Change Patterns: Co-evolving Requirements and Architecture

Koen Yskout *Riccardo Scandariato*
Wouter Joosen

Report CW 593, August 2010

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Change Patterns: Co-evolving Requirements and Architecture

Koen Yskout *Riccardo Scandariato*
Wouter Joosen

Report CW 593, August 2010

Department of Computer Science, K.U.Leuven

Abstract

Emerging classes of systems are more and more subject to changes in their requirements and environment assumptions. Such changes have a far-reaching impact across several artifacts. This paper argues that patterns of co-evolution (or change patterns) can be observed between "privileged" pairs of artifacts, like the requirements specification and the architectural design. The paper introduces change patterns as a precise framework to systematically capture and handle change. The approach is based on model-driven engineering concepts and is accompanied by a tool-supported process. Changing trust assumptions are presented as an example of security-related evolution, and are used to illustrate the approach.

Keywords : Co-evolution, MDE, security requirements, architecture.

1 Introduction

Supporting evolution in a principled way is becoming of uttermost importance for larger and larger classes of systems. For instance, the European Community has stimulated the research in the area of eternal systems, i.e., "long-lived systems requiring minimal intervention and management to thrive in spite of changes in usage, host device, network context or data and data protection formats." Such systems (including smart cards, portable devices, home appliances, and so on) are characterized by an operational time that is much longer than the building time. Due to their longevity, such systems inevitably face changes in their environment, their (security) requirements, as well as their design and implementation.

As a second example, services in the "Internet of Future" will be the result of very complex compositions spanning across multiple administrative domains, and will be operated by a heterogeneous consortium of stakeholders (service providers, platform providers, content providers, consumers) that will dynamically evolve over time, often in opportunistic ways due to business considerations. In this scenario, change is deemed as a normal business condition. For instance, trust relationships among the stakeholders of such services are very likely to change over time, along with other (security) requirements.

Clearly, change has a multi-level impact reaching from the requirements analysis down to the run-time configuration of systems. As illustrated by Mens et al., achieving co-evolution between different types of software artifacts is a challenging task that is still open to seminal research [13].

This work presents a novel approach to deal with co-evolution in a precise and practical way: *change patterns*. In general, a change pattern provides a framework to systematically evolve a certain artifact in face of changes taking place in another artifact, which is developed at a different level of abstraction. Intuitively, some artifacts expose a tighter relationship vis-a-vis change. That is, they co-evolve in ways that go beyond a generic, imprecise ripple effect. For instance, the development of the requirements and the software architecture are known to be closely intertwined [15]. In these cases, patterns of co-evolution (or change patterns) are likely to be observed.

As its main contribution, this paper introduces the idea of change patterns and provides a precise definition based on model-driven engineering principles. This foundation is illustrated via a specific family of change patterns that allows the co-evolution of trust requirements and secure software architectures. Further, change patterns are leveraged to build a (mini-)process that guides a software architect towards the design of a highly-evolvable system. The process assumes that a sufficiently complete catalog of change patterns is at hand. While an illustrative catalog has been developed and referenced by the authors, the creation of an extensive collection of change patterns goes beyond the scope of the intended contribution for this paper. Finally, to showcase the feasibility of the presented ideas, a prototype tool is presented.

Trust and trustworthiness are the basis of security, because the need for security in a system originates from the presence of untrusted entities. Therefore,

to be able to effectively secure a system, it is important to know (and explicitly state) which entities are trusted for certain tasks, and which are not. In the context of the emerging classes of systems mentioned above, the trust requirements are subject to constant change. Therefore, they are of particular importance in terms of evolution, and have been addressed in this work.

The rest of this paper is structured as follows. In Section 2 a high-level, informal description of change patterns is given. Section 3 introduces the running example used throughout this paper. The example comes from an industrial case study used for the initial validation of the presented approach. Sections 4 and 5 present the formal definition of change patterns and their illustration in the context of changing trust assumptions. Section 6 describes the supporting process and the prototype tool. A discussion is provided in Section 7. Finally, Section 8 presents the related work and Section 9 concludes the paper.

2 Change patterns at a glance

A change pattern is a reusable source of knowledge concerning the co-evolution of two related artifacts. The changes in a given artifact (called the driver) are characterized via a change scenario. In order to cope with this change, a change pattern provides guidance about how to transform the second artifact (called the companion). The guidance provides a principled way of executing a transformation that fulfills certain constraints, e.g., the minimization of the impact. The transformations in the companion can, in turn, generate feedback to the driver.

Depending on the type of artifacts involved, there can be many families of change patterns. For instance, patterns of evolution are likely to be found among design and implementation, requirements and test cases, and so on. To illustrate this promising idea, this paper focuses on requirements in the driver seat and the architecture as the companion. A change pattern from this family can be denoted as *change pattern*$_{RA}$, although the subscript is often left implicit in the remainder of this paper.

When the requirements of a system evolve, the system's architecture most likely needs to be updated to accommodate the changed requirements. Keeping the requirements and architecture synchronized, such that the system at all times fulfills its requirements, is a big challenge for a software architect.

Change patterns can help to achieve this challenge, by providing advice to the architect. In this paper, a change pattern characterizes a generic kind of change at the requirements level. This change is captured in a *change scenario*, which consists of a *requirements template* that describes a generic requirements model (i.e., independent from a concrete system), and a *change description* relative to that template. To interpret a scenario in the context of a concrete system, a *binding* needs to be defined in order to match the requirements template and the concrete system's requirements model. These concepts are further explained in Section 4.

Moreover, the pattern provides a collection of *architectural solutions* that

enable the system to respond to the change, while minimizing the impact on the architecture. This is important when the system that evolves has already been deployed, and recalling the system to carry on major changes to the architecture is prohibitive. Similar to the change scenario, the architectural solution also contains a generic *architectural template*, and a description of the necessary change relative to that template, called the *guidance*. The architectural template provides a reference point to start applying the change. This way, the guidance can be expressed in generic terms. Again, to apply the guidance in the context of a concrete system, a binding needs to be defined between the architectural template and the concrete system's architectural model. Finally, the solution describes the *feedback*, which is a description of the influence of the architectural changes on the requirements model. These concepts are further explained in Section 5.

3 Running example

The illustrations in this paper are based on a prototype of a 'home gateway', currently in development by a telecommunications operator. The home gateway is a device placed at a customer's home, through which the customer's devices (e.g., a laptop, a PDA, home appliances, and so on) connect to the operator network and the Internet.

The gateway performs access control, prohibiting access to insecure devices, for instance when a virus is detected or the configuration is not up-to-date. Additionally, the gateway acts as a service platform, enabling third-party service providers to offer home services to the customer, for example a domotics service. The service platform is based on the OSGi [18] specification, and home services are installed as bundles onto this platform.

In this paper, the focus lies on the infrastructure for purchasing and obtaining home service bundles.

4 Modeling change

Based on model-driven engineering principles [3], this section describes the models and metamodels that are used in this work to represent requirements, architectures, and change, together with the adopted notations.

4.1 Requirements model

We assume that the requirements are expressed in a requirements model R that conforms to a requirements metamodel $\mathscr{R}$ (written $R : \mathscr{R}$).

The requirements metamodel $\mathscr{R}$ itself conforms to a meta-metamodel $\mathscr{M}$. Multiple meta-metamodels have been defined and are used in practice, for instance OMG's EMOF [16] and EMF's Ecore [17]. All of these offer similar generic concepts, such as a Class and a Structural Feature (which is a property of a Class) in Ecore.

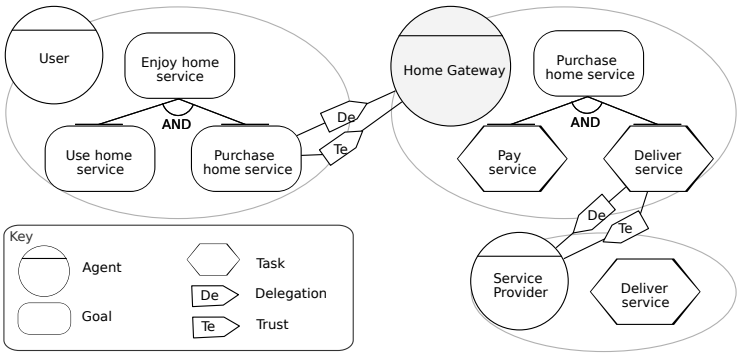


Figure 1: Requirements model in SI* for the running example (R)

Illustration Trust requirements are chosen as an illustration in this paper. Since trust relationships are a forefront concept in SI* [10], the requirement models in this paper are expressed using SI*. Note, however, that the general applicability of the presented approach is not impacted by this choice.

The SI* requirements model R for the running example is given in Figure 1. It contains three agents: User, Home Gateway and Service Provider. The home gateway is the system under development. The user and the third-party service provider are examples of external agents.

The model also defines various services. A service in SI* can be a goal (shown as a rounded rectangle), a task (represented using a hexagon) or a resource (depicted using a rectangle, but not shown in the figure). Some agents delegate the execution of a service to another agent, represented using a 'Delegation of Execution' (De) relationship. For example, the home gateway delegates the task of delivering a purchased home service to the service provider. Trust relationships are represented using a 'Trust of Execution' (Te) relationship, e.g., the home gateway agent trusts the service provider to (at least) correctly deliver a home service. Finally, it is possible to model 'Distrust of Execution' (Se) relationships (not shown in this picture).

In this work, all metamodels are expressed in Ecore. We have defined a metamodel $\mathscr{R}_{\text{SI*}}$ for a subset of SI*, which contains the concepts of agents, services (goals, tasks and resources), delegation and trust relationships. For instance, an agent is defined as an Ecore Class with a Structural Feature for its name. A trust relationship is also defined as a Class, with features such as the source agent, the target agent and the kind of the relationship.

4.2 Architectural model

The architecture is represented using a model A that conforms to an architectural metamodel $\mathscr{A}$. Typical architectural metamodels define concepts like components, ports, connectors, and required and provided interfaces. Clearly, an architectural metamodel also conforms to some meta-metamodel.

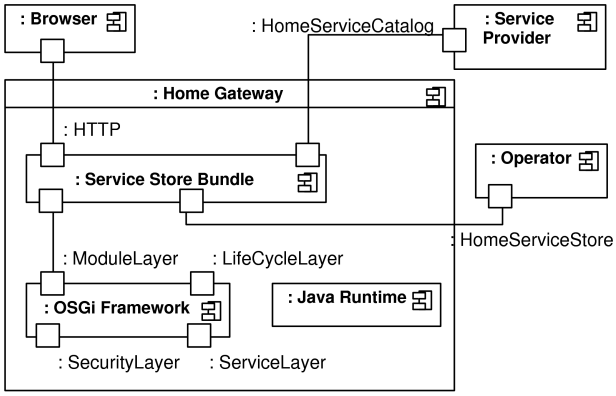


Figure 2: Initial architecture

Illustration In this work, UML 2 is used as the architectural metamodel, which has been modeled in Ecore as part of the Eclipse UML2 project.

The chapters on Components and Composite Structures of the UML specification are of particular interest, since they offer concepts that are well suited to represent architectural models. We graphically represent UML models using the structure diagrams and composite structure diagrams, which are also defined in the specification.

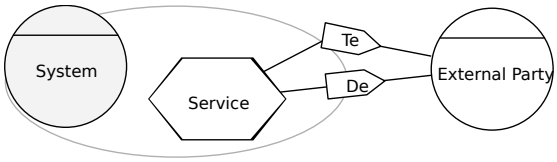
The architectural model for the running example is shown in Figure 2. The Home Gateway component is based on the Java Runtime and the OSGi framework [18]. The purchasing of home services is handled by the Service Store OSGi Bundle, which is accessed by the customer using a Browser. The bundle connects to the Operator to obtain a list of third-party service providers and services, which it presents to the customer. Purchasing a home service is handled by directly contacting the Service Provider.

4.3 Templates and binding functions

As explained in Section 2, a change pattern describes change in an abstract and context-independent way. To this aim, the notion of a template model is leveraged. A template model, written $\hat{T}$, captures a generic structure of model elements. It needs to be connected with a concrete model T that is subject to change. Note that the template and the concrete model both conform to the same metamodel.

To specify the connection between the template and the concrete model, a binding function $\mathbb{B}_{\hat{T}\to T} : \hat{T} \to T$ is defined. The binding must be well-formed, which means that all template elements need to be bound (it is a total function), and all relationships between the template elements need to be preserved.

Illustration Consider the requirements template $\hat{R}$ in Figure 3a. It contains a System agent that requires the execution of a Service task, for which it



(a) Requirements template in SI* ($\hat{R}$)

Requirements template $\hat{R}$	Requirements model R
sys : agent(System)	hg : agent(Home Gateway)
ext : agent(ExternalParty)	sp : agent(Service Provider)
svc : task(Service)	ds : task(Deliver Service)
rel(De, sys, ext, svc)	rel(De, hg, sp, ds)
rel(Te, sys, ext, svc)	rel(Te, hg, sp, ds)

(b) Binding $\mathbb{B}_{\hat{R}\to R}$

Figure 3: A requirements template and a binding for the running example

needs and trusts an External Party.

A well-formed binding $\mathbb{B}_{\hat{R}\to R}$ between the template model $\hat{R}$ and the requirements model R of the example in Figure 1 is given in Figure 3b. The System agent is bound to the Home Gateway, the External Party to the Service Provider, and the generic Service task is bound to the Deliver Service task. Also the delegation and trust relationships from the template are bound to the corresponding concrete elements.

4.4 Change model

The models discussed so far have been considered in isolation. In the context of evolution, subsequent versions of a model need to be related, e.g., when a new model has been obtained by modifying an element from the previous one. To express this relationship, we utilize a *change model* which defines specific changes from one model to another. In the rest of this subsection, requirement models are used as an example, but the same concepts are also applicable to an architectural (or any other) model.

4.4.1 Change metamodel

There are two possibilities for the definition of a change metamodel. First, the change metamodel $\Delta_{\mathscr{R}}$ can be defined in terms of the metamodel $\mathscr{R}$. It then defines change concepts such as adding or modifying an instance of a certain concept from $\mathscr{R}$. For example, a change metamodel $\Delta_{\text{SI*}}$ would include the concept 'modify the source agent of a trust relationship'. While more intuitive, this representation has the disadvantage that each distinct requirements metamodel requires its own definition of a corresponding change metamodel. For instance, the KAOS metamodel does not include the concept of a trust relationship as

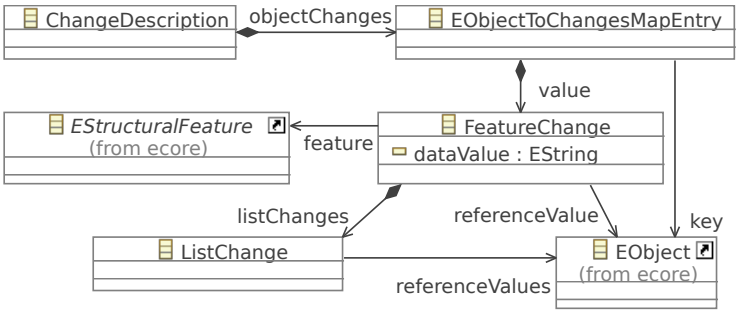


Figure 4: Ecore change metamodel Δ_{Ecore} (simplified) [17]

in SI*, but on the other hand defines the concepts of a requirement and an expectation. The change metamodel Δ_{KAOS} thus needs to differ from $\Delta_{\text{SI*}}$.

Alternatively, it is possible to define a change metamodel $\Delta_{\mathscr{M}}$ which depends exclusively on the meta-metamodel $\mathscr{M}$ (i.e., Ecore in our case). Such a change metamodel adds an extra level of indirection by moving the references to $\mathscr{R}$ to its instances (i.e., concrete change models). For example, for the Ecore meta-metamodel mentioned earlier, Δ_{Ecore} defines the concept of modifying an object's value for a Structural Feature. The modification of a relationship's source agent in SI* is not part of the change metamodel Δ_{Ecore}. It is expressible in a concrete change model, however, by specifying the object (the relationship that undergoes the change), the Structural Feature that is changed (the 'source agent' feature), and the new value. Note that this differs from the previous possibility, where 'modify the source agent of a relationship' was a first-class concept from $\Delta_{\text{SI*}}$. Because of its genericity, we follow this approach in this work.

While less intuitive, the chosen representation has as its main advantage that it can describe the changes in all models conforming to any metamodel that has been defined using the meta-metamodel $\mathscr{M}$. In practice, this means that the change metamodel for some meta-metamodel can be defined and implemented once and re-used for all metamodels defined in that meta-metamodel. For instance, if the metamodels for SI* and KAOS are both defined using Ecore, the same change metamodel can be used to describe all changes in both SI* and KAOS models.

Further, an implementation of Δ_{Ecore} is already available in the Eclipse Modeling Framework (EMF) for the Ecore meta-metamodel [17, Chapter 17].

Illustration In EMF, the change metamodel for Ecore is centered around changes related to Structural Features, as shown in Figure 4. A Structural Feature can represent an attribute of a primitive data type, or a reference to another object. A ChangeDescription consists of a mapping of Ecore objects (EObjects) to a set of FeatureChanges, each describing a change to some Structural Feature of that object. A FeatureChange can either represent a change to a single-valued feature, in which case it refers the new value object, or a multi-valued feature,

in which case it refers to a set of ListChanges. A ListChange can represent the addition, removal or movement of a value in the list.

4.4.2 Change characterization

As sketched in Figure 5a, we say that a change, defined with respect to a template model $\hat{T}$, is modeled as a *change characterization* $\hat{\delta}$. The change characterization expresses the change with respect to the elements in $\hat{T}$, resulting in a modified model $\hat{T}'$. To apply the change characterization to a concrete model T, we can derive the change model $\delta = \hat{\delta}[\mathbb{B}_{\hat{T}\to T}]$ by substituting each occurrence of a template element $\hat{t} \in \hat{T}$ in $\hat{\delta}$ with its concrete binding image $\mathbb{B}_{\hat{T}\to T}(\hat{t})$. The model T' (after the change) can be obtained by applying the change model δ, i.e., $T' = \hat{\delta}[\mathbb{B}_{\hat{T}\to T}](T)$. Note that a change characterization $\hat{\delta}$ describes the difference between the models T and T'. This is different from a transformation, which describes the process of achieving that change.

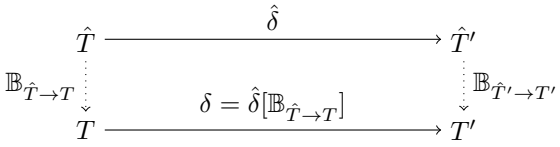
The binding $\mathbb{B}_{\hat{T}'\to T'}$ between the resulting template model $\hat{T}'$ and the resulting concrete model T' can be automatically derived when δ is applied: starting with a copy of the original binding $\mathbb{B}_{\hat{T}\to T}$, elements removed by $\hat{\delta}$ are removed from the binding, and elements that are added by $\hat{\delta}$ are bound to the corresponding new concrete elements introduced by δ. Accordingly, given the models $\hat{T}$, T, change characterization $\hat{\delta}$ and binding $\mathbb{B}_{\hat{T}\to T}$, the resulting models $\hat{T}'$, T', the change model δ and the binding $\mathbb{B}_{\hat{T}'\to T'}$ can be automatically derived.

Illustration Consider again the requirements template $\hat{R}$ shown in Figure 3a. It is possible that the System loses trust in the External Party over time. This means that the trust relationship (Te) is converted to a distrust relationship (Se), as shown in Figure 5b. Given our metamodel $\mathscr{R}_{\text{SI}^*}$ expressed in Ecore, this change is expressed using the Ecore change metamodel Δ_{Ecore} as hinted in Figure 5c. The 'kind' Structural Feature of the appropriate relationship object is changed from 'trust of execution' (Te) to 'distrust of execution' (Se).

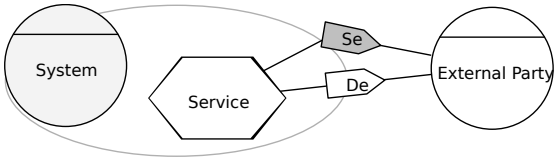
This is the change characterization $\hat{c}$ (a concrete example of $\hat{\delta}$), defined with respect to the template model $\hat{R}$. Using the binding $\mathbb{B}_{\hat{R}\to R}$ in Figure 3b, the change model c is obtained from $\hat{c}$ by replacing the occurrences of all template elements by their concrete image. This change model represents the change from a trust relationship to a distrust relationship between the Home Gateway and the Service Provider for the task Deliver Service.

4.5 Change patterns revisited

As mentioned in Section 2, a change pattern$_{RA}$ describes a generic evolution at the requirements level, and provides the appropriate actions to take with respect to the system's architecture when that evolution occurs. Hence, a change pattern is composed of two parts: the change scenario and the set of architectural solutions. An overview of the different parts of a change pattern and the



(a) Change characterization and binding functions



(b) Requirements template after change ($\hat{R}'$)

ChangeDescription { **objectChanges**:
  EObjectToChangesMapEntry {
    key: rel(Te, System, ExtParty, Service)
    value: **FeatureChange** {
      feature: Relationship::kind
      dataValue: 'Se'
} } }

(c) Example change characterization $\hat{c}$ modeled in Δ_{Ecore} (pseudo-notation)

Figure 5: Change characterization

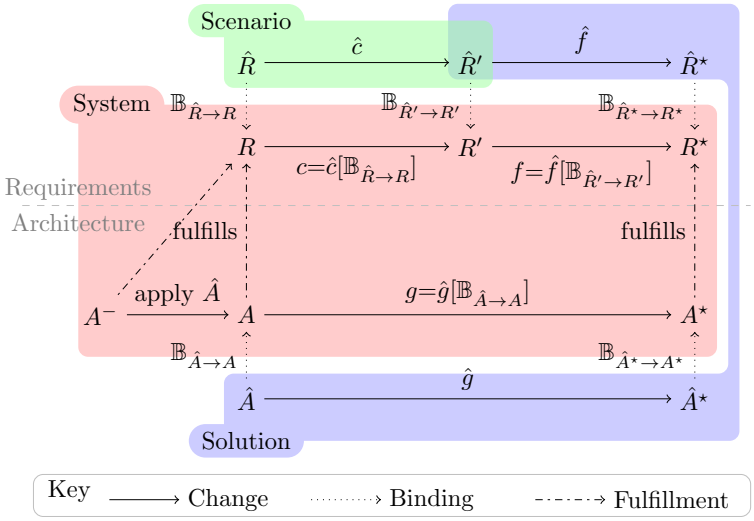


Figure 6: Formalization of a change pattern

relationships with a concrete system is given in Figure 6, using the notation introduced previously.

The *change scenario* (the green part in the figure) describes a generic kind of change at the requirements level. That change scenario is defined by a change characterization, called the *change description* $\hat{c} : \Delta_{\mathscr{M}}$. The change characterization is defined with respect to the *requirements template* $\hat{R} : \mathscr{R}$. The *anticipated model* $\hat{R}' : \mathscr{R}$ corresponds to the situation after the change has occurred and can be computed from $\hat{R}$. To interpret a scenario in the context of a concrete system (the red part of the figure), a binding function $\mathbb{B}_{\hat{R}\to R}$ needs to be defined. If multiple instances of the same change scenario occur, a separate binding needs to be provided for each instance.

An architectural solution (the blue part in the figure) consists of three elements. First, the *architectural template* $\hat{A}$ describes the reference infrastructure for the solution. It must be applied to a concrete system in order to use the solution, which means that a binding function $\mathbb{B}_{\hat{A}\to A}$ needs to be provided. Second, the *guidance*, in the form of a change characterization $\hat{g}$, describes how the change scenario can be implemented via a refactoring of the architecture, based on the architectural template $\hat{A}$. The resulting architecture $\hat{A}^\star$ can be derived from $\hat{A}$. Finally, the *feedback* $\hat{f}$ is a change characterization at the requirements level that describes how the solution affects the anticipated model $\hat{R}'$ when the architecture is updated by the guidance, resulting in a new template $\hat{R}^\star$.

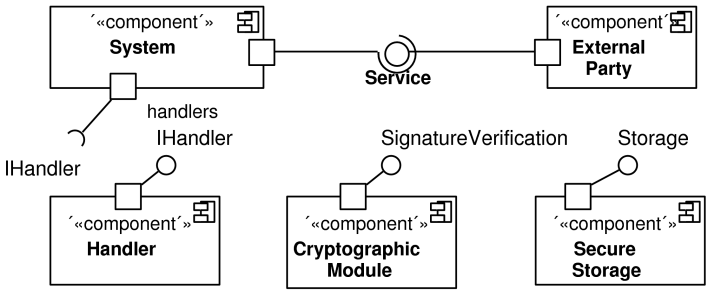


Figure 7: Architectural template ($\hat{A}$)

5 Architectural solutions

An architectural solution prescribes a strategy that can be followed when the associated change scenario occurs, while minimizing the impact on the architecture. Each solution is an alternative and may provide different guarantees and have different non-functional properties associated with it. Therefore, when a suitable solution is chosen, these aspects need to be known and taken into account.

The three parts of each solution (the architectural template, guidance and feedback) are explained in the following subsections.

5.1 Architectural template

An architectural template $\hat{A} : \mathscr{A}$ describes a certain structure and configuration of architectural elements. It generically describes the situation before the requirements have changed, while ensuring that all elements are identified that play a role in the architectural refactoring expressed by the guidance (discussed in the next subsection).

A template $\hat{A}$ is considered as applied to the architecture A if there exists a binding $\mathbb{B}_{\hat{A}\to A}$ that binds the elements from template $\hat{A}$ to elements from A. For clarity, the architecture without an applied architectural template is denoted as A^{-}.

Illustration Consider the 'disappearing trust' change pattern that characterizes the change from $\hat{R}$ in Figure 3a to $\hat{R}'$ in Figure 5b. This change pattern collects two possible solutions. The first solution describes how the system can monitor the external party, such that anomalies can be detected during the execution of the task. This solution is not detailed further in this paper. The second solution entails requiring a commitment from the external party before actually requesting the execution of the task. This corresponds to the non-repudiation pattern described in [5].

The architectural template for this solution is shown in Figure 7. The template is a composition of four parts. The first part (the top row) is aligned with

the requirements template. It defines two components 'System' and 'External party', corresponding to the agents in Figure 3a. Both components communicate through the 'Service' interface, which corresponds to the delegation of execution of the 'Service' task.

The next three parts, on the bottom row from left to right, need to be present in order to support the change guidance (described the next subsection). They prescribe (1) a mechanism such that the system's operations can be intercepted, and the invocation can be inspected (and possibly blocked) by configurable handler components; (2) the presence of a Cryptographic Module, which provides operations for verifying digital signatures. This module will be used by the system to verify the digitally signed commitments; (3) a component offering Secure Storage. This component will be used by the system to securely store the received commitments, so they can be used as proof in case of dispute.

Note that not all parts are connected to the System, e.g., the Cryptographic Module. This is because these parts are not strictly necessary at the present time, but must certainly be available when the guidance needs to be executed. At minimum, they force the architect to reason about the feasibility of introducing these dependencies at a later time.

5.2 Guidance

The guidance part of an architectural solution describes how the architecture must be updated after the requirements model has evolved to the anticipated situation. When this evolution happens, the guidance is applied automatically.

As mentioned, the guidance is a change characterization $\hat{g}$ defining the transition from the architectural template $\hat{A}$ to an updated (template) architecture $\hat{A}^{\star}$. By providing a binding $\mathbb{B}_{\hat{A}\to A}$, the guidance can also be applied in the context of the concrete architecture A.

Illustration The changes described by the guidance for the 'disappearing trust' change pattern, with respect to the architectural template $\hat{A}$ from Figure 7, can be broken down to four steps. First, the guidance defines the creation and connection of a new handler that is responsible for obtaining the commitment from the external party. Second, the external party is extended with support for providing such commitment, and the new handler is connected to the external party. In the last two steps, the new handler is connected to both the cryptographic module and the secure proof storage. More specifically, the guidance represents the following changes:

1. (a) define a new component CommitmentHandler that is generalized by the Handler component; (b) connect the System to the CommitmentHandler by defining a connector between the handlers port from System and the port on the Handler component;
2. (a) define a new interface CommitmentProvisioning, packaged in the ExternalParty component; (b) add a new port to component ExternalParty, which provides the interface CommitmentProvisioning; (c) add a new port to component CommitmentHandler, which requires the interface Com-

```
ChangeDescription { objectChanges:
  EObjectToChangesMapEntry {
    key:                    ExternalParty
    value:                  FeatureChange {
      feature:              packagedElement
      listChanges:          ListChange {
        kind:               ADD
        referenceValues:    new UML::Interface {
          name:             "CommitmentProvisioning"
} } } } }
```

Figure 8: Excerpt from the guidance $\hat{g}$ modeled in Δ_{Ecore} (pseudo-notation)

mitmentProvisioning; (d) connect CommitmentHandler to ExternalParty using the corresponding ports;

3. (a) add a new port to component CommitmentHandler, which requires the interface SignatureVerification; (b) connect CommitmentHandler to CryptographicModule using the corresponding ports;
4. (a) add a new port to component CommitmentHandler, which requires the interface Storage; (b) connect CommitmentHandler to SecureStorage using the corresponding ports.

Due to the complexity of the Ecore change model and the UML metamodel, we omit a detailed description of the entire guidance, except for change 2a from the list above, which is given as an illustration in Figure 8.

Behaviorally, the commitment handler gets invoked before the actual request for the service is issued. The handler will first request a commitment from the external party. When the commitment is received, its digital signature must be verified and, if valid, the commitment needs to be securely stored. Only then the actual request can be issued.

5.3 Feedback

A particular solution often has an impact on the requirements once the guidance $\hat{g}$ is applied. This impact is called *feedback*.

By following the guidance, additional requirements may be imposed on the system. Equally, the guidance might be effective only if additional assumptions can be expected to hold. These new requirements and assumptions are not part of $\hat{R}'$, and thus the requirements model needs to be updated to reflect the new architecture.

Alternatively, the effects of the foreseen evolution can be softened or reverted by means of some solutions. Thus, following the guidance may restore the requirements model to the model before the change, as if the change has never happened, or the impact of the change may be diminished. The feedback also captures these effects on the requirements model.

The effect of feedback is closely related to the Twin Peaks model [15]. The

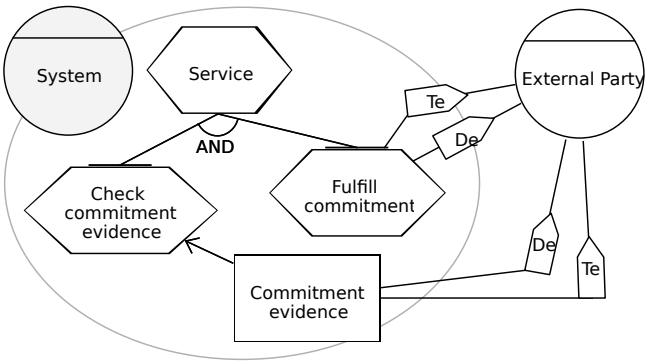


Figure 9: Template requirements model after feedback ($\hat{R}^\star$)

original Twin Peaks model describes the interwoven refinement of requirements and architecture in the design phase of the system. Here, however, we identified a different kind of feedback, which arises from evolution (and not refinement). When the requirements evolve after the system has been completely designed and deployed, and the architecture is updated to conform to these new requirements, this architectural update may again have an impact on the requirements. The impact can lead to the introduction of new elements in $\hat{R}'$ or the modification of elements already present in $\hat{R}'$.

The feedback is represented using a change characterization $\hat{f}$. It describes how the "after" requirements template $\hat{R}'$ needs to be changed to a model $\hat{R}^\star$ when the guidance is applied, such that the resulting architectural template $\hat{A}^\star$ fulfills the requirements $\hat{R}^\star$. It is not always necessary to provide feedback, and it can thus be an empty (i.e., changeless) change characterization.

Illustration The feedback for the 'disappearing trust' pattern describes the transition from the anticipated requirements model $\hat{R}'$ in Figure 5b to the one shown in Figure 9. The notion of a commitment, and the functionality for obtaining it, needs to be introduced in the requirements model after the guidance is followed, in order to synchronize the requirements and the architecture. The decrease of trust has been (partially) mitigated by means of this commitment; it is believed that the external agent will properly execute the service, because it has non-repudiably committed to it. Therefore, the trust relationship is restored. Due to space constraints, further technical details of the feedback are not described in this paper.

5.4 Soundness constraint

The three parts of an architectural solution for a change scenario (the architectural template, guidance and feedback) cannot be defined completely at will by the pattern's designer. Instead, the following property must always be ensured:

Applying the guidance $\hat{g}$ on an architectural template $\hat{A}$ results in

a valid architectural model $\hat{A}^\star$ which fulfills the requirements model $\hat{R}^\star$ that is obtained by applying the feedback $\hat{f}$ on the change scenario's anticipated model $\hat{R}'$.

6 Using change patterns

Change patterns can be collected in a catalog. For instance, a catalog has been created for the family of *change patterns*$_{RA}$ related to trust relationships [19]. However, an architect needs a (mini-)process to effectively use a catalog in order to design an evolvable architecture. The process described in this Section is accompanied by a proof-of-concept tool providing automated assistance [19].

The first input to the process is a requirements model R, representing the set of requirements and assumptions of the system. Additionally, the process requires an initial architectural model A^- that already fulfills the requirements, but has not yet been prepared for the evolution of these requirements. Therefore, the process cannot be used to design an architecture from scratch, but requires that the architecture is at least roughly defined already.

In the implementation, an additional traceability model is introduced to connect the requirements and architectural model for the change patterns that have been applied. In the traceability model, the bindings and decisions that are made during the process are recorded. Using this model, it is for instance possible to retrieve the applied change patterns and the solutions that have been chosen.

The process consists of three stages. First, during the change assessment phase, the requirements model is analyzed and the applicable change patterns from the catalog are selected. Second, in the preparation phase, the architecture is prepared with respect to the selected patterns. Both of these phases happen during the design (or refactoring) of the system. The final phase of the process occurs when the system's requirements actually change. Here, the architectural and requirements model are modified to a new, consistent state. Each phase is presented in some more detail in the next subsections.

6.1 Change assessment phase

In this phase, matches are sought between the system's requirement model R and the evolution scenario from each change pattern in the catalog. An evolution scenario fits with the system requirements if a well-formed binding $\mathbb{B}_{\hat{R}\to R}$ can be defined. For each fitting pattern, a *change scenario instance* is computed, which is defined by the combination of the (generic) change scenario and the binding with the concrete system. Also, the new "after" model $R' = \hat{c}[\mathbb{B}_{\hat{R}\to R}](R)$ is computed. This exploration of potentially interesting change scenario instances can be automated via pattern recognition techniques. The found scenario instances are stored in the traceability model for later reference.

For each change scenario instance, its relevance, importance and likelihood needs to be estimated. To determine the relevance, the transition from R to

R' is investigated. The architect, the analyst and the other stakeholders need to assess whether this transition is meaningful and realistic for the system and environment at hand. This may not always be easy, and may require some creative thinking. If the scenario is not relevant, it is discarded and the process continues with the next scenario instance.

The importance and likelihood estimation is similar, and in fact closely related, to the risk analysis of the system. The stakeholders will have to decide which of the following alternatives is the better choice in the context of the system: allocating resources to support the evolution scenario now, even though it may never happen, or ignoring the evolution scenario until it takes place, possibly leading to a greater cost in the future. Additionally, observe that other solutions, independent of the architecture, might be possible and preferable, and should be considered.

The decision to support or discard a scenario instance, and the reasons for that, should be explicitly documented. This information can be integrated in the traceability model.

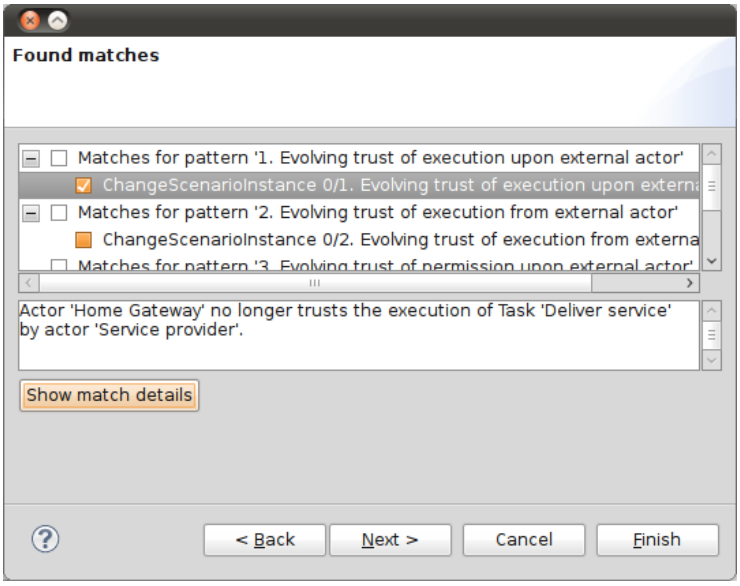
Illustration By matching the requirements template from Figure 3a to the initial requirements model in Figure 1, a change scenario instance is found in the form of a binding (Figure 3b). The obtained change scenario instance challenges the trust relationship between the home gateway and the third-party service provider. Assuming that the stakeholders agree that the loss of this trust relationship is relevant, likely and important, the scenario is selected for inclusion.

In the current implementation, the change scenario instances are discovered automatically using a Prolog-based pattern matcher. The resulting change scenario instances are presented to the user (Figure 10a). The user can then view the details for each of these scenarios, and select to accept or reject the change, as shown in Figure 10b. The scenario instances, together with the choices of the user, are persisted in the traceability model.

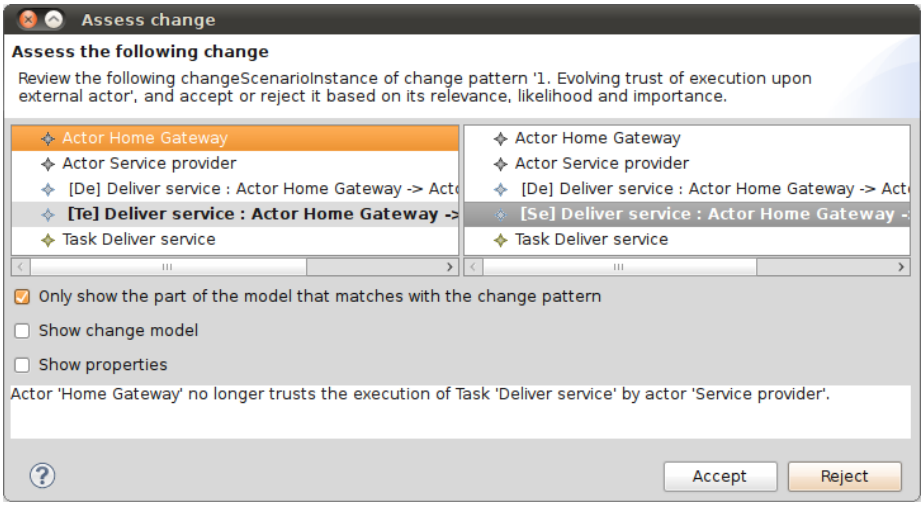
6.2 Preparation phase

If it has been decided that the scenario needs to be supported, a suitable solution needs to be selected. The suitability is determined by, for instance, the feasibility of applying the architectural template $\hat{A}$ to the current architecture, the guarantees given by the solution, the implied trade-offs, and the solution's feedback. If necessary, multiple solutions for the same scenario can be selected and implemented simultaneously, e.g., to implement multiple layers of protection. Again, the chosen solutions are recorded in the traceability model. Note that, if no solution from the catalog seems to be suitable, a custom solution should be defined, possibly outside the change patterns framework. Providing an extensive catalog of change patterns is not our main goal at this time, and the current catalog should be extended in the future.

When a solution has been chosen, it needs to be ensured that the associated architectural template $\hat{A}$ is in place. The architectural model A^- is prepared for



(a) The found change scenario instances



(b) Assessing a change scenario instance

Figure 10: Support provided by the prototype

the scenario by applying the template, resulting in a new architectural model A.

Illustration Take the case where the solution based on commitments from Section 5 is chosen by the stakeholders. The initial architecture of the system (Figure 2) is prepared so that it implements the template $\hat{A}$ from Figure 7. The resulting architecture is shown in Figure 11, together with the binding $\mathbb{B}_{\hat{A}\to A}$. Elements that correspond to elements from the template are shaded in gray and annotated using notes.

The service store bundle is extended with a handler mechanism. This allows external handlers to intercept the purchase operation and perform additional actions, possibly preventing the purchase from being requested. Currently, no handlers have been configured yet. To provide cryptographic operations, the Java Cryptography Architecture (JCA) is used, which is included in the default Java Runtime Platform. The operator further decides that the secure storage, necessary to store proof of the service provider's commitment, will not be located on the home gateway itself, but provided by the operator's platform.

At present, the architect has to manually inject the template. Afterwards, the architect can use the tool to specify the binding between the architectural template and the architecture, which is then stored in the traceability model.

6.3 Evolution phase

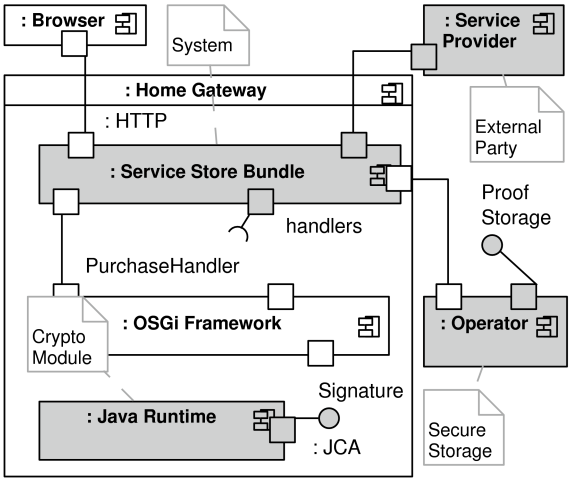
Later in the lifetime of the application, if and when the requirements of the system evolve, the traceability model can be used to determine whether some evolution corresponds to a foreseen scenario instance. If this is the case, the guidance $\hat{g}$ that belongs to the chosen solution is applied automatically, and the application's architecture is updated to the new situation.

Additionally, the requirements model is updated by applying the feedback $\hat{f}$. This step can also be performed automatically, but is not yet implemented.

Illustration Assume the trust upon a service provider disappears, because customers complain about the provider's failure to deliver purchased services. This corresponds to the change scenario for which a change pattern was selected and installed. The guidance from the solution needs to be applied to the system's architecture using the binding from Figure 11b. This is done automatically by the tool when the change scenario instance is triggered. The guidance is executed, resulting in an updated UML2 model.

The resulting architecture is shown in Figure 12. Thanks to the preparation, the impact of this change is limited: the commitment handler is added to the home gateway and wired to the other components, and the service provider needs to offer a commitment provisioning service. Other components, especially those within the home gateway device which are expensive to recall or update, remain unchanged during this evolution.

Note that the requirements model also needs to be updated, according to the feedback given by the solution. Due to space constraints, the resulting requirements model is not shown, but it can be inferred by combining Figures 1,



(a) Prepared architecture (A)

Architectural template $\hat{A}$	Architecture A
System	Service Store Bundle
A	Service Provider
Cryptographic Module	Java Runtime
Secure Storage	Operator
IHandler	PurchaseHandler
Service	Purchase
SignatureVerification	Signature
Storage	Proof Storage

(b) Binding ($\mathbb{B}_{\hat{A}\to A}$)

Figure 11: Prepared architecture and binding

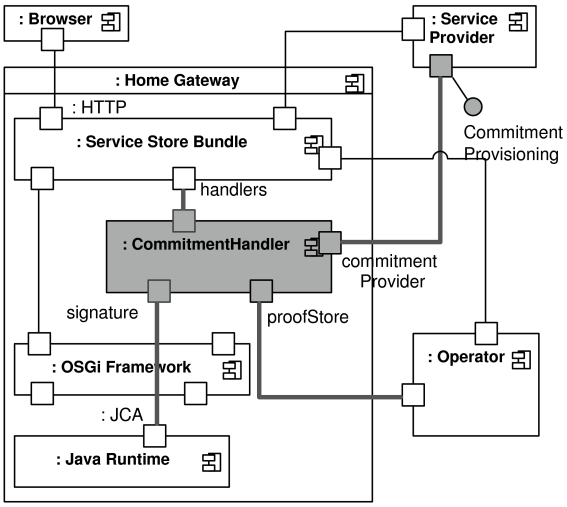


Figure 12: Evolved architecture ($A^\star$)

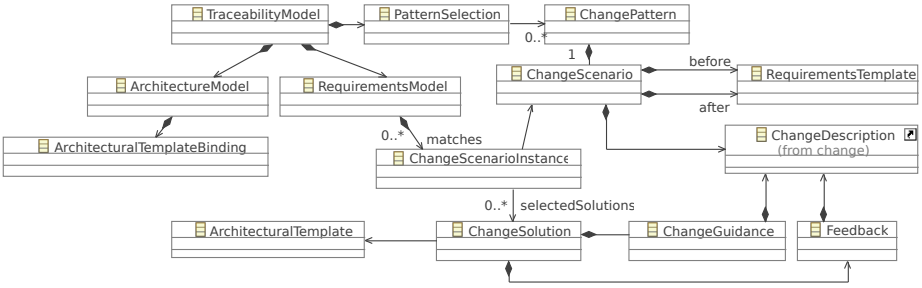


Figure 13: Change patterns Ecore model (simplified)

3b, and 9.

6.4 Behind the scenes

The prototype tool is implemented as an integrated collection of Eclipse plugins, and is based on the Eclipse Modeling Framework (EMF). All concepts from this paper are modeled using an Ecore model. A simplified version of the model can be found in Figure 13.

The matching of a change scenario with a requirements model is done by a matcher. Each change pattern can provide its own implementation of a matcher. For the change patterns using the SI* requirements metamodel, the implementation is based on Prolog. The matcher automatically translates the requirements template to a Prolog rule, with the template elements as variables. The body of the rule describes the structure of the template using the variables. The con-

crete requirements model is translated to a set of facts. A Prolog query then computes all variable bindings, each of which corresponds to a binding in the sense of this work.

Change models are implemented using the ChangeDescription class from EMF. To apply the change description from the scenario to the actual requirements model, we extended ChangeDescription to BoundChangeDescription. A BoundChangeDescription wraps a regular change description, and enriches it with a binding. Before applying the change description, the binding is used to convert all template objects to their concrete counterpart objects.

The tool takes advantage from the standard Eclipse extension mechanism by defining two new extension points. New patterns or new solutions for existing patterns can easily be added by creating a plugin that provides an extension. While the current tool can handle only SI* and UML models, it has been designed to be easily extensible with other meta-models.

7 Discussion

The main goal of a change pattern$_{RA}$ is to reduce the architectural impact of a change in the requirements, by shifting the cost (effort, time, and so on) for the change management from the maintenance phase to the design phase. A change can impact the architecture to various degrees. A change with local impact only changes a small and related set of architectural elements. A non-local change modifies multiple elements, but still complies with the overall architectural design (e.g., the used styles are preserved). Contrarily, an architectural change modifies the architecture in more fundamental ways. A change pattern should enable the architect to limit the impact of a requirements change during system maintenance to a local architectural change, perhaps by performing non-local or architectural changes at design time.

