Evolution Styles: Using Architectural Knowledge as an Evolution Driver

Carlos E. Cuesta
Rey Juan Carlos University
Department of Computing Languages and Systems II
ETS Ingeniería Informática, Campus de Móstoles, 28934 Madrid
carlos.cuesta@urjc.es
Phone: +34 91 488 8532

Elena Navarro
University of Castilla-La Mancha
Computing Systems Department
Avda. España s/n, Albacete, Spain
elena.navarro@uclm.es
Phone: +34 967599200 -2461
Fax: +34 967599224

Dewayne E. Perry
The University of Texas at Austin
Department of Electrical and Computer Engineering
perry@ece.utexas.edu

Cristina Roda
University of Castilla-La Mancha
Computing Systems Department
cristina.roda@alu.uclm.es

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. In order to carry out the 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, as it helps the architect to 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 SA, but taking into account the Architectural Knowledge as 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.

Keywords-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 in order 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 which would be later extended and developed several times [4], [6], [7]. Therefore, as the need of change will last over the whole life cycle of the systems, the introduction of proper processes, techniques, and tools that help us to deal with it becomes an essential part of the software development and maintenance process.

In order to 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 to 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 it may also evolve in order 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 become a critical issue in order to maintain its consistency, quality, etc.

In the very beginning of the SA area of research, the Architectural Design Rationales (ADR) that explain the SA specification were understood as very valuable assets of the architecting 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 Architectural Design Rationales (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.

Taking account of the importance of 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 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 that considered the introduction of ADDs and ADRs in their description to guide the architect while evolves the SA.

This paper is structured as follows. After this introduction, section 2 describes the necessary background about software evolution and Architectural Knowledge. Section 3 presents our proposal AKdES, which intends to combine the novel concept of evolution styles with the basis provided by Architectural Knowledge. 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 ETAK framework proposed by Noppen and Tamzalit [19]. In this work, a new concept is introduced named *architectural trait* that 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 regarding 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 knowledge base. However, this architecture knowledge base does not describe DDs and DRs per se, but different SA specifications that can be re-used to address different needs. Therefore, although it exhibits some similarities to our approach, it also exhibits clear differences.

Other related work has been presented in the area of architecture-based self-adaptive software that considers the exploitation of a knowledge-base to enact the software evolution as well. For instance, the Knowledge-Based Architectural Adaptation Management (KBAAM) 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 adaptations responses must be generated – that is, which structural modifications or adaptations policies 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 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 (AEM) [21] specifies adaptability requirements, guides the architect for their consideration during the SA design by means of a set of guidelines and for their analysis to determine whether these requirements are met. Other work focuses on the SA evolvability evaluation during the final phase of its design. In this category, a well known proposal is the Software Architecture Analysis Method (SAAM) [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 Architectural Knowledge. As stated by Avgeriou et al. [23], the AK turns into an integrated representation of SA specification, Architectural Design Decisions (ADDs) and Architectural Design Rationales (ADR). Most of the work [24–26] in this area has been related to the ADDs modeling in order to describe their structures, traceability relationships, etc. Other work [27], [28] has 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. The former is that presented by Tang et al. [32]. They present a graphical model named Architecture Rationale and Element Linkage (AREL) to describe the causal relationships between ADD and architectural elements that is described as Bayesian Belief Networks (BBN, [33]). Using this representation three probability-based reasoning methods are used: (i) first, *predictive reasoning* is used to determine which design elements could be affected by change if some architectural elements were to change; (ii) next, *diagnostic reasoning* is introduced to determine what could be the causes for those architectural elements to change; (iii) finally, *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 change before it happens. Therefore, this work can be considered as complementary, as our work is oriented to guide the evolution process.

The latter is that proposed by Tibermachine *et al.* [34]. They propose the introduction of a family of languages, called ACL, that allow the specification of the architectural choices associated to the decisions that are made during the development process. Two are the main constituents of ACL: (i) the *core constraint language* (CCL), a modified version of OCL that provides navigation operations, operators and usual quantifiers; and (ii) a set of Meta-Object Facility (MOF [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 both elements is called a *profile*, being defined a profile for each stage of the development process and modeling language. For instance, authors propose in this approach that a profile could be used to formalize the architectural constraints related to the architectural design stage and the Acme ADL [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 CCL 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 of, rather than to the guidance of the evolution process.

The notion of *architectural style*, as originally introduced by Perry & 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, i.e. 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 which fulfils these
constraints gathers the members with that style. As the style gets more specific, this range is smaller, and the prescription becomes almost a description.

There is an obvious relationship between software evolution and architectural styles, though 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 system’s evolution. Therefore different configurations of an evolving architecture have the same style – i.e. in general terms, 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, though 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 – i.e. 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). In order 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. Though Tamzalit’s work explicitly refers to the impact of evolution styles on Architectural Knowledge [19], [38], none of these proposals makes 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, i.e. 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 above – 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 Garlan et al.’s 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 language or platform, though it has to be adapted to any specific approach. Our only assumption is that we use Structured Architectural Knowledge (SAK) – that is, that the system includes an explicit representation of AK.

