Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. (2012)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1575

Evolution styles: using architectural knowledge as an evolution driver

Carlos E. Cuesta[1], Elena Navarro[2,*,†], Dewayne E. Perry[3] and Cristina Roda[4]

[1]*Department of Computing Languages and Systems II, ETS Ingeniería Informática, Campus de Móstoles, Rey Juan Carlos University, 28934, Madrid, Spain*
[2]*Computing Systems Department, University of Castilla-La Mancha, Avda. España s/n, 02071, Albacete, Spain*
[3]*Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX USA*
[4]*Vector Corp., Parque Científico y Tecnológico de Albacete, Paseo de la Innovación 3, Centro de Empresas de I+D, 02006, Albacete, Spain*

ABSTRACT

Software evolution is an increasingly challenging and compelling concern for every developed software system because of the changes in the requirements, the technology, etc. When software evolution is carried out, software architecture emerges as one of the cornerstones that should be considered from two different points of view: as an artifact for the evolution, as it helps the architect plan and restructure the system, and as an artifact of the evolution, because it has to be modified as well. This paper focuses on the second point of view—that is, on the evolution of the software architecture, but taking into account architectural knowledge as a key driver of the process. Given that architecture rationale and design intent are critical in evolving software systems, it is imperative that they be captured in some useful form to aid that evolution process. We present a new approach for evolution styles that extends them by considering in their description the architectural knowledge as a valuable asset of the evolution process. Copyright © 2012 John Wiley & Sons, Ltd.

Received 29 July 2011; Revised 26 September 2012; Accepted 10 October 2012

KEY WORDS: software architecture evolution; architectural knowledge; evolution styles

1. INTRODUCTION

Software evolution is an essential feature of every developed system [1, 2]. There are always compelling arguments that lead us to change the developed system to adapt it to market trends, technological advances, or, simply, new customers' requirements. This feature was already pointed out in the eighties when the first law of software engineering was stated by Bersoff *et al.* [3]: 'No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle'; even previously, Lehman had stated the first law of software evolution [4, 5], a work that would be later extended and developed several times [4, 6, 7]. Therefore, as the need for change will last over the whole life cycle of the systems, the introduction of proper processes, techniques, and tools that help us deal with it becomes an essential part of the software development and maintenance process.

When we tackle software evolution, software architecture (SA) emerges as one of the cornerstones that should be considered from two different points of view: as an artifact *for* the evolution and as an artifact *of* the evolution. That is, SA is an artifact that can be used *for* evolving software as it acts as a shared mental model of a system expressed at a high level of abstraction [8], helping the architect plan and restructure the system by abstracting away from technological issues [9]. This alternative has been named *architecture-based software evolution*. But SA is also one of the results *of* the evolution because

*Correspondence to: Elena Navarro, Computing Systems Department, University of Castilla-La Mancha, Avda. España s/n, Albacete, Spain.
†E-mail: elena.navarro@uclm.es

it may also evolve to be consistent with the changes as they emerge. Therefore, the introduction of mechanisms, techniques, processes, etc. that guide the architect while evolving the SA becomes a critical issue to maintain its consistency, quality, etc.

In the very beginning of the SA area of research, the *architectural design rationales* (ADRs) that explain the SA specification were understood as very valuable assets of the architecture process. Perry and Wolf, in a well-known article [10], defined the SA as a model composed of *elements*, *form*, and *rationale*. The first item refers to the description of components and what would be later defined as connectors; the second item refers to constraints in their properties and relationships; and the third one was defined as the motivation for the choice of style, form, or elements, which explains why this choice satisfies the system requirements. However, the rationale has been scarcely considered until recently when a group of researchers highlighted again its importance [11, 12]. It can be seen as a computational structure, composed of small assets of design knowledge, tracing back to some requirements and forward towards an implementation: the extended discourse of the system's design, defining our *architectural knowledge* (AK). AK is therefore composed of architectural elements, requirements, and a number of design assets. There are several ways to represent them—for example, we talk about architectural design decisions (ADDs) and ADRs, which comprise a concrete decision in the process and the reasoning behind it. When only the final architecture is described, all this information is *unrepresented design knowledge* [13]; but now architecture includes this information as part of the rationale.

When we take into account the importance of a rationale to understand why the system is the way it is, it becomes a challenging question to evaluate what the implications are while evolving the system. Bratthall *et al*. [14] dealt with this question by carrying out an experiment with 17 subjects from both industry and academia and concluded that most of the interviewed architects stated that by using the ADRs they could shorten the time necessary to carry out the change tasks. Interviewed subjects also concluded that the quality of the results was better using ADRs when they had to predict changes on unknown real-time systems. Therefore, there are compelling arguments for the exploitation of the rationale while the SA is being evolved. However, this use of the rationale turns it into a *passive actor* of the evolution process as it is simply used as a documentation artifact. It is at this point where this work focuses its attention: can a rationale become an *active actor* of the evolution that drives its application? In this work, we describe how this question can be answered positively by means of evolution styles [5]. Concretely, an extension to these styles, which we have called *AK-driven evolution styles* (AKdES), is presented, which considers the introduction of ADDs and ADRs in their description to guide the architect while evolving the SA.

This paper is structured as follows. After this introduction, Section 2 describes the necessary background about software evolution and AK. Section 3 presents our proposal, AKdES, which intends to combine the novel concept of evolution styles with the basis provided by AK. Section 4 introduces a specific process, ATRIUM [15], as a proof of concept to show the applicability of our proposal. Section 5 demonstrates by means of a case study how our approach can be put into practice. And finally, Section 6 describes our conclusions and further research.

2. RELATED WORK

The implications of the SA in software evolution have been highlighted largely in the literature [16–18]. Most of the works focus their efforts on the exploitation of the SA as a key driver for the evolution, that is, what have been called an artifact for the evolution. It is worth noting that these works exploit the SA specification as the AK of the system. For instance, an interesting work in this area is the Evolution Tailored with AK framework proposed by Noppen and Tamzalit [19]. In this work, a new concept is introduced named *architectural trait*, which refers to the properties the architect wants to consider during the evolution no matter the evolution pattern he wants to apply. These architectural traits are defined by identifying a set of components and connectors that the architect wants to examine to determine their relevance in the evolution to be performed. Therefore, the use of architectural traits entails turning the SA evolution into an evaluation process that assesses the relevance of the architectural traits with regard to the architecture space that the architect retrieves from the architecture

DOI: 10.1002/smr

knowledge base. However, this architecture knowledge base does not describe design decisions (DDs) and design rationales (DRs) *per se* but different SA specifications that can be reused to address different needs. Therefore, although it exhibits some similarities to our approach, it also exhibits clear differences.

Another related work that also considers the exploitation of a knowledge base to enact the software evolution has been presented in the area of architecture-based self-adaptive software. For instance, the Knowledge-Based Architectural Adaptation Management approach [20] introduces as first-class architectural entities both adaptation policies and relevant system knowledge, which are dynamically managed and reasoned over by an expert system to drive the adaptation process carried out by the framework. The system knowledge refers to *architectural observations* that come from either events of the system itself or external information communicated to the system and can be related to structural changes or potential problems. An architectural adaptation manager uses these architectural observations to determine which adaptation responses must be generated—that is, which structural modifications or adaptation policy changes must be performed. In a similar way to the previous work, this work does not deal with ADDs or ADRs.

Several approaches have focused on the evolution of the SA where they consider SA as an artifact of the evolution. Several points of view have been considered for this evolution. Some works have focused on how the quality factor *evolvability* can be introduced and considered during the SA design. For instance, the adaptability evaluation method [21] specifies adaptability requirements and guides the architect in their consideration during the SA design by means of a set of guidelines and in their analysis to determine whether these requirements are met. Another work focuses on the SA evolvability evaluation during the final phase of its design. In this category, a well-known proposal is the SA analysis method [22], which evaluates the SA relative to several scenarios that describe likely future changes to the system, helping to estimate the amount of work to carry them out.

Finally, another alternative for dealing with SA evolution is by using AK. As stated by Avgeriou *et al*. [23], the AK turns into an integrated representation of SA specification, ADDs, and ADRs. Most of the works [24–26] in this area have been related to the ADD modeling to describe their structures, traceability relationships, etc. Other works [27, 28] have focused on the definition of supporting tools for AK management. Many approaches have also highlighted the existing close relationship between AK and SA evolution. For instance, Burge, Carroll, and McCall stated in [29] that as the rationale describes the history of how and why the system has been modified over time, it 'should be captured for the change' to determine where problems have usually happened and where they are likely to emerge. Other approaches [30, 31] have also highlighted that the necessary work to specify AK really pays off when the system is in its maintenance and evolution phases by helping to reduce the costs of these phases.

However, despite the importance of AK during the evolution process, to the best of our knowledge, only two approaches expressly exploit the AK as an active actor of the evolution process. One is that presented by Tang *et al*. [32]. They presented a graphical model, named Architecture Rationale and Element Linkage, which is described as a Bayesian belief network, to describe the causal relationships between ADD and architectural elements [33]. Using this representation, three probability-based reasoning methods can be applied: (i) *predictive reasoning* is used to determine which design elements could be affected by a change if some architectural elements were to change; (ii) *diagnostic reasoning* is introduced to determine what could be the causes for those architectural elements to change; and (iii) *combined reasoning* is used to combine the previous results and reason about what the likely changes to the system are. This work is very interesting in that it enables the architects to predict the impact of change before it happens. Therefore, this work can be considered as complementary, as our work is oriented to guiding the evolution process.

Another one is that proposed by Tibermacine *et al*. [34]. They proposed the introduction of a family of languages, called architectural constraint languages, that allow the specification of the architectural choices associated to the decisions that are made during the development process. There are two main constituents of the architectural constraint languages: (i) the *core constraint language*, a modified version of the Object Constraint Language, which provides navigation operations, operators, and usual quantifiers; and (ii) a set of Meta-Object Facility [35] meta-models to describe abstractions found in the main modeling languages that are used at different stages of the life cycle. Each pair of

DOI: 10.1002/smr

both elements is called a *profile*, being defined a profile for each stage of the development process and modeling language. For instance, authors proposed in this approach that a profile could be used to formalize the architectural constraints related to the architectural design stage and the Acme Architecture Description Language [36]. The main advantage of this proposal is that architectural decisions, such as architectural styles [37] or design patterns used for the system, can be described by means of the formal language core constraint language and be related to the specific architectural elements. It means that whenever a change is performed, the conformance of the architectural model can be checked after its evolution. Therefore, this approach is more oriented to validation rather than guidance of the evolution process.

The notion of architectural style, as originally introduced by Perry and Wolf, is a concept 'which abstract elements and formal aspects from various specific architectures' [10]. It is a *prescriptive* rather than a *descriptive* concept: the same system or *configuration* may simultaneously comply with several style definitions, that is, the resulting architecture would have several styles at the same time. The idea is that the style defines a series of constraints—the range of systems that fulfill these constraints gathers the members with that style. As the style becomes more specific, this range is smaller, and the prescription becomes almost a description.

There is an obvious relationship between software evolution and architectural styles, although it has not always been made explicit. The style guarantees that the architecture holds a series of properties—and usually, these properties are high-level features that are maintained during the system's evolution. Therefore, different configurations of an evolving architecture have the same style—that is, in general terms, the architectural evolution respects the defined style. Disregard of architecture and constraining styles lead to architectural drift and erosion [11].

In recent times, several authors have tried to exploit this intuition, developing a new, although related, concept—that of evolution style. In short, where the original architectural style described the constraints to be fulfilled by a set of systems, in an evolution style, this set of systems would be the set of potential configurations of an evolutionary system—that is the style defines the constraints to be fulfilled by any possible evolution of the system. Where the original notion prescribed the architecture, this variant prescribes its evolution.

Tamzalit and others [19, 38, 39] have developed a series of papers on the topic. Their notion of evolution style uses the original concept of architectural style to delimit the range of changes to a given system. They intend to provide generic transformations (*evolution patterns*) that describe these changes as series of steps (*evolution operations*). To be generic, these transformations have been defined to fulfill the constraints of an architectural style—therefore, evolution is within (or towards) a style. These concepts have been developed on top of graph transformations, at least at the level of a proof of concept. Although Tamzalit's work explicitly refers to the impact of evolution styles on AK [19, 38], none of these proposals make use of existing AK to define or even influence those styles. Much of this information is system specific, and this makes it even more relevant to our work. Our proposal intends exactly to fill this gap, that is, to define evolution styles that make explicit use of AK.

Inspired by their work, Garlan *et al*. [9] have also made some developments in this context. Their evolution styles are directly defined as transformations on styles: they provide an *initial* style, a *target* style, and possibly several sequences of intermediate evolutionary steps, known as *evolution paths*. These paths define different ways to evolve from the initial to the target style, using architectural *operations* to generate 'evolving' intermediate styles. They also define a set of *path constraints*, which are specified using temporal logic and predefined evaluation functions. This makes it possible for the architect to perform certain analysis. Our approach, AKdES, is to some extent similar to those mentioned—the style will be described as a sequence of steps. But the defining feature is that our evolution decisions are influenced by the existing AK. In the Garlan *et al*. approach, path constraints are bound to make decisions using just structural information, such as the number of nodes—thus the need for predefined functions. However, in our approach, any decision ever made is still available, so our evolution decisions can take even past decisions into account. In particular, our approach is able to detect when the conditions that led to a decision in the past do not hold anymore and to decide if this change in conditions is enough to cause, or least to enable, an evolutionary step. Our approach is designed not to depend on any existing description

DOI: 10.1002/smr

language or platform, although it has to be adapted to any specific approach. Our only assumption is that we use structured AK (SAK)—that is, that the system includes an explicit representation of AK.

Table I summarizes the conclusions presented in this state-of-the-art section. Every existing proposal is compared with the rest of them (including our proposal, AKdES) using a set of features. Namely, the second column states if the proposal has explicit support for AK (or some comparable concept, in the case of *partial* support), whereas the third one indicates if this AK is structured. The fourth one relates to evolution and states if this proposal has an explicit support for evolution using some structural concept. The fifth one indicates if the proposal uses explicit AK to *evaluate* if some evolution must happen, whereas the sixth states if the proposal includes *explicit* evolutionary steps as the result of that evolution. Finally, the last column indicates if the kind of evolution it supports is significant at the architectural level.

As can be observed, Table I makes it easy to identify four different categories. The first group (the first four proposals) is composed of methods that do not use explicit AK, but some comparable abstraction; except for Knowledge-Based Architectural Adaptation Management, all of them use this knowledge just for evaluation purposes, but they do not trigger an evolution process as the result of this evaluation. Only two of them consider this evolution at the architecture level. The second group (next two proposals) includes methods that have explicit AK, and this is evaluated for evaluation purposes, but it does not include explicit evolution structures or processes. The third group (next two proposals) joins the existing definitions of evolution styles; almost reversing the second group, they include support for evolution structures and processes, but they do not use AK to make their decisions. Finally, the last group (including our own proposal, AKdES) is defined by having explicit AK support, and it uses this knowledge to trigger an evolution process, supported in turn by explicit structures (i.e., styles), acting at the architecture level. In summary, this can be found at the intersection of the second and third groups—including both AK and evolution styles.

3. PROPOSAL: COMBINING EVOLUTION STYLES WITH ARCHITECTURAL KNOWLEDGE

3.1. Architectural knowledge as an evolution driver

As noted earlier, the notion of evolution style has been initially defined [39] as both a variation and an extension of the concept of architectural style, as originally conceived by Perry and Wolf [10]. Styles express the central idea of the *prescriptive* view of SA—that is, the architecture prescribes a set of design conditions, which are specified as a set of architectural constraints: every configuration that fulfills these constraints complies with the corresponding architectural style. The notion has become popular, and it is also considered from a descriptive point of view—where it is usually expressed as a *vocabulary* set of architectural elements and a set of integration constraints, mainly of a topological nature.

Table I. A comparison of existing proposals considering both architectural knowledge and evolution to some extent.

	Architectural knowledge	Structured AK	Evolution structures	AK-based evaluation	AK-based evolution	Architecture-level evolution
ETAK	≈ (traits)	×	≈	√	×	×
KBAAM	≈ (observations)	×	√ (policies)	√	√ (dynamic)	×
AEM	≈ (requirements)	×	×	√	×	√
SAAM	≈ (scenarios)	×	×	√	×	√
AREL	√	√ (BBN)	×	√	×	×
ACL	√ (CCL)	×	×	√	×	×
Tamzalit	×	×	√ (patterns)	×	×	√
Garlan	×	×	√ (paths)	×	×	√
AKdES	√	√ (BBN)	√ (patterns)	√	√ (styles)	√

√, full support; ≈, partial support; ×, not supported.

DOI: 10.1002/smr

Styles are also obviously related to evolution, as they can also constrain it. This can be considered from both sides of the coin. First, a *dynamic* architecture, by definition, also specifies an architectural style. That is, the architecture evolves (describes several different configurations), but at the end of the day, it continues being one and the same entity. Thus, the whole sequence of intermediate configurations has to maintain a set of basic constraints, which should capture the *essence* of the evolving architecture. By definition, this set of constraints specifies an architectural style—as previously noted [40].

On the other hand, we can take the opposite view. Rather than deduce the constraints of the style from the stable part of the architectural evolution, we can *prescribe* the boundaries of that evolution using the notion of style. Therefore, from this point of view, the architectural style is a set of constraints that the system *must* fulfill *at every stage* during its evolution. The system can evolve, in principle, in any way that does not explicitly violate any of the rules described in the style. Indeed, this meaning has been implicitly present from the first definition of the concept [10]—we should just emphasize that the architectural style constrain not only the *current* architecture but also any future version of it.

Thus, the notion of evolution style emerges not exactly as a new concept—but as a specific variant of the generic notion of architectural style; indeed, it is presented that way in the original proposal [38]. Of course, there are differences—notions that we specifically introduce to harness even further the evolution scheme. In particular, the different proposals defining the notion of evolution style provide additional constraints for the evolutionary process. The common idea is *to introduce some kind of restriction about the way in which this evolution may happen*, that is, about the sequence(s) of intermediate steps that can be carried out.

As already mentioned, the existing proposals [9, 38] that define the notion of evolution style have used different ways to constrain this evolution, which are specified in the following:

- Tamzalit's approach, as presented in [38], is conceptually able to use topological, behavioral, or communication-oriented constraints. However, this proposal so far has only dealt with topological rules. These evolution styles have a *syntactic* part and a *semantic* part; but only the first one is relevant to constrain evolution. The syntactic specification consists of a *header*, which defines the evolution task, and an optional *competence*, which provides an implementation method (often deferred to external technologies). The header provides three essential constraints for the style: a *precondition*, a *postcondition* (i.e., a goal), and more importantly, an *invariant*. These conditions are expressed using predicate logic or a comparable formalism (e.g., Object Constraint Language).
- The Garlan *et al*. approach [9] is similar to the preceding one, although it has a more operational flavor. As already noted, it begins with an initial style and has a target style; the way to evolve from the first to the last describes an evolution path, possibly several ones. The emphasis on evolution is clearer in this case, as constraints are imposed to evolution paths, rather than to the base architecture. Therefore, the style still conforms to the original definition [10], but here, it is clearly evolution itself that is being constrained. The operational nature is defined by construction—paths are divided in discrete evolution steps; and their constraints are described using a temporal logic.

Therefore, not only are both proposals similar in structure and scope, they have also been specifically conceived to constrain evolution using just topological information. This is not bad in itself—in fact, in the context of SA, this is both a coherent choice and a usual practice. But, at the same time, it does not exploit all the potential of the concept.

As already noted, the growing body of research on AK makes it possible to obtain and use much more information about the architecture, which is not limited to topological constraints anymore. For instance, this information includes many of the choices made during architectural design (i.e., design decisions); therefore, when the circumstances that led to a certain decision vary, the potential to perform a change in the architecture appears. Obviously, this provides even better opportunities for evolution than just topological information, as it is based in the system's own specific requirements.

Then, our proposal, AKdES, is to use AK in the context of evolution styles, both to constrain and to trigger evolution in SAs. Therefore, in our proposal, evolution styles are similar to those in previous

DOI: 10.1002/smr

proposals (particularly to those of the Garlan *et al*. approach) but explicitly include AK techniques. In fact, this combination is also implicit in the original definition of style [10], considering the role of the rationale. However, it has never been discussed before in this context. Tamzalit's approach has an explicit interest for 'evolution reuse', and it even makes an explicit reference to the exploitation of AK [38]. However, this is just a side reference, and existing work concentrates just on topological constraints.

The central idea of our proposal is summarized in Figure 1. As already noted, it is inspired to some extent by the Garlan *et al*. approach, although many details are different. We can say that our proposal is based on that of Garlan *et al*. as much as their proposal is based on Tamzalit's. This means that there are many commonalities, but the approach itself is quite different, and it is supported by very different technologies.

Figure 1 depicts a concrete *step* in the evolution of an architectural model. Each 'frame' represents *a different moment in time*, building a temporal sequence as indicated by the *time* arrow. The model in every frame describes the current architecture (configuration) and the relevant AK. In this example, the AK is structured; the names of the AK relationships from the decision tree have been taken from [41]. Namely, the first frame (at time t) depicts a configuration with two components (right oval), along with the *decision tree*, which led to this configuration. This tree shows also that a certain decision (A) has influenced our final choice, implying the selection (constrains) of configuration C instead of the alternative B, which is rejected (*inhibits*). The rejected configuration (B) would have led to a three-component architecture (left oval). This is the initial situation.

Then, something happens: an evolution step, as reflected by the central arrow (Δ). In short, that change means that our previous A decision is modified. And then, the new situation appears as a frame in time $t+1$. The configuration has now three components, and the decision tree has been altered. Now a new decision (A') is inhibiting the old choice (still named C), and now the B alternative is selected (our choice is now *constrained* to this specific configuration). Therefore, an evolution step has happened.

In short, our proposal intends to define evolution *styles* in the context of a SAK-based system, similar to Garlan's proposal [9], as sequences of such steps, evolving the system from the configuration in time t to the situation in $t+n$, and defining a consistent and reusable evolution process in between.

3.2. *Defining evolution styles*

Our definition of an evolution style is constructive: rather than providing a direct definition, we start by describing the elements that compose it until we reach a final form. Then, we use a bottom-up, rather than a top-down approach. Our main purpose is *knowledge reuse*, not only in the architecture but even in its evolution—indeed, the main goal of architectural styles and even of knowledge itself.

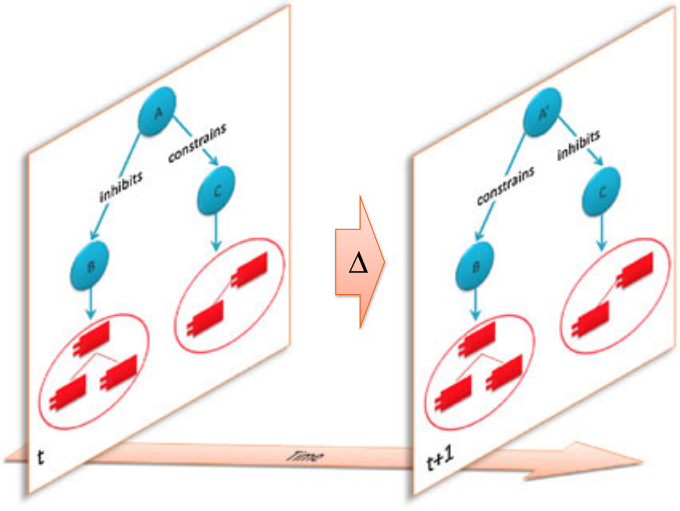

Figure 1. Architectural knowledge to evolve software architecture.

DOI: 10.1002/smr

This section provides an *abstract* definition for an evolution style, and it can be applied to any process that fulfills the requirements summarized in Section 3.6. However, when we choose a specific process, we can obtain a much *concrete* definition. We outline ours in Section 4. Then, in our proposal, an evolution style is composed of these elements:

- *Evolution conditions*: a (potentially) system-wide condition that triggers a change, that is, requiring an evolution step. In short, the identification of a situation that should be handled by evolving the system. Many of these evolution conditions must be understood as changes on the initial assumptions, and therefore, it can affect previously existing decisions (ADDs). Section 3.3 discusses these elements in some more detail.
- *Evolution decisions*: a choice made as reaction to an evolution condition. This choice must be stored as part of the AK, possibly using a different category than other design decisions. Note that this decision will still be related to other architectural decisions, using the usual relationships.
- *Evolutionary steps*: what happens when a decision is chosen—that is, evolution happens. Note that this definition depends on the *decision*, not on the specific architecture. Therefore, a single evolutionary step may imply several technological steps at the architectural level—even a full reconfiguration; but just a single decision is used.
- *Evolution patterns*: a sequence of evolution decisions, possibly chained within a logical reasoning, which could be described as an evolution rationale. A pattern defines a sequence of steps: every decision creates an alternative branch and can lead us to a new evolution decision, until a termination condition is met. The result of the application of such a pattern results in the form of a decision tree within the AK structure. Evolution paths in the Garlan *et al.* approach [9] can be considered as a related construction of these (or a variant case).
- *Evolution styles*: a set of evolution patterns that are conceptually related—that is they affect either the same set of features or a set of related features that, together, could achieve some combined effect. Note that this set of patterns need not to be connected, and of course, it includes the unitary set as a special case; that is, the simplest evolution style consists of a single pattern.

Therefore, the structure of an evolution style in our proposal can be quickly summarized as such: an evolution style is a set of conceptually related evolution patterns. An evolution pattern describes the answer to a specific evolution condition, which triggers a decision process where a different configuration is chosen among several alternatives. Once the alternative configuration has been selected (an evolution decision is taken), the system executes an evolutionary step, modifying the AK structure in the process. If this step satisfies the evolution condition, the process finishes; if not, the evolution may continue.

Figure 2 depicts a representation of an evolution pattern or, more precisely, the *effects* of applying a certain evolution pattern in the AK structure. Our definition is constructive, and the notion of an evolution pattern is inherently dynamic. So, to understand this picture, we should consider it as the final result of applying a pattern consisting of three steps, that is, the situation at time $t+3$.

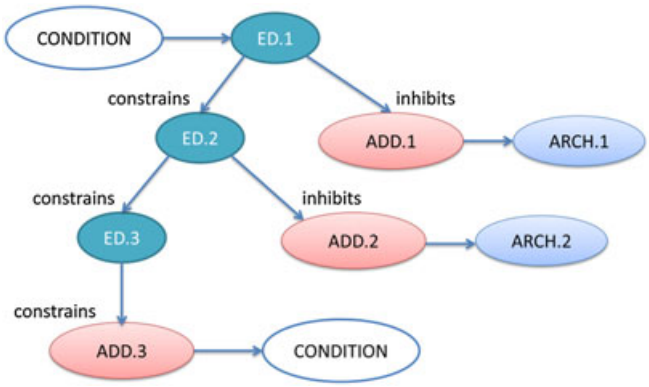

Figure 2. A presentation of an evolution pattern and its effects.

DOI: 10.1002/smr

The process develops as follows. First, an evolution condition (represented by the top-left oval) is detected, and the application of the pattern is triggered. Then, a certain ADD (ADD.1) is chosen, and as a consequence, the architecture evolves from the initial version (not shown in the picture) to a different configuration (ARCH.1). This is the first evolutionary step, and the decision to perform this step is captured in the first evolution decision (ED.1). Therefore, in the first version, ED.1 implies the choice of ADD.1, and hence, they would be initially related by a constrains relationship.

Then, we assume that the evolution condition requires further evolution, and a second evolutionary step must be performed. As a result of this, the previous decision (ED.1) changes, and a new one (ED.2) is made. The change on ED.1 means that now ADD.1 is rejected, and therefore. the relationship between them changes to inhibits (as shown in Figure 2). This step is similar to the presentation made in Figure 1. At the end of the second evolutionary step, the decision ED.2 implies the choice of a new ADD (ADD.2) and of the corresponding configuration (ARCH.2).

The evolution process continues; and a third evolution step is executed. Then, like in the previous case, a new decision (ED.3) is made, and the previous one (ED.2) changes. This is exactly the situation as depicted in Figure 2: now, ED.2 inhibits ADD.2 and constrains (i.e., implies) ED.3 instead. The decision ED.3 implies the design decision ADD.3, which leads to the final architecture (not depicted in Figure 2). Then, the termination condition (bottom-right oval) is met, and the application of the pattern finishes.

Of course, many different patterns may appear; also, many evolution paths can be taken—the emphasis here is that every time a pattern is applied, its effects modify the AK structure. This makes it possible to trace the application of previous patterns and possibly to abstract from these structures, to obtain new patterns from pre-existing sequences of evolutionary steps.

3.3. Evolution conditions: detecting the need for evolution

By integrating AK in the evolution process, we intended to assist the architect in detecting *when* and *how* the architecture should evolve. AK can be used for evolution in three ways:

- AK can help to *detect* when a *change* is allowed.
- AK can help to *detect* when a *change* is required.
- AK can help to determine which evolution style should be chosen to evolve the system, among several potential candidates.

In this section, we focus on the first option. Much of the research on architectural evolution has focused on *how* we can do an architectural reconfiguration, but less emphasis has been put on deciding *when* to do it. The reason is, probably, that much of the relevant information was not explicit (it was ‘unrepresented design knowledge’ [24, 26, 27])—but now, this information is included in the AK.

The decision to evolve is semi-automatic, in the sense that a human has usually the last word. But our reason to trigger the evolutionary step can often be encapsulated in a logical formula, an evolution condition. When this condition is true, then evolution should usually happen.

Evolution conditions are described as a logical expression—or alternatively as plain text—which can gather and combine any number of the following factors:

- Situations in the current system architecture—that is, mostly structural or topological features (e.g., number of connections), as in many other proposals
- Situations in the external context—that is, something that happens on the outside, including human intervention—and therefore unpredictable by its own nature
- *Active decisions*—that is, an architectural decision made in the past and included in the AK, which is still directly affecting the current SA. Questioning if such a decision is still justified considering the current context, and even possibly revoking it, is a standard method for triggering a change to the system.
- *Past decisions*—that is any decision made in the past and included in the AK, which was either *taken* or *rejected*. This past decision can still affect active decisions by means of AK relationships (as seen in Figure 1 and, particularly, in Figure 2) or can be used as a ‘memory log’ to decide upon the current decision, accessed by means of traceability.

DOI: 10.1002/smr

Evolution conditions, described in this way, are able to access more information about an existing system than many other approaches—indeed, they are potentially able to know as much about the system as the architect himself. Therefore, they provide the means to define every single step within our evolution style.

3.4. Evolution decisions: linking to architectural knowledge

As already noted, our approach to software evolution relies on the assumption that the system to evolve has an explicit architectural rationale (i.e., the evolutionary system has also an explicit specification). In the ideal case, this architectural rationale is not just a plain text description (which would be useful nonetheless), but a structured representation that we are able to handle—that is what we have called SAK. Again, neither the language nor the concrete representation is important: the only strong requirement is to be able to access and use this information.

The SAK assumes an explicit representation of AK and that this representation includes a set of internal relationships—i.e., it is structured. The set of relationships may vary with the architectural language or platform. There are several proposals in the literature: to a certain extent, the set of relationships is not as important as the structure (the network) they define [42], which could even constrain our future evolution. Nevertheless, for the remainder of the discussion, we will use the set that we ourselves described in [41] and is also used in Figures 1 and 2.

Essentially, we just need to know that constrains is a direct relationship (decision A implies decision B) and that inhibits is an inverse relationship (decision C hinders but does not forbid decision D). There are other relationships, more prominently *excludes*, but in this article, we just need to refer to these two.

The connection between the evolution process and the SAK is provided by evolution decisions, as defined in Section 3.2 and exemplified in Figure 2. In a sentence, any decision about the system is an ADD, and it deals with the system itself; but a change in the architecture defines an evolution decision, that is, a decision about ADDs. Of course, evolution decisions are not strictly necessary: the AK network can describe the same ADDs without using them—but their purpose is to serve as a link to AK and also to assist in the definition of evolution styles.

Therefore, the style is built bottom-up: the architect, while working with a concrete system, identifies a significant step (i.e., a concrete decision) in evolution, which can be abstracted from the specific AK to a generic situation. Then, he gathers sequences of such steps, and these define patterns. And finally, a set of related patterns defines an evolution style.

As in many other cases, these styles are created out of scattered fragments of knowledge—but they are still built in a way that guarantees conceptual coherence. Evolution styles must be applied to a system—or, more precisely, into the AK of a system. This will be known as the *base system* for the style. As styles are conceived to be generic, they must be adequately adapted and parameterized—but once they are superimposed to specific decisions and components, they can be useful in many different contexts.

3.5. Applying an evolution style

The evolution style is conceived to be applied as part of a semi-automatic process: every step in every pattern corresponds to an evolution decision—and both in architecture and in evolution, those decisions are usually taken by humans [43]. In that sense, any evolution, that is, a change to the AK and therefore to the associated architecture, has to be explicitly approved by the architect.

Evolution styles would be useful even if they were purely documental; however, we must assume some kind of automatic support. For instance, the existing model-driven support for design decisions and architectural styles that is present in the Morpheus toolset [44] can be easily extended to describe this kind of structures and to provide assistance to the human architect. The resulting system would be able to *suggest* a potential evolution to the architect, and he would just need to *accept* it—then, evolution will happen.

The application of an evolution style, with the support of an automatic system, is simple once we have evolution patterns in the form of an AK decision tree—just like the one in Figure 2. First, the system must detect an evolution condition: this can be performed automatically if the condition is a logical expression; when it is provided as plain text, it is the architect who must decide about it. Then, the sequence of evolutionary steps is followed in the original order: the first ED is considered, and the corresponding

DOI: 10.1002/smr

ADD is applied. Hence, the first step is performed. If the evolution condition has been satisfied, the process can stop here; otherwise, the second ED is applied, the previous ADD is inhibited, and a new ADD is taken. This is repeated, step by step, until the termination condition is met. The termination condition can be quite simple: for instance, just that the architectural pattern has finished.

Of course, in the few cases when human intervention is not required, evolution styles might have everything they need to take a decision—hence, they could *automatically* perform the full evolution process without further assistance, using the same model-driven techniques referred to earlier. However, this kind of situation is rare, and it does not describe the general case, so it will not be considered in the following.

3.6. Requirements to apply the proposal

As aforementioned, the proposal has been defined (so far) in an abstract way, so that these ideas can be used in many contexts, no matter the specific process we use for evolution. However, in the previous sections, a number of ideas have been assumed to be able to apply it. Therefore, the process must at least comply with the following requirements:

- *SAK*: the approach consists in exploiting this AK; hence, it must be available. As already noted, our proposal takes advantage of the structure of this knowledge—therefore, SAK must be assumed. The elements of this knowledge must include ADDs and rationales (ADR).
- *Process description*: our process must at least cover development until the architectural design stage; from this and the previous point, we can safely assume that the process provides support and stores information from requirements to the architecture.
- *Degree of automation*: the complexity of this approach makes automation a necessity, rather than an option. As we are dealing with several kinds of models (requirements and architecture), their evolution, and their transformations, we also require a *model-driven development* (MDD)-based approach.

In summary, our *abstract* proposal can be applied to any development process that stores SAK from requirements to architecture and that is able to handle this information and process by means of an MDD-based approach.

4. PROOF OF CONCEPT: ATRIUM FOR EVOLUTION

To validate the approach, we have selected a process that allows us to deal with AK and to manipulate its models in an easy way, ATRIUM [15]. It has been designed for the concurrent definition of requirements and SA, providing automatic/semi-automatic support for traceability throughout its application. In the following sections, we explain why ATRIUM has been chosen, briefly introduce ATRIUM, and present our approach.

4.1. Reasons to choose ATRIUM

As noted in Section 3.6, our abstract definition of architectural styles can be applied to any development process that fulfills three requirements. ATRIUM does comply with these requirements, and therefore, it is able to use the ideas in our proposal. The reasons are specified as follows:

- ATRIUM provides explicit support for AK, as explained in [44]. In fact, it has been extended to provide a very complex structure of AK in a simple way, using just three AK relationships [41]. Therefore, it does *support* SAK and, in fact, has been explicitly chosen for the richness and detail of its SAK support.
- ATRIUM was specifically designed to support the development process *from the requirements to the architectural stage*. The details of this process will be summarized in the next section.
- ATRIUM is an MDD process [45] and, in fact, was designed as such from the start. First, it provides support for the process itself, from requirements to architecture [15]. Second, it also supports the management of AK using MDD-based techniques [44]. Hence, this support can also be extended to support evolution.

DOI: 10.1002/smr

In the rest of the article, and for the sake of clarity, we will assume that we are using the ATRIUM-specific version of our proposal, rather than the most abstract one. However, our conclusions should hold for any other process that fulfills the noted requirements.

4.2. Describing ATRIUM

Figure 3 shows, with the use of SPEM 1.1 [46], the main activities of ATRIUM. These activities must be iterated over to define and refine the different models. These activities are described as follows:

- *Modeling requirements*: This activity allows the architect to identify and specify the requirements of the system to be by using the ATRIUM goal model, which is based on Knowledge Acquisition in Automated Specification [47] and the non-functional requirement framework [48]. This activity uses as input both an informal description of the requirements stated by the stakeholders and the ISO/IEC 25010:2011 Software Product Quality Requirements and Evaluation [49]. The latter is used as a framework of concerns for the system to be. In addition, the architectural style to be applied is selected during this activity.
- *Modeling scenarios*: This activity focuses on the specification of the ATRIUM scenario model, that is, the set of architectural scenarios that describe the system's behavior under certain operationalization decisions [50]. Each ATRIUM scenario identifies the architectural and environmental elements that interact to satisfy specific requirements and their level of responsibility.
- *Synthesize and transform*: This activity has been defined to generate the proto-architecture of the specific system [44]. It synthesizes the architectural elements from the ATRIUM scenario model that builds up the system to be, along with its structure. This proto-architecture is a first draft of the final description of the system that can be refined in a later stage of the software development process This activity has been defined by applying model-to-model transformation techniques [51], specifically, using the QVT-Relations language [52] to define the necessary transformations. It must be pointed out that ATRIUM is independent of the architectural meta-model used to describe the proto-architecture because the architect only has to describe the needed transformations to instantiate the architectural meta-model he deems appropriate. However, the necessary transformation to generate PRISMA architectural models [53] has been already defined.

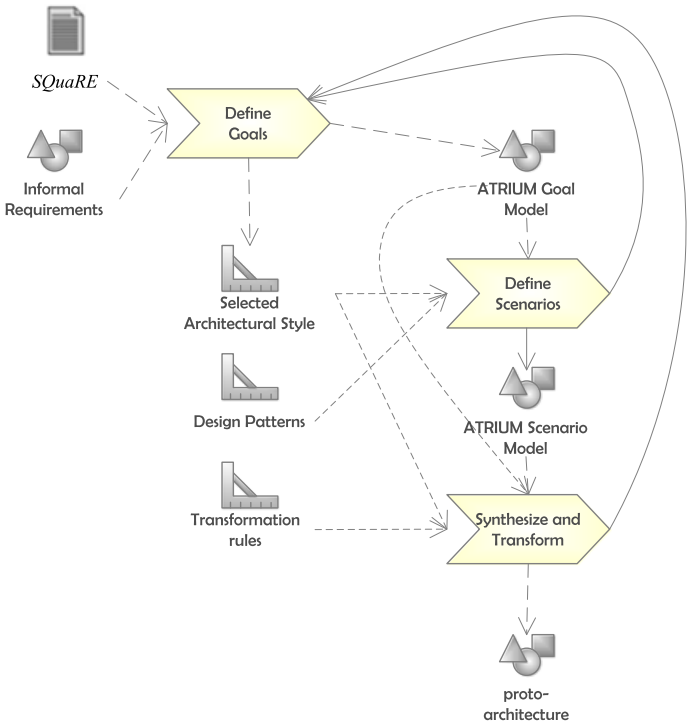

Figure 3. Describing ATRIUM.

DOI: 10.1002/smr

4.3. Architectural knowledge in ATRIUM

As presented in [41, 44], DDs and DRs are introduced from the very beginning of the software development process, specifically, from the requirement stage, thanks to their specification in the ATRIUM goal model. *Goal*, *requirement*, and *operationalization* are the building blocks of this model, as shown in Figure 4. Goals constitute expectations that the system should meet. Requirements are services that the system should provide or constraints on these services. Finally, an operationalization is a description of an architectural solution, that is, an architectural *design choice* for the system to be to meet the users' needs and expectations. They are called operationalizations because they describe the system behavior to meet the requirements, both functional and non-functional. For this reason, two key attributes are included while they are described: DD and DR. A seamless transition is performed from requirements to operationalizations by means of the *contribution* relationship, to specify an *operationalization* contributing to/preventing the satisfaction of a *requirement* facilitating the automatic analysis of architectural alternatives [15].

One of the main advantages of AK management is the capability to explore the reasoning in the SA by exploiting the network of AK. To provide ATRIUM with this facility, several relationships were defined in its meta-model to allow the analyst to describe the AK as a network [41]. As shown in Figure 4, these relationships were first specified on operationalizations, as they are in charge of describing both the DDs and the DRs and can be defined as follows:

- *Constrains* is a binary and unidirectional relationship with positive semantics. Let us consider A and B operationalizations, describing different design decisions. Having a *constraint* relationship from A to B means that B's design decision cannot be made unless A's design decision is also made.
- *Inhibits* is a binary and unidirectional relationship used to specify negative semantics. Let us consider A and B operationalizations, describing different design decisions. Having an *inhibition* relationship from A to B means that if A's design decision is made, it hinders the making of B's design decision.
- *Excludes* is a binary and unidirectional relationship with stronger negative semantics than inhibits. Let us consider A and B operationalizations, describing different design decisions. Having an *exclusion* relationship from A to B means that if A's design decision is made, it prevents B's design decision from being made.

As presented in [44], one of the advantages of ATRIUM is that it facilitates the generation of the DDs along with the proto-architecture, so that each architectural element is related to the set of DDs that motivated its specification and the DRs that justify those decisions. Figure 4 shows (part of) the

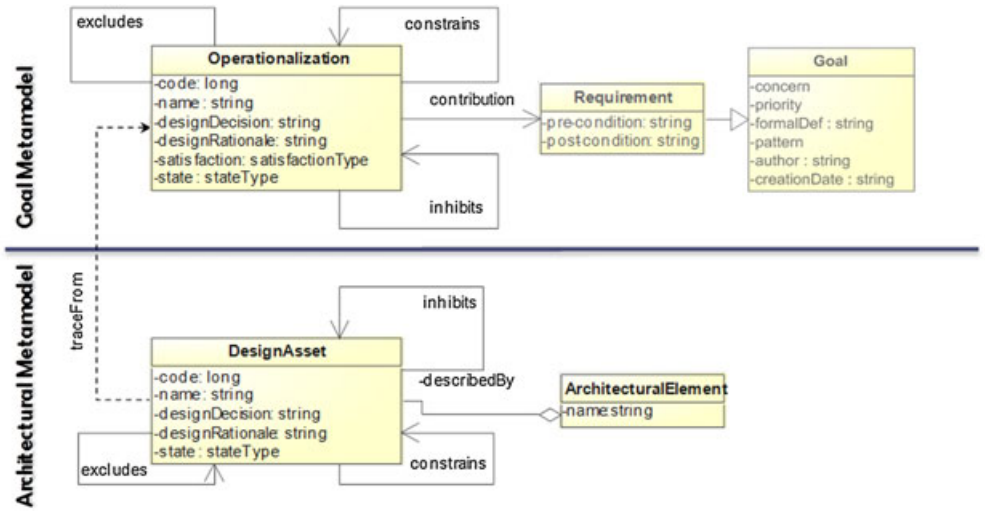

Figure 4. Describing architectural knowledge in ATRIUM.

DOI: 10.1002/smr

architectural meta-model. It can be observed that every ArchitecturalElement is related to a set of DesignAssets that describe both its DDs and DRs. As can be observed in Figure 4, the DesignAssets can be related by means of constrains, excludes, and inhibits relationships in a similar way to the operationalizations in the goal meta-model.

It is worth noting that the main difference is between operationalizations in the goal model and DesignAssets in the architectural model. The former are in charge of specifying all the DDs and DRs that were analyzed during the specification of the system, that is, they describe the reasoning carried out to evaluate which were the best alternatives for the system. The latter describe the reasoning behind the current specification of the system, that is, why the system has its current specification. Therefore, they help to maintain AK from different perspectives.

4.4. Describing evolution styles in ATRIUM

Considering the constructive definition of evolution styles, as provided in Section 3.2, it is quite clear that most of the relevant notions are already present in ATRIUM, and hence, they do not require any extension of the standard framework. Therefore, they are described as follows:

- *Evolution conditions*: The condition that triggers an evolution process. It can be described as plain text or using a logical formula. Particularly in the second case, the automatic support in ATRIUM can be extended to provide some automatic detection of the condition; but most of the time, this would be conceived and decided by the architect himself (i.e., by human intervention). In summary, there is no need to explicitly describe these conditions as part of the model.
- *Evolution decisions*: These provide the decision to perform an evolutionary step. They are a special case of design decisions, as they are *decisions on decisions*. Just like conventional ADDs describe information and choices about architectural elements, these evolution decisions describe choices about ADDs themselves. As already noted in Section 3.4, they are provided to serve as a link between the AK and the evolution process—but they are not strictly necessary in terms of the AK. However, their role is very important in describing an evolution style, as they provide the basic skeleton for this structure—in the form of a decision tree. Therefore, ATRIUM provides explicit support for these elements, in the form of the EvolutionAsset entity. This is defined in the architectural meta-model, as shown in Figure 5, as a special case of DesignAssets.
- *Evolutionary step*: It describes an action, that is, the evolution from a situation to another situation. Hence, it does not require any explicit representation. However, every such step leaves a definite trace in the structure of the AK. Just consider the abstract process as defined in Section 3.2 and seen in Figure 2. Essentially, for every step, a decision is made (captured as an EvolutionAsset). This decision affects a certain ADD (captured as a DesignAsset, which in turn relates to several ArchitecturalElements) and depicts this influence using a constrains relationship. The sign of this relationship can eventually be modified (as already seen) by further decisions.
- *Evolution pattern*: As a pattern is a sequence of evolutionary steps, it is not necessary to provide any additional concept to describe this notion. In fact, provided that every step is captured by the

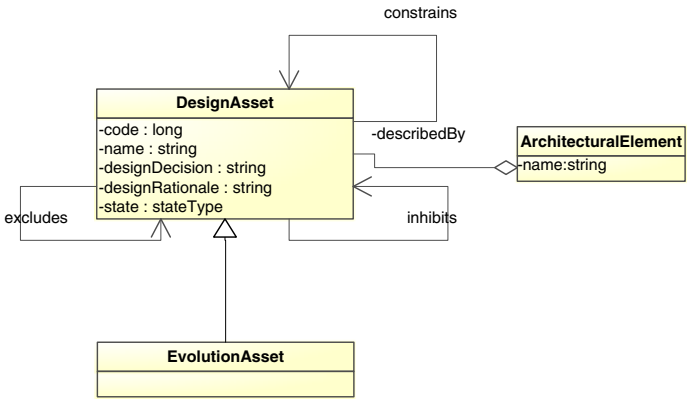

Figure 5. Defining EvolutionAssets in the architectural meta-model.

DOI: 10.1002/smr

three elements construction mentioned in the previous point (an EvolutionAsset, a DesignAsset, and the relationship between them), a pattern is shown as the sequence of such triplets—where every evolution decision is expanded as an additional branch, therefore taking the form of a decision tree. Again, the structure provided in Figure 2 depicts such a decision tree, and there is no need to provide any special construct to capture it—apart from giving it a name so it can be reused.

- *Evolution style*: Similarly, an evolution style is just a set of evolution patterns—therefore, it does not require any special construction either.

In summary, once the EvolutionAsset is provided, it suffices to be able to use already existing concepts (in particular DesignAssets and their relationships) to constructively build up a reusable representation for an evolution style—once it is parameterized—to abstract this definition from the concrete elements it affected in its first occurrence. The only special requirement of evolution patterns and styles is a distinctive name and a mapping (to apply the generic pattern to specific elements)—the rest is already provided.

Figure 5 shows how EvolutionAsset is described in the architectural meta-model. As it is defined as an extension, it inherits not only all its attributes, helping to define decisions and rationales about other DesignAssets, but also its relationships, which are used to establish how it affects other DesignAssets.

Moreover, although EvolutionAsset has been defined only at the architectural meta-model, it could be also defined in the goal meta-model. Thus, the architect could exploit model-to-model transformations to generate these elements in the goal model in an automatic way. This alternative could be helpful to carry out the evolution by taking into account both the AK and the requirements of the system. This alternative would be of interest to describe evolution conditions in the goal model, although the implications of its use are currently under evaluation.

The application of these evolutionary steps, in ATRIUM, follows the same lines as the generic definition provided in Section 3.2. First, an evolution condition is detected, and then an evolution decision (documented as an EvolutionAsset) is made; this evolution causes a concrete choice in the architecture, which is captured as a DD within a DesignAsset, and provides the corresponding relationship. The sequence defined by this process defines a decision tree by combining such triplets, therefore creating the equivalent of a structure that can be traversed and reused.

5. EXAMPLE: EVOLVING A CLOUD ARCHITECTURE

To illustrate the concepts introduced through the previous sections, in this section, we present a practical case for an evolving architecture—and how the management of AK leads to this evolution and to the definition of evolution styles.

Instead of a trivial example, we present a real-world case study, including a complex architecture with a certain set of features and that faces a complex problem. The purpose is twofold: first, to describe a problem in an interesting context, showing that our approach is not a just a ‘lab construct’ and can still be applied within a non-controlled environment; and second, to show the actual power of these concepts, which is not perceived until applied to a complex problem.

Another interesting feature of this example is the reason to evolve, which is *increasing costs*. Therefore, the change is not required for some technical reason (and the system is complex enough to have plenty of these), but to fulfill a real-world necessity. This is relevant because this is the kind of situations that can only be adequately described when the AK is made explicit.

5.1. Initial situation: the cloud-based radio station system

The example we present in this article has been developed in the context of *cloud computing* [54]. The reasons behind this choice are the current emphasis on this approach, which helps to situate the practicality of our proposal and also the fact that *cloud architectures* are of great importance for cloud-based applications—that is, the architectural level is particularly significant for its function, and therefore, its evolution is relevant for the system as a whole.

DOI: 10.1002/smr

Perhaps the most important feature of cloud computing, and undoubtedly what distinguishes it from other related proposals, such as software as a service (SaaS), is *scalability* (also known as *elasticity*). A cloud-enabled application is executed within an *elastic* environment, which means that the application is able to automatically react to an increasing demand of resources—when they are necessary, simply there are more resources available.

Without loss of generality, let us conceive a cloud-enabled application as a set of independent *services* related to each other by means of *queues* and managed by specific *controllers*. This setup is sometimes referred to as 'the canonical cloud architecture' [55]. Most of the management policies have to be either distributed in the architecture or managed within those controllers.

Our example describes a cloud-based, on-demand 'radio station' that broadcasts on the Internet by using *streaming* techniques. This station has stored a large set of programs, which are uniquely identified. When a listener (a user) tunes into the station, he requests for some specific program; the system returns by providing the URI of a streaming server, where the user can now listen to the requested program. The process ends when the broadcasting finishes and every used resource is set free.

As depicted in Figure 6, the system is composed by three kinds of services: a *database service* (*DBS*), a *streaming service* (*SS*), and a *data storage service* (*DSS*). These services are managed by three controllers, respectively known as the *tuner*, the *monitor*, and the *terminator*. Each one of them has its own *queue* to receive and store requests.

Every time that a listener enters the station, he searches for a certain program—that is, he triggers a request on the tuner's queue. The tuner locates the requested program in the database (DBS), and then the monitor creates a new instance of the streaming service (SS). This SS obtains the recording of the program from the storage (DSS) and starts to stream its contents on a newly created URI. Once the program has finished, the monitor notifies the terminator to destroy the old SS, releasing the associated resources.

Of course, the actual system is more complicated; this paper simplifies the presentation to concentrate on the issues related to evolution, which are our main concern here.

5.2. *Brief discussion on the example*

These services are conceived as software services 'in the cloud' within an archetypical cloud platform. This means that all these services (our DBS, SS, and DSS) are implemented and exported using the SaaS approach, where users access them as clients of a service. However, as already noted, using a service-oriented approach is not enough. In fact, these services are usually designed almost in the same way as conventional applications: they are *cloud* applications mostly because of *where* they are deployed—they are scalable and resilient because this is supported by the underlying platform. But this platform is also service oriented, and it is presented in the form of an 'infrastructure as a service' (IaaS). In summary, our application is defined as a set of user-level (SaaS) services, which are in turn supported by a set of system-level (IaaS) services.

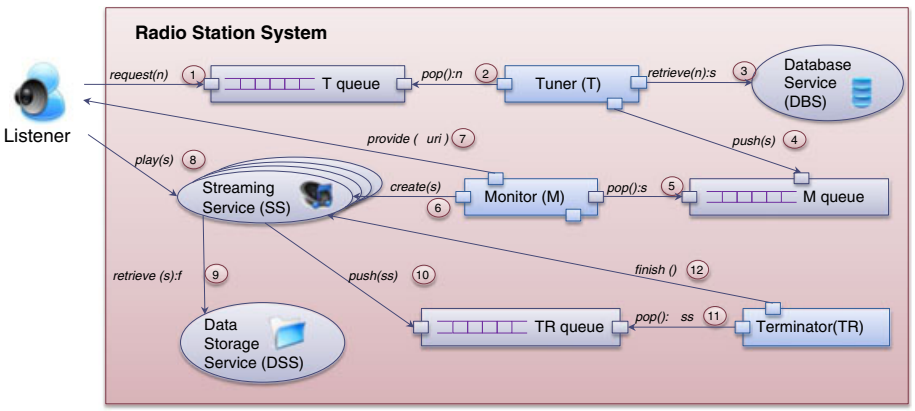

Figure 6. Radio station system at runtime: version 1.0.

DOI: 10.1002/smr

Both the 'canonical cloud architecture' and our specific example, as presented earlier and in Figure 2, are 'mixing' both kinds of services, so the architecture might *seem* a bit complex. Of course, they are *just* services: our architecture is just a set of interacting services—and having a well-defined workflow, which is the case, that makes it quite simple.

For the sake of clarity, a short explanation is introduced in the following. Our *user-level services* (DBS, SS, and DSS) are SaaS services; every one of them runs over a lower-level IaaS service. Our controllers (tuner, monitor, and terminator) are effectively controlling these underlying services and how they provide resources to the top-level ones: thus, they have to be considered IaaS themselves, but their function defines our application's scalability. Finally, the different queues (T, M, and TR) are services themselves—in fact, instances of the same service. Their existence is almost mandatory for cloud architectures: elasticity implies that the number of clients of a service, at a given moment, can exceed the capability of any service. Queues are therefore provided to ensure that no request (or response) becomes lost. Queuing services work effectively as the 'connectors' in this architecture, and obviously, it is safe to classify them as IaaS, as they serve as the basic infrastructure.

Therefore, at the infrastructure (IaaS) level, we just need a storage service, a computing service, a generic database service, and a queuing service, with their corresponding controllers, to respectively allocate our DSS, many SS instances, the DBS, and all the queues at the user level. When the popular Amazon AWS platform is used, for instance, they would have been managed by S3 storage, EC2 computing, the SimpleDB service, and SQS queues [55]—our SaaS services would run on top of these.

Of course, there might be more elements—for example, the *billing* subsystem has been deliberately omitted. However, it must be present in any cloud-based application: every service in the cloud *costs money*. Of course, we would assume its function, that is, the customer still is charged, but it has not been included in the architecture to simplify the presentation.

5.3. Detecting the need: excessive dynamic allocation

Let us suppose that the presented system is satisfactory in terms of efficiency and functionality: the radio station works as expected, and the user experience is positive. The chosen program is broadcast, and performance is right, even during occasional (and sudden) peaks of audience.

However, after some time, it is clear that the system is too expensive: elasticity *costs* money. A new listener implies an access to the DBS, a new computational instance of the SS, and one or several loadings from the DSS. Each one of these steps is chargeable and gets included in the bill. But this also means that if the station achieves success, it will have many listeners, who will cause also many expenses. Usually, our income is expected to cover these expenses, but sometimes, this is not the case—for many reasons. Therefore, we are in a curious situation. From a technical point of view, the system can grow as much as desired: the elastic environment guarantees that there are no scalability issues. But the growth rate can still be a problem, in this case, from the *business* point of view.

This way of working, however, is considered standard in current cloud architectures, probably because of the low prices of current cloud providers. However, it is obvious that it is not very efficient: *every time* a new listener accesses the station, a new instance of the SS must be created. In terms of functionality, that is the perfect solution, but it is obviously a waste of resources. In fact, this argument is also found in the well-known performance anti-pattern, *excessive dynamic allocation* [56]. This anti-pattern criticizes the practice of creating a new instance of an object or server to provide a service to a new client—even comparing this (in the best known metaphor for the pattern) with creating a new gas pump in a gas station every time that a new car requests service. The anti-pattern was originally in the context of Web systems and therefore has a number of similarities to our situation.

This need for the system to evolve can only be detected by a human—but examining the AK structure is also of great help. Sometimes, this process can even be partially supported by automatic tools. For instance, part of our previous work in the Morpheus toolkit [57] adds the capability to assist in the detection of anti-patterns (like the one mentioned earlier) by checking the network of relationships in the AK. In terms of our proposal, this means to decide upon the *evolution condition*. As we noted, this condition can be expressed in logical terms or using plain text. In our example,

DOI: 10.1002/smr

it is easy to quantify that condition (i.e., the cost rises over a certain limit), and it is also simple to relate this rise of the costs to a performance problem. Therefore, to explore performance anti-patterns, such as excessive dynamic allocation, should be a logical choice on the part of the architect.

Once the architect has detected the evolution condition, an evolution process begins. Then, an evolutionary step is planned. The first time, this step and the associated decision will be performed manually; but it will be also captured as part of an *architectural pattern*—and ultimately, it will be available for future reuse as part of a larger evolution style, as described in the next section.

5.4. Describing the solution: defining an evolution style

In this subsection, we will describe the *reasoning* we use to describe the evolution of the architecture. This rationale is going to be explicitly captured as part of the AK, including the evolutionary steps—and therefore, it will be used to *build* an evolution pattern as defined in Section 3.2.

To simplify the presentation, we have chosen the same graphical format to that in Figure 2. In fact, they are very similar; but the reader must have in mind that this version describes actual changes in the cloud architecture, whereas the previous one just provided generic terms for any decision.

First, the evolution condition, requiring the architecture to evolve, is detected (Section 5.3). The first time this condition is evaluated, it leads to an evolution decision, that is, the EvolutionAsset EA.1 in Figure 7. The architect considers, as a first alternative, to use the queue and serve the requests with a first come, first serve policy—so listeners have to wait to be served. This choice defines an alternate branch, that is, a new DesignAsset DA.1, and it is captured within the AK structure as a standard ADD. This ADD provides a new configuration (AA.1), which implements the request queues.

Therefore, the decision EA.1 causes an evolutionary step, where the architecture evolves from the initial version described in Section 5.1 to the alternative configuration AA.1. However, it is soon obvious that this solution is not satisfactory—in terms of the evolution condition, it is possible that costs have diminished; but an excessive response time would cause the loss of listeners.

And then, the evolution process continues. A new evolution asset (EA.2) must be taken, and the first consequence is that ED.1 changes, now rejecting (i.e., inhibits) the previous choice, AB.1.

The new decision EA.2 leads the architect to consider a different alternative: the corresponding ADD will be labeled as DA.2. Now, the architecture (AA.2) to be considered uses a *bounded number of instances* of the SS service. It provides a fixed cost, apparently fixing our problem, but it has the negative outcome of limiting the number of simultaneous listeners—which causes, in the end, the same problems that the AB.1 decision already had. Therefore, this second evolutionary steps does not provide the final solution.

The architect considers now a different approach (EA.3). Many users would be interested exactly in the same program, and even at the same time: therefore, instead of creating a specific server every time, we consider *sharing the same server instance* to serve all these listeners. This alternative (DA.3) is therefore our next choice—implying that the previous one (DA.2) becomes inhibited and the

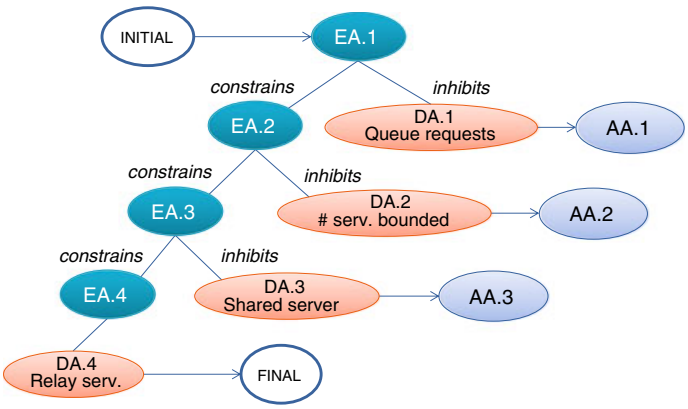

Figure 7. Final decision tree for the application of the evolution style.

DOI: 10.1002/smr

evolutionary step towards a new architecture (AA.3). However, the shared instance would have a limited capability, as we are not using the full power of the elastic environment anymore. Then, we would only be able to serve a limited number of listeners, and the situation becomes similar to what happened in DA.2. Therefore, this solution is also rejected, and the evolution process continues. However, this time we are already close to a satisfactory solution.

Finally, the architect considers a related solution (EA.4): to use a *relay system*. It is still true, as considered earlier, that many listeners would like to access the same program. Then, instead of connecting all of them to the same server, we provide *additional* servers. For this, we need a fourth kind of service, which was not present in the original architecture, the *relay service* (RS). This service simply connects to the live stream of some existing SS and starts broadcasting exactly the same contents it is receiving, using its own URI. An RS can safely serve a certain number of clients, and therefore, the workload is distributed among several instances of these services. Therefore, this leads us to a new design decision, in which the selected branch (DA.4) relies on the relay approach to fix the problem. As we will see later, this defines the *final* step in this specific evolution, ultimately leading to the final architecture (Figure 8).

This process is summarized in Figure 7. As we already indicated in Section 3.2, this figure does not depict the actual evolution pattern—instead of that, it describes the structure of the AK (i.e., the decision tree) resulting from its application. But it is clear enough to serve also as a representation of the evolution pattern we have just defined. This can be also an evolution style, if this is the only pattern considered; in fact, this is just a matter of scale.

We must take into account that it does not actually matter if these evolutionary steps are actually performed, providing changes in the architecture, or if they are just considered as (detailed) alternatives during the system's design. In both cases, they must be included as part of our AK; even rejected solutions are important, as they need not to be considered again. This is exactly what 'unrepresented design knowledge' meant, and it is the main reason for capturing AK.

In summary, the process begins with a single evolution decision (EA.1), which leads to other decisions (EA.2–4), and every time suggests an alternative solution (DA.1–3). This defines a logical chain of decisions leading from the initial question (EA.1) to the final choice (EA.4). Every decision defines an evolutionary step: every alternate branch would propose a different architecture (AA.1–3), so if a different decision had been made, the result would have been quite different. But following our logical chain of decisions (all of them related), we end up accepting the final solution (DA.4), which is the result of a sequence of evolutionary steps: and this is exactly our definition of an evolution pattern.

An evolution style, as we already defined (3.2) will be a set of such evolution patterns, which would have a common goal. The particular case of a style with a single pattern is also allowed, so we can also consider this example as a full evolution style, defining the change from Figure 7 to Figure 8.

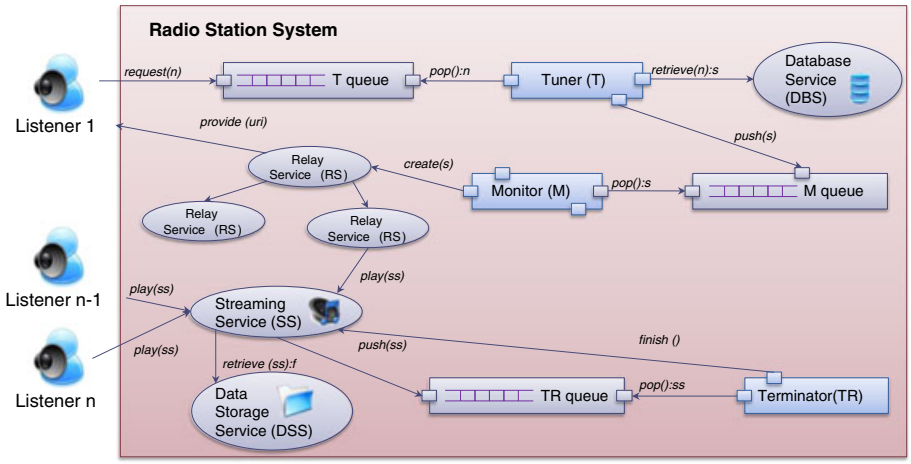

Figure 8. Radio station system at runtime: version 2.0.

DOI: 10.1002/smr

5.5. Final situation: result of applying the evolution style

Although just tangentially related, two important comments must be made about the chosen solution: first, about the way in which this single choice ('to use a relay service') can affect the whole architecture; and second, to explain why this is a sensible solution.

First, note that we need just one SS instance, the original; the first RS just needs to access the stream provided by this instance and begins streaming itself. As more instances of the same program are requested, another instance of the RS could be necessary. This second RS does not need to connect to the original SS, but to the first RS (Figure 8). Thus, the relay system creates a *chain of relays*, each one able to serve several clients and also the next relay.

So, we have the following: first, the original architecture (Figure 8) has evolved from a client–server style to a hybrid architecture, which includes a peer-to-peer chain of relays (Figure 8); and second, the workflow has to be adapted so that the monitor creates a new RS instance when a new listener needs it and provides the listener with the corresponding URI.

This solution is not an obvious solution: the system still creates many server instances. And the RS also consumes resources, so it also costs money. However, this solution is cheaper: first, fewer instances are created; and second, every instance is also cheaper. The RS is probably less complex than the SS; but even more importantly, the RS *need not* use the data storage at all, as it obtains data from the other service. Therefore, it is purely *computational*.

5.6. The resulting evolution style in ATRIUM

As already noted in Section 4.4, there are not many special constructs to describe the evolution style we have just described in ATRIUM, once the notion of evolution decision has been provided. Of course, the basic part of ATRIUM stays unaltered; therefore, we have the standard elements to describe the requirements (a goal model), the architecture (a component model), and the AK. The model of the latter includes all the relevant elements as defined in the ATRIUM meta-model (and as already exposed in Section 4.4), independent of their relationship to the evolution process. Therefore, every design decision will be captured in a DesignAsset including both the decision and its rationale, directly related to the implementation provided as an aggregate of (one or many) ArchitecturalElements; and to the corresponding requirements in the goal model, as traced through the corresponding operationalization.

Therefore, considering that the rest of the architecture is already provided, for our specific example in Figure 7, the extension in the AK structure caused by our evolution style would include the following:

- Four DesignAssets, one to describe every alternative branch (DA.1–4), apart from those that already existed previously.
- In the minimal case, at least four ArchitecturalElements to describe every alternative configuration (AA.1 to AA.3 and also the final architecture). In fact, every architecture has many of these elements, and several of them (as those describing the DBS, the DSS, and so on) are similar or even identical in the different versions.
- The four EvolutionAssets, one to describe every evolution decision (EA.1–4).
- And of course, the corresponding relationships: three inhibits, from each EA.1–3 to DA.1–3, and four constrains, from every EA to the next one, and from EA.4 to DA.4.

Note that applying this evolution style, once it has been stored, does not require any special provision on the part of ATRIUM—it just has to be able to traverse the decision tree and to generate the corresponding version of the architecture once the relevant operationalization has been selected. Both capabilities are already present in the current version of the Morpheus toolkit; the only issue is that these operations must be applied dynamically, during the system's runtime.

Please note that ATRIUM covers several stages in the specification phase, from initial requirements to the proto-architecture [15]. Hence, the process itself has little to do with many well-known issues of dynamic systems (such as state transfer), as they would be completely dependent on the implementation technology. For instance, when ATRIUM is used to generate PRISMA components, it would be the PRISMA technology that is in charge of performing these actions.

DOI: 10.1002/smr

In summary, the example shows how our management and use of the AK make it possible to decide how to evolve from the original architecture to the final approach—and in the process, how to define an evolution style to reuse this knowledge.

6. CONCLUSIONS AND FURTHER WORK

The SA is a critical driver in the development and evolution of software systems. Further, we claim that architecture knowledge is an equally critical driver. We offer two main conclusions: first, that AK can be considered itself as an evolution driver, in the sense that it provides much information of particular relevance to the evolution process; and second, that much of the evolution process itself can be captured as part of the AK, using evolution concepts at the architectural level.

To simplify the reuse of this knowledge, we also propose a specific structure that captures the information and decisions related to architectural evolution: the evolution style. We describe the structure of this style and the process to define and reuse it. To illustrate the feasibility and usefulness of this approach, we describe in some detail the use of this evolution style in the context of a cloud-based application example and provide a clear perspective of the involved notions.

Future work in this context includes the full integration of this (generic) proposal into a specific approach. At present, we are already including these notions in ATRIUM [57], a process that defines and manages SAs from the requirements phase. Incorporating evolution styles into ATRIUM (and its associated toolset) will be just a matter of extending the AK network to include the new kinds of relationships (those related to evolution) and to provide an additional cycle in its model-driven development process, which would apply evolution styles on top of the current architecture and its associated knowledge.

Further work is related to the evaluation of AKdES. Although we have already examined their practicality thanks to the use of ATRIUM, we are in the process of evaluating it from an empirical point of view. In this context, we are assessing existing proposals, such as DESMET [58], that guides us in the evaluation of AKdES. However, the major concern at this point is that there is no standard method that AKdES can be compared with. This drawback turns this evaluation into a very challenging future work.

ACKNOWLEDGMENTS

This work has been partially supported by grants (PEII09-0054-9581) from the Junta de Comunidades de Castilla-La Mancha and also by research grants (TIN2008-06596-C02-01, TIN2012-31104 and CSD2007-022 by Program CONSOLIDER INGENIO 2010) from the Spanish Ministry of Science and Innovation. Professor Perry is supported in part by NSF CISE grants IIS-0438967 and CCF-0820251.

REFERENCES

1. Buckley J, Mens T, Zenger M, Rashid A, Kniesel G. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice* 2005; **17**(5):309–332.
2. Madhavji NH, Fernández-Ramil J, Perry DE (eds.). *Software Evolution and Feedback: Theory and Practice*. John Wiley: Hoboken, NJ, 2006.
3. Bersoff EH, Henderson VD, Siegel SG. *Software Configuration Management. An Investment in Product Integrity*. Addison-Wesley: New York, 1980.
4. Lehman MM, Belady LA. *Program Evolution*. Academic Press: New York, 1985.
5. Lehman MM, Fernández-Ramil JC. Software evolution. In *Software Evolution and Feedback: Theory and Practice*, Madhavji NH, Fernández-Ramil JC, Perry DE (eds.). Wiley: Hoboken, 2006; 7–40.
6. Lehman MM, Ramil JF, Wernick P, Perry DE, Turski WM. Metrics and Laws of Software Evolution—The Nineties View. *Fourth International Software Metrics Symposium METRICS'97*, 1997; 20.
7. Cook S, Harrison R, Lehman MM, Wernick P. Evolution in software systems: foundations of the SPE classification scheme. *Journal of Software Maintenance and Evolution: Research and Practice* 2006; **18**(1):1–35.
8. Holt R. Software architecture as a shared mental model. *ASERC Workshop Software Architecture*, 2001.
9. Garlan D, Barnes JM, Schmerl B, Celiku O. Evolution styles: foundations and tool support for software architecture evolution. *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, 2009; 131–140.
10. Perry DE, Wolf AL. Foundations for the study of software architecture. *ACM Software Engineering Notes* 1992; **17**(4):40–52.

DOI: 10.1002/smr

11. Bosch J. Software architecture: the next step. *1st European Workshop in Software Architecture* (EWSA'04), 2004; 194–199.
12. Gorton I, Babar A. Architecture knowledge management: concepts, technologies, challenges. *Working IEEE/IFIP Conference on Software Architecture* (WICSA'07), 2007; 31–31.
13. Tyree J, Akerman A. Architecture decisions: demystifying architecture. *IEEE Software* 2005; **22**(2):19–27.
14. Bratthall L, Johansson E, Regnell B. Is a design rationale vital when predicting change impact? A controlled experiment on software. *2nd International Conference on Product Focused Software Process Improvement* (PROFES 2000), 2000; 126–139.
15. Montero F, Navarro E. *ATRIUM: Software Architecture Driven by Requirements*. IEEE: New York, 2009; 230–239.
16. Parnas DL. Software aging. *16th International Conference on Software Engineering*, 1994; 279–287.
17. Bass L, Clements P, Kazman R. *Software Architecture in Practice* (2nd edn), Addison-Wesley: Boston, 2001.
18. Breivold HP, Crnkovic I. Analysis of software evolvability in quality models. *2009 35th Euromicro Conference on Software Engineering and Advanced Applications*, 2009; 279–282.
19. Noppen J, Tamzalit D. ETAK: Tailoring Architectural Evolution by (re-)using Architectural Knowledge. *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge—SHARK '10*, 2010; 21–28.
20. Georgas JC, Taylor RN. Towards a knowledge-based approach to architectural adaptation management. *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems—WOSS '04*, 2004; 59–63.
21. Tarvainen P. Adaptability Evaluation of Software Architectures; A Case Study. IEEE: New York, 2007; 579–586.
22. Kazman R, Bass L, Abowd G, Webb M. *SAAM: A Method for Analyzing the Properties of Software Architectures*. IEEE Computer Society Press: Washington, DC, 1994; 81–90.
23. Avgeriou P, Lago P, Grisham P, Perry DE. Sharing and reusing architectural knowledge—architecture, rationale, and design intent. *Sharing and Reusing Architectural Knowledge, Architecture Rationale and Design Intent* (collocated to ICSE), 2007; 109–110.
24. Jansen A, Bosch J, Avgeriou P. Documenting after the fact: recovering architectural design decisions. *Journal of Systems and Software* 2008; **81**(4)536–557.
25. Tang A, Jin Y, Han J. A rationale-based architecture model for design traceability and reasoning. *Journal of Systems and Software* 2007; **80**(6):918–934.
26. Kruchten P, Lago P, van Vliet H. Building up and reasoning about architectural knowledge. *2nd International Conference on Quality of Software Architectures* (QoSA'06), 2006; **4214**:43–58.
27. Tang A, Avgeriou P, Jansen A, Capilla R, Muhammad AB. A comparative study of architecture knowledge management tools. *Journal of Systems and Software* 2010; **83**(3):352–370.
28. Henttonen K, Matinlassi M. *Open Source Based Tools for Sharing and Reuse of Software Architectural Knowledge*. IEEE: New York, 2009; 41–50.
29. Burge JE, Carroll JM, McCall R, Mistrík I. *Rationale-Based Software Engineering*. Springer: Heidelberg, 2008; 316.
30. Dueñas JC, Capilla R. The decision view of software architecture. *2nd European Workshop Software Architecture* (EWSA 05), 2005; 222–230.
31. Capilla R, Nava F, Carrillo C. Effort estimation in capturing architectural knowledge. *23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008; 208–217.
32. Tang A, Nicholson AE, Jin Y, Han J. Using Bayesian belief networks for change impact analysis in architecture design. *Journal of Systems and Software* 2007; **80**(1):127–148.
33. Pearl J. *Causality: Models, Reasoning, and Inference*. Cambridge University Press: Cambridge, MA, 2000.
34. Tibermacine C, Fleurquin R, Sadou S. A family of languages for architecture constraint specification. *Journal of Systems and Software* 2010; **83**(5):815–831.
35. OMG. MOF. http://www.omg.org/mof/. 2011. [Online]. (Available from: http://www.omg.org/mof/). [Accessed: 11-Apr-2011].
36. Garlan D, Monroe RT, Wile D. Acme: architectural description of component-based systems. In *Foundations of Component-Based Systems*, Leavens GT, Sitaraman M (eds.). Cambridge University Press: Cambridge, MA, 2000; 47–67.
37. Watson R, Bhattacharya S, Perry DE. Statically defined dynamic architecture evolution. *1st International Workshop on Automated Configuration and Tailoring of Applications* (ACOTA 2010) collocated with ASE 2010, 2010.
38. Le Goaer O, Tamzalit D, Oussalah MC, Seriai A-D. Evolution styles to the rescue of architectural evolution knowledge. *Proceedings of the 3rd International workshop on Sharing and Reusing Architectural Knowledge* (SHARK'08), 2008; 31–36.
39. Tamzalit D, Oussalah MC, Le Goaer O, Seriai A-D. Updating software architectures: a style-based approach. *International Conference on Software Engineering Research and Practice*, 2006; 313–318.
40. Perry DE. Generic architecture descriptions for product lines. *Software Architectures for Product Families* (ARES 1998), 1998.
41. Navarro E, Cuesta CE, Perry DE. Weaving a network of architectural knowledge. *Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, 2009; 241–244.
42. Rosso CD. Comprehend and analyze knowledge networks to improve software evolution. *Journal of Software Maintenance and Evolution: Research and Practice* 2009; **21**(3):189–215.
43. Mittermeir RT. Facets of software evolution. In *Software Evolution and Feedback: Theory and Practice*, Madhavji NH, Fernández-Ramil J, Perry DE (eds.). Wiley: Hoboken, 2006; 71–94.
44. Navarro E, Cuesta CE. Automating the trace of architectural design decisions and rationales using a MDD approach. *Second European Conference on Software Architecture* (ECSA 2008), 2008; **5292**:114–130.

DOI: 10.1002/smr

45. Selic B. The pragmatics of model-driven development. *IEEE Software* 2003; **20**(5):19–25.
46. OMG. Software process engineering metamodel (SPEM), version 1.1 formal/05-01-06. 2005.
47. Dardenne A Goal-directed requirements acquisition. *Science of Computer Programming* 1993; **20**(1–2):3–50.
48. Chung L, Nixon BA, Yu E, Mylopoulos J. *Non-functional Requirements in Software Engineering*. Kluwer Academic Publishing: Boston, 2000.
49. ISO/IEC. ISO/IEC 25010:2011—software product quality requirements and evaluation (SQuaRE)—system and software quality models. 2011.
50. Navarro E, Letelier P, Ramos I. Requirements and scenarios: running aspect-oriented software architectures. *Working IEEE/IFIP Conference on Software Architecture* (WICSA'07), 2007; 23–23.
51. Czarnecki K, Helsen S. Classification of model transformation approaches. *IBM Systems Journal* 2006; **45**(3):621–645.
52. OMG. QVT, MOF query/views/transformations, ptc/05-11-01. Final adopted spec. 2005.
53. Costa-Soria C, Hervás-Muñoz D, Pérez J, Carsí JÁ. A reflective approach for supporting the dynamic evolution of component types. *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, 2009; 301–310.
54. Armbrust M, Fox A, Griffith R, Joseph AD, Katz RH, Konwinski A, Lee G, Patterson DA, Rabkin A, Stoica I, Zaharia M. Above the clouds: a Berkeley view of cloud computing. *TechRep No. UCB/EECS-2009-28*, 2009.
55. Varia J. Building GrepTheWeb in the cloud, part 1: cloud architectures. 2008.
56. Smith CU, Williams LG. Software performance antipatterns; common performance problems and their solutions. *27th International Computer Measurement Group Conference*, 2001; 797–806.
57. Navarro E, Cuesta CE, Perry DE. Antipatterns for architectural knowledge management. *International Journal of Information Technology and Decision Making* (in press) 2011.
58. Kitchenham BA. DESMET: a method for evaluating software engineering methods and tools. In *Experimental Software Engineering Issues: Critical Assessment and Future Directions*, vol. **706**, Rombach H, Basili V, Selby R (eds.). Springer: Heidelberg, 1993; 121–124.

AUTHORS' BIOGRAPHIES:

Carlos E. Cuesta is an Associate Professor of Software Engineering at Rey Juan Carlos University, in Madrid (Spain), where he is also the current Academic Secretary (Deputy Dean) at the School of Computer Science and Engineering. Previously, he has held positions as Teaching Assistant (1997–2002), Lecturer (2002–2003), and Associate Professor (2003–2006) at the Department of Computer Science in the University of Valladolid, also in Spain. Since he moved to Rey Juan Carlos University (2006), he has been the Director of the Master and Doctorate Programs on Information Technologies and Computer Systems (2006-2010), the Associate Dean of Academic Affairs, and Head of Studies (2010–2012) of his School. He has been Program Co-Chair of the second Joint meeting of two major conferences in Software Architecture: the Tenth Working IEEE/IFIP Conference on Software Architecture and the Sixth European Conference on Software Architecture (WICSA/ECSA 2012). Previously, he was the Conference Chair for the first edition of the European Conference (ECSA 2007). He is also an active member of the ECSA Steering Committee (continuously since 2006); he also participates in many other Program Committees and editorial efforts. He is a founding member of the VorTIC3 research group at Rey Juan Carlos University and has been previously linked to the ISSI group at Polytechnic University of Valencia (Spain). His main research interests are related to Software Architecture and also include many bindings to Service Orientation, Reflective and Self-Adaptive Systems, Model-Driven Development, Invasive Composition, and Concurrency.

Elena Navarro is an Associate Professor of Computer Science at the University of Castilla-La Mancha (Spain). Prior to this position, she worked as a researcher at the Informatics Laboratory of the Agricultural University of Athens (Greece) collaborating in the CHOROCHRONOS project funded by the Training and Mobility of Researchers program of the EU. Previously, she served as a staff member of the Regional Government of Murcia, at the Instituto Murciano de Investigación y Desarrollo Agrario y Alimentario, collaborating in the INTERREG II Project funded by the EU. During her master degree studies at the University of Murcia, she was a holder of several research scholarships funded by the Regional Government of Castilla-La Mancha and the National Government. She obtains her bachelor's degree and PhD degree at the University of Castilla-La Mancha and her master's degree at the University of Murcia (Spain). She is currently an active collaborator of the LoUISE group of the University of Castilla-La Mancha. Her current research interests are Requirements Engineering, Software Architecture, Model-Driven Development, and Architectural Knowledge.

DOI: 10.1002/smr

Dewayne E. Perry is the Motorola Regents Chair of Software Engineering at The University of Texas at Austin (UT Austin) and the director of the Empirical Software Engineering Laboratory (ESEL). The first half of his computing career was spent as a professional programmer and a consulting software architecture and designer. The next 16 years were spent as a software engineering research MTS at Bell Laboratories in Murray Hill, New Jersey. He has been at UT Austin since 2000. His research interests include empirical studies in software engineering, software architecture, and software development processes. He is particularly interested in the process of transforming requirements into architectures and the creation of dynamic, self-managing, and reconfigurable architectures. He is a member of the ACM SIGSOFT and IEEE Computer Society, has been a coeditor-in-chief of Wiley's Software Process: Improvement and Practice as well as an associate editor of IEEE Transactions on Software Engineering, and has served as organizing chair, program chair, and program committee member on various software engineering conferences.

Cristina Roda is a software engineer at Vector Corp. She received her MSc in Computer Science from the University of Castilla-La Mancha at the Superior Polytechnic School of Albacete, Spain, in 2012. Previously, she finished her bachelor's degree in Computer Science at the University of Castilla-La Mancha at the Superior Polytechnic School of Albacete in 2010. Her research interests are in Software Engineering, Model-Driven Development, Software Architecture, and Architectural Knowledge.

DOI: 10.1002/smr