Additionally, the proposed process forces the architect to explicitly assess a set of evolution scenarios for the system, corresponding to the scenarios found in the catalog of patterns. This systematic exploration helps the architect in achieving a more complete solution. The success of the approach, however, largely depends on the extensiveness of the catalog that is available to the architect, as the best solution for a foreseen evolution scenario may not be part of the catalog. This paper is meant to introduce and formalize the fundamental concepts related to change patterns; providing an extensive catalog is beyond the scope of this work.

Even if a suitable pattern is found in the catalog, the applicability the architectural solution can be limited by the existing architecture's style. It is possible that the architecture needs to be significantly refactored in order to integrate the architectural template. When this is the case, it is important for the architect to carefully consider the alternative options, such as choosing a different solution that fits better, or deciding not to prepare for the evolution scenario at that point in time.

As a remark, it was noted that the guidance captures the net difference

between the architectural template models before and after the change, but does not describe the actual process of achieving that change. Applying the guidance to a concrete architectural model may require additional changes, e.g., when an architectural metamodel is used in which additional consistency rules are imposed on the instances. In such cases, the solution should be augmented with a model transformation that achieves the extra changes and has the same effect as the guidance. Despite the slight conceptual difference, accommodating for model transformations does not conflict with the overall approach and the required modifications to the implementation are straightforward.

8 Related work

Software co-evolution has been identified as a challenging research frontier [13]. Consequently, the state of the art is limited, especially when it comes to precise approaches. In the context of change management, Han [11] uses the term 'change patterns' to describe event-condition-action rules propagating change across two artifacts (namely from class design to implementation). The change propagation is essentially reactive and unidirectional, i.e., no feedback is foreseen. Côté et al. [6] describe an informal approach to evolution, based on problem frames, a requirements engineering methodology. Architectural patterns are assumed to be associated to corresponding problem frames via traceability links. In light of evolution, new sub-frames may emerge in the specification and the architect can extend the design by incorporating additional architectural patterns, whose selection is facilitated by the above-mentioned links.

Considerable research has been done in the area of requirements and evolution. For instance, change cases [7] are an extension of use cases, aimed at capturing requirements for expected future changes. In the context of the NFR framework, changes related to non-functional requirements have been investigated in [4]. More research about requirements evolution, and a research agenda, are given in [8]. Specifically for security requirements, Nhlabatsi et al. [14] provide an extensive survey of security requirements engineering approaches and their support for evolution.

Research in architectural evolution is also relevant for this work, especially as a way to characterize and define the guidance. For example, Garlan et al. [9] propose the concept of 'evolution styles', which are patterns that can be used by the architect to plan incremental evolution paths (orthogonal to requirements evolution) from an initial architecture to some target architecture. Evolution styles can be seen as a form of guidance, giving strategies to the architect on how to approach an architectural evolution. Graph transformations have also been used to model architectural evolution, for instance in [12]. The TranSAT transformation language [1] achieves evolution by weaving architectural aspects into the architecture.

In summary, the requirements and the software architecture research communities have developed methods that facilitate the evolution of either requirements or architectures. However, the state of the art is still lacking a principled

and precise approach that synergically handles the co-evolution of both.

9 Conclusions

In this paper, change patterns have been introduced as a promising idea that uncloses the opportunity for a new stream of research. The main goal of a change pattern is to provide guidance to cope with the impact of a change scenario across artifacts. The change patterns have been precisely described in model-driven terms and their potential as a means to support automation has been illustrated.

Initial validation of these ideas in the context of an industrial case study is providing encouraging results. However, empirical experiments are foreseen to perform a quantitative evaluation.

Further, the authors are experimenting with algorithmic techniques to detect the occurrence of change scenarios in one artifact (e.g., by means of incremental model queries [2]) in order to trigger the execution of the appropriate guidance and feedback in the co-evolving artifact. This way, fully automated co-evolution can be achieved, which extends the applicability of change patterns to the realm of self-adapting systems.

References

[1] O. Barais, A. Meur, L. Duchien, and J. Lawall. Software architecture evolution. *Software Evolution*, pages 233–262, 2008.

[2] G. Bergmann, A. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Okrös. Incremental Model Queries over EMF Models. In *ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems*, 2010.

[3] J. Bézivin. Model driven engineering: An emerging technical space. *Generative and Transformational Techniques in Software Engineering*, pages 36–64, 2006.

[4] L. Chung, B. Nixon, and E. Yu. Dealing with change: An approach using non-functional requirements. *Requirements Engineering*, 1(4):238–260, 1996.

[5] L. Compagna, P. El Khoury, A. Krausová, F. Massacci, and N. Zannone. How to integrate legal requirements into a requirements engineering methodology for the development of security and privacy patterns. *Artificial Intelligence and Law*, 17(1):1–30, 2009.

[6] I. Côté, M. Heisel, and I. Wentzlaff. Pattern-based Exploration of Design Alternatives for the Evolution of Software Architectures. *International Journal of Cooperative Information Systems*, 16(3-4):341–366, 2007.

[7] E. Ecklund Jr, L. Delcambre, and M. Freiling. Change cases: use cases that identify future requirements. *ACM SIGPLAN Notices*, 31(10):342–358, 1996.

[8] N. Ernst, J. Mylopoulos, and Y. Wang. Requirements evolution and what (research) to do about it. pages 186–214. Springer, 2009.

[9] D. Garlan, J. Barnes, B. Schmerl, and O. Celiku. Evolution styles: Foundations and tool support for software architecture evolution. In *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture 2009 & European Conference on Software Architecture 2009*, pages 131 –140, 2009.

[10] P. Giorgini, F. Massacci, and N. Zannone. Security and trust requirements engineering. *Lecture notes in computer science*, 3655:237, 2005.

[11] J. Han. Supporting impact analysis and change propagation in software engineering environments. In *The 1997 8th IEEE International Workshop on Software Technology and Engineering Practice, STEP*, pages 172–182, 1997.

[12] T. Mens, J. Magee, and B. Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *Computer*, pages 42–48, 2010.

[13] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *Principles of Software Evolution, Eighth International Workshop on*, pages 13–22, 2005.

[14] A. Nhlabatsi, B. Nuseibeh, and Y. Yu. Security requirements engineering for evolving software systems: A survey. *Journal of Secure Software Engineering*, 1:54–73, 2009.

[15] B. Nuseibeh. Weaving together requirements and architectures. *IEEE Computer*, 34(2):115–117, 2001.

[16] Object Management Group. OMG Meta Object Facility Core Specification, Version 2.0. `http://www.omg.org/mof`, 2006.

[17] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2009.

[18] The OSGi Alliance. OSGi – The Dynamic Module System for Java. `http://www.osgi.org`.

[19] K. Yskout, R. Scandariato, and W. Joosen. Change patterns website. `http://people.cs.kuleuven.be/koen.yskout/changepatterns`.