Table 1 summarizes the conclusions presented in this state-of-the-art section. Every existing proposal is compared to the rest of them (including our proposal AK-driven Evolution Styles, AKdES) using a set of features. Namely, the second column states if the proposal has explicit support for Architectural Knowledge (or some comparable concept, in the case of partial support); while 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, while the sixth states if the proposal includes explicit evolutionary steps as the results of that evolution. Finally, the last column indicates if the kind of evolution it supports is significant at the architectural level.

<table>
<thead>
<tr>
<th>Architectural Knowledge</th>
<th>Structured AK</th>
<th>Evolution Structures</th>
<th>AK-based Evaluation</th>
<th>AK-based Evolution</th>
<th>Architecture-Level Evolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETAK</td>
<td>≈ (traits)</td>
<td>×</td>
<td>≈</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>KBAAM</td>
<td>≈ (observations)</td>
<td>×</td>
<td>✓ (policies)</td>
<td>✓</td>
<td>✓ (dynamic)</td>
</tr>
<tr>
<td>AEM</td>
<td>≈ (requirements)</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>SAAM</td>
<td>≈ (scenarios)</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>AREL</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>ACL</td>
<td>✓ (CCL)</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Tamzalit</td>
<td>×</td>
<td>×</td>
<td>✓ (patterns)</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Garlan</td>
<td>×</td>
<td>×</td>
<td>✓ (paths)</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>AKdES</td>
<td>✓</td>
<td>✓ (BBN)</td>
<td>✓ (patterns)</td>
<td>✓</td>
<td>✓ (styles)</td>
</tr>
</tbody>
</table>

Table 1 A comparison of existing proposals considering both AK and evolution to some extent
(✓ full support, ≈ partial support, × not supported)

As can be observed, Table 1 makes easy to identify four different categories. The first group (the first four proposals) is composed of methods which do not use explicit Architectural Knowledge, but some comparable abstraction; except KBAAM, 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 which 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 AK

3.1 Architectural Knowledge as an evolution driver

As noted above, 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 & Wolf [10]. Styles express the central idea of the prescriptive view of software architecture – that is, the architecture prescribes a set of design conditions, which are specified as a set of architectural constraints: every configuration which fulfils 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.

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 which the system must fulfill at every stage during its evolution. The system can evolve, in principle, in any way which 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 does not only constrain 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 which 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, i.e. about the sequence(s) of intermediate steps that can be done.

As already mentioned, the existing proposals [9], [38] which define the notion of evolution style have used different ways to constrain this evolution. Specifically:

- 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 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. OCL).

- Garlan et al.’s approach [9] is similar to the above, though 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 what 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, both proposals are not only similar in structure and scope – they have been also specifically conceived to constrain evolution using just topological information. This is not bad in itself – in fact, in the context of software architecture this is both a coherent choice and 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 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 which 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, AK-driven Evolution Styles (AKdES), is to use Architectural Knowledge in the context of evolution styles, both to constrain and trigger evolution in software architectures. Therefore, in our proposal evolution styles are similar to those in previous proposals (particularly to those in Garlan et al.’s approach), but explicitly including Architectural Knowledge 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 Architectural Knowledge [38]. However, this is just a side reference, and existing work concentrates just on topological constraints.

Figure 1 Architectural Knowledge to evolve Software Architecture

The central idea of our proposal is summarized in Figure 1. As already noted, it is inspired to some extent by Garlan et al.’s approach, though many details are different. We can say that our proposal is based on Garlan et al.’s as much as their proposal is based in 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 Architectural Knowledge. In this example, the AK is structured, the names of the Architectural Knowledge relationships from the decision tree have been taken from [41]. Namely, the first frame (at time t) depicts a configuration with two components (see 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 (see 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, similarly 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 which 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.

This section provides an *abstract* definition for an evolution style, and it can be applied to any process which fulfils 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 which triggers a change, i.e. requiring an evolution step. In short, the identification of a situation which 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 – i.e. 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 Garlan et al.’s approach [9] can be considered as a related construction of these (or a variant case).

- **Evolution styles**: a set of evolution patterns which are conceptually related – i.e. they either affect the same set of features, or a set of related features which, together, could achieve some combined effect. Note that this set of patterns needs not to be connected, and of course it includes the unitary set as a special case; i.e., 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 A presentation of an evolution pattern and its effects](image-url)

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 evolution pattern is inherently dynamic. So, to understand this picture, it should be considered as the final result of applying a pattern consisting of three steps; i.e. the situation at time $t+3$. 
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 architectural design decision (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 the 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 architectural design decision (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, as well as 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 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 Architectural Knowledge in the evolution process, our intention is 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 – i.e. mostly structