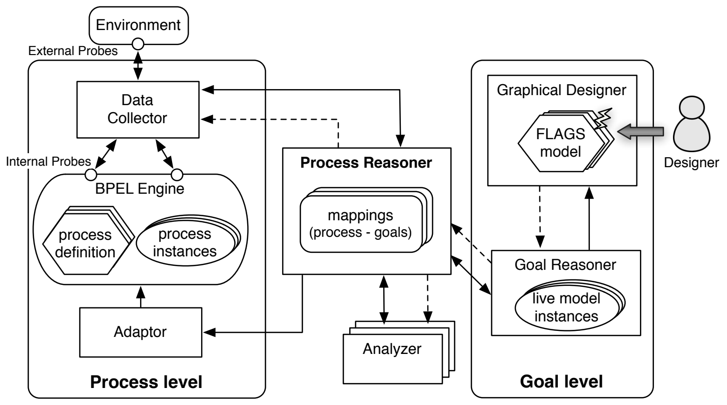


Fig. 2. Runtime infrastructure.

At the process level a *BPEL engine* supports the execution of the process instances. We use a modified version of ActiveBPEL [8] engine, which is augmented with Aspect Oriented Programming [9]. It provides a customizable "aspect" that can be used to intercept the process execution at specific points, for example to collect the process internal state or apply some adaptation actions. The *Data Collector* manages the collection of runtime data from the process or the environment. The process state is collected by the *Internal Probes* that are activated when the process reaches specific execution points. Conversely, the data from the surrounding environment are gathered by the *External Probes* that expose a proper interface to be configured and enacted at runtime.

Runtime data are sent to the *Goal Reasoner*, through the Process Reasoner, and are used to update the elements of the goal model (i.e., events, entities, satisfaction of leaf goals and performed adaptations). The Process Reasoner also evaluates the satisfaction of leaf goals, by invoking a specific *Analyzer*, depending on the kind of constraint (i.e., temporal or untimed) that must be checked. When the goal model changes, the Goal Reasoner notifies the Process Reasoner, which propagates these modifications on the running and next process instances. To manage the interplay between the goal model and the process, the Process Reasoner uses a bidirectional *mapping* between the elements of the FLAGS model, and those of the implementation model. Note that the adaptations that are applied at runtime may generate different versions of the implementation model and the FLAGS model. For this reason, the Process Reasoner must store different mappings, depending on the versions of the models that are in use.

The Process Reasoner orchestrates the adaptation at the process level, according to the directives of the implementation model. It monitors the runtime data to check if the trigger and conditions of an adaptation are satisfied. In this case, it activates proper adaptation actions through an *Adaptor*, which can temporarily change the execution flow of a single process instance or may change the implementation model, by modifying the definition of the process and the adaptation capabilities. To adapt the running process instances, the Adaptor intercepts the execution at a “safe” point, where the modifications can be correctly applied, since they do not break any conversation or transaction. To adapt the next process instances the Adaptor transparently deploys a new version of the process in the BPEL engine.

At the goal level, the *Graphical Designer* [10] allows the designer to create a new version fo the FLAGS model and modify it at runtime. It also stores all versions of the FLAGS model that are actually in use. The Goal Reasoner manages the live instances of the FLAGS model. It creates a new instance every time the Process Reasoner signals that a process instance is started. This live instance must conform to the last version of the FLAGS model. Every time a live instance of the FLAGS model is updated, a set of rules recompute the satisfaction of high level goals, evaluate the triggers and conditions of the adaptation goals, and decide which adaptation goal must be applied, if necessary. In our implementation the Goal Reasoner is based on JBoss rule engine [11].

Every time the last version of the FLAGS model is modified at runtime, a new version is created. Consequently, all live instances of the FLAGS model and their corresponding process instance must migrate to the new version of the model, if possible. A new version of the implementation model is inferred. In particular, the new version of the process is inferred and deployed on the BPEL Engine, the new mappings between the goal model and the process are generated and stored at the Process Reasoner, and the components at the process level must be properly re-configured to update and apply process level adaptations for the modified instances of the FLAGS model. A live instance of the FLAGS model is canceled every time the Process Reasoner signals that the corresponding process instance has terminated. In case there is no running instance of a process associated with an old version of the FLAGS model, that version is removed together with its corresponding mappings at the Process Reasoner.

4 Managing Adaptations @Runtime

This section describes how adaptations are applied at runtime and their impact on the implementation and the FLAGS model. The implementation model may change when process level adaptations are performed. The FLAGS model can change when the designer manually modifies it or when goal level adaptations are applied, and, in these cases, the implementation model must evolve accordingly.

4.1 Process Level Adaptations

Adaptations at the process level are applied on a single process instance and may perform a small set of actions. They can change the process execution flow (action $perform$) or substitute a partner service with another one (action *substitute*). These adaptations require to keep the process blocked at specific execution points, while their conditions are checked and their actions are applied, if necessary. The trigger of these adaptations must clearly identify the execution point where the adaptations are performed. Since conditions must be evaluated "on-the-fly", while the process is blocked, they must be expressed as untimed formulas. The Process Reasoner can verify them by invoking one of the Analyzers that have been plugged in the infrastructure.

To apply action *perform(*$[o_1, \ldots, o_n], g/o$*)*, the operations included in the first argument $(op_1, \ldots, op_n)$ are translated into a sequence of activities that must be temporarily performed. These activities can be executed, for example, by invoking an external partner service. The second argument is associated with a concrete execution point where the process execution must be restored. This action can be applied at runtime through a suitable Adaptor, such as Dynamo [12], which provides recovery actions call(wsdl, operation, ins) and restore(destLocation). The former calls an operation exposed by an external web service, which is identified by its wsdl. The third parameter (ins) represents the data that are to be sent to the service. Action restore takes the process back to the point of execution immediately prior to the destLocation (indicated with an XPath expression), and resumes the process execution from there.

To apply action *substitute(*a_1*,* a_2 *[, o])*, the BPEL partner links p_1, p_2 that are associated with agents a_1, a_2 must be identified. This action substitutes p_1 with p_2 for all process activities in which p_1 is used, in case the third parameter is not specified. Otherwise, it is necessary to identify the process activities associated with operation o, and substitue p_1 with p_2 for all of them. Both adaptation actions can be supported by Dynamo. In the first case, we can use recovery action rebindPartnerLink(name, wsdl), which changes partner link p_1, identified by a name, with p_2, identified by its wsdl. In the second case we can apply action rebind(wsdl, operation), which substitutes partner link p_1, which performs activity operation, with p_2, identified by its wsdl.

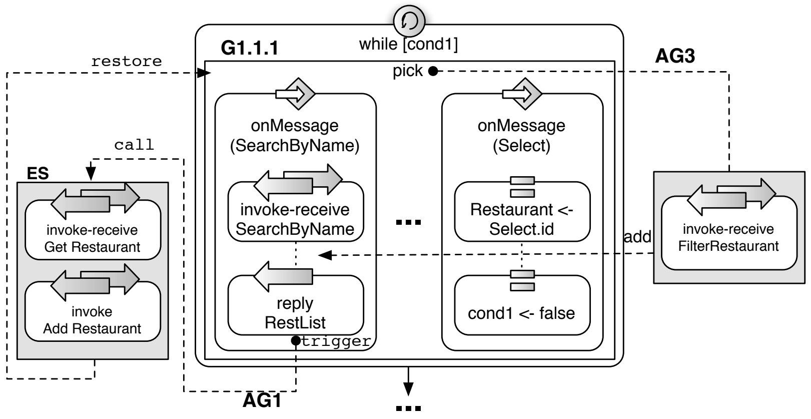


Fig. 3. BPEL process for the Click&Eat application.

For example, AG1 is an adaptation goal that applies action $perform$(*GetRestaurant*, *AddRestaurant*, G1.1.1) on a single process instance. To support its enactment at runtime, the Internal Probes must be instrumented to intercept the process execution at the points when event *Show* (trigger) takes place. In our example (see Figure 3) this corresponds to the execution point after activity reply *RestList*. This way, every time the process terminates this activity, its execution is blocked, and, in case the condition is satisfied, an external service (ES) is invoked. It performs a sequence of activities associated with the operations *GetRestaurant* and *Add Restaurant* and restores the process execution before activity pick, which is the execution point where G1.1.1 starts to be activated.

All adaptations that are applied on all process instances have a permanent effect on the implementation model. The trigger of these adaptations only identifies the moment when the condition can be verified. A set of "safe points" must be identify the process point at which an adaptation must be applied. In case a running process instance has already passed all safe points, it cannot be migrated. All the other adaptations at the process level are temporarily deactivated, until all instances complete the migration to the next version of the process, when possible. The new mappings between the new version of the process and the last version of the FLAGS model must be also added to the Process Reasoner. Finally the adaptation capabilities at the Data Collector and Process Reasoner may also change.

AG2 falls in this category, since it applies action *fix(AG1)* to modify the process definition. After the Process Reasoner performs AG1 (trigger) for one of its process instances, the condition associated with AG2 must be verified. AG2 applies a set of modifications at the Process Reasoner, since it removes AG1 and all the directives necessary to support its execution. AG2 also modifies the process definition by inserting a set of activities after the execution point identified by the trigger of AG1 (i.e., after activity reply *RestList*). These activities are: an if activity, which verifies the condition of AG1, and the activities that were temporarily performed by external service *ES*. A new version of the process that includes all these modifications is deployed as well in the BPEL Engine.

4.2 Goal Level Adaptations

For goal level adaptations things work slightly differently. Their trigger and conditions are checked by the Goal Reasoner every time a new element of the goal model is updated. When an adaptation at the goal level is applied, the Goal Reasoner suspends all further updates of the goal model and stores all runtime data received from the Process Reasoner in a buffer, until the adaptation completes. It also sends a notification to the Process Reasoner to block all running process instances at the end of the activity they are currently executing. An adaptation at the goal level generates a new version of the FLAGS model, and must apply the corresponding modifications onto its live instances. Note that only those instances that comply with the last version of the FLAGS model can potentially migrate to the new version. The migration can take place only if the corresponding process instance has not already passed the safe points where modifications must be applied.

A new version of the implementation model is created accordingly. In particular, the Process Reasoner creates a new version of the mappings between the process and the goal model and indicates what are the process instances that must actually follow this mapping. Consequently the probes and the Process Reasoner change their configuration to respectively intercept the process at different points, collect different data, evaluate different leaf goals (specified with different constraints) and activate different adaptation actions. Finally the definition of all running and next process instances must change to comply with the modifications of the goal model. For example, if some goals or operations are added/removed, the corresponding activities in the process definition must be added/removed. In case agents are added/removed, the corresponding partner links must be added/removed from/to the process definition and so on.

After all aforementioned actions are performed, the execution of all process instances can be restored. All runtime data received by the Process Reasoner that are not associated with any element at the goal level are automatically removed. The Goal Reasoner will start to process the data stored in its buffer only after all process instances correctly migrated to their new version (if possible). The Goal Reasoner will discard all the data that are associated with leaf goals, entities and events that do not exist anymore or have been modified. Goal level adaptations can be triggered while a process level adaptation, which modifies the process definition, is being performed on the process. In this case, it is necessary to guarantee that only the previous adaptation is applied, or both of them are applied. In other cases a goal level adaptation can be triggered while one or more process instances are performing a process level adaptation that just temporarily modifies the process execution flow. In this case the process instance must terminate the execution of the temporary activities and must be blocked before performing the activity associated with the restore point.

In our example, AG3 applies a goal level adaptation when the satisfaction of G1.5 is low (trigger). It changes the definition of goal G1.1.1 and its operationalization. As an effect, the Goal Reasoner creates a new version of the FLAGS model, manage the migration for the live instances of the model, and adds a new version of the mappings to the Process Reasoner. At the process level the next process instances can be migrated by deploying a new version of the BPEL process, which complies with the new version of the FLAGS model. The running process instances can be migrated by intercepting their execution before activity pick and adding activity invoke-receive *FilterRestaurant* after invoke-receive *SearchByName*, as shown in Figure 3. A similar procedure is followed when a new version of the FLAGS model is manually created by designer, who directly modifies the last version of the FLAGS model. This can happen when, for example, the requirements of the system change, as shown in Figure 1(b). In this case, the Graphical Designer notifies the Goal Reasoner that propagates the modifications on the live instances of the FLAGS model and on the process. A goal model at runtime allows us to reshape the adaptations at the process level, depending on the modifications applied on the process. In this example all the adaptation goals that have been defined can persist, since goal G1.1.1 and event *Show* are not removed. However the way adaptations are supported at runtime changes. For example, at the process level AG3 will be added after another activity (invoke-receive *SearchOpenByName*), and will modify the new definition of G1.1.1 by adding its conjunct constraint to its modified consequent.

5 Discussion

Different solutions have been already proposed to support requirements evolution. Courbis and Finkelstein [13] analyze the requirements in a number of possible environments where the system can execute. This analysis is used to identify alternative requirements definition and architectural choices to foster "design for change". Note that identifying all possible changes in advance is not always possible. For this reason requirements@runtime [5] are fundamental to trace the modifications of both the requirements and the operative environment where the system is executing. Ali et al. [14] use requirements at runtime to support their evolution when wrong assumptions are discovered. Requirements evolution can be performed automatically, by changing the priority given to software variants, or can be manually performed by the designer. However this work neglects how requirements changes are applied onto the system.

To support requirements at runtime the link between the requirements and the underlying implementation cannot be lost. The changes in the system requirements at runtime may trigger the execution of a set of analysis (e.g., consistency check, requirements verification) that have been traditionally performed offline. Baresi and Ghezzi [15] foster this idea. In particular, they claim that the rigid boundary between development time and runtime must be broken and more support must be provided to analyze and re-design the software at runtime. According with this idea, Epifani et al. [16] use a live model of the system to reason about its reliability. They use runtime data to feed the model and perform probabilistic analysis to improve its accuracy. The updated model can be used to detect or predict if a reliability property will be violated in the running implementation. Our work also maintains the link between the requirements model and the implementation model and apply unforeseen requirements and application features through aspects, as already proposed by Courbis and Finkelstein [17]. A similar idea has been also proposed in the DiVA project [18], which models variability dimensions as aspects and performs dynamic aspect weaving as model transformations when an adaptation need occurs.

Our approach needs further improvements. It lacks a mechanism to verify the consistency and correctness of the models updated by the designer. It should better handle conflicts between adaptations that are activated at the same time. We are also planning to exploit requirements@runtime in the security domain. Security is a critical property whose violation must be avoided. The countermeasure used to support this property depend on the context and, for this reason, it may be fundamental to detect changes that can take place in the assets to be protected or in the environment. These changes may trigger suitable analysis at runtime and activate new countermeasures necessary to continue to guarantee the satisfaction of security requirements.

References

1. Baresi, L., Pasquale, L., Spoletini, P.: Fuzzy Goals for Requirements-Driven Adaptation. In: Proc. of the 18th Int. Requirements Eng. Conf. (2010) 125–134
2. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C.: RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In: Proc. of the 17th Int. Requirements Eng. Conf. (2009) 79–88
3. Silva Souza, V.E., Lapouchnian, A., Robinson, W.N., Mylopoulos, J.: Awareness Requirements for Adaptive Systems. In: Proc. of the 6th Int. Symposium on Software Eng. for Adaptive and Self-Managing Systems. (2011) 60–69
4. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. John Wiley (2009)
5. Sawyer, P., Bencomo, N., Whittle, J., Letier, E., Finkelstein, A.: Requirements-Aware Systems: A Research Agenda for RE for Self-adaptive Systems. In: Proc. of the 18th Int. Requirements Eng. Conf. (2010) 95–103
6. OASIS WSBPEL TC: Web services business process execution language. `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`
7. B. Cheng et al.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Software Engineering for Self-Adaptive Systems. Volume 5525. Springer Berlin / Heidelberg (2009) 1–26
8. Active Endpoints: The activebpel engine. `http://www.activevos.com/community-open-source.php`
9. G. Kiczales et al.: Aspect-Oriented Programming. In: Proc. of the 11th European Conference on Object-Oriented Programming. (1997) 220–242
10. Baresi, L., Pasquale, L.: An Eclipse Plug-in to Model System Requirements and Adaptation Capabilities. In: Proc. of the 6th IT-Eclipse Workshop. (2011) (to appear)
11. JBoss Drools Team: Drools expert. `http://jboss.org/drools`
12. Baresi, L., Guinea, S.: Self-Supervising BPEL Processes. IEEE Trans. on Software Eng. **37**(2) (2011) 247–263
13. Bush, D., Finkelstein, A.: Requirements Stability Assessment Using Scenarios. In: Proc. of the 11th Int. Conf. on Requirements Eng. (2003) 23–32
14. Ali, R., Dalpiaz, F., Giorgini, P., Souza, V.E.S.: Requirements Evolution: From Assumptions to Reality. In: Proc. of the 12th Int. Conf. BMMDS/EMMSAD. (2011) 372–382
15. Baresi, L., Ghezzi, C.: The Disappearing Boundary Between Development-time and Run-time. In: Proc. of the Workshop on Future of Software Eng. Research. (2010) 17–22
16. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: Proc. of the 31st Int. Conf. on Software Eng. (2009) 111–121
17. Courbis, C., Finkelstein, A.: Towards Aspect Weaving Applications. In: Proc. of the 27th Int. Conf. on Software Eng. (2005) 69–77
18. DiVA-Dynamic Variability in complex, adaptive systems. `http://www.ict-diva.eu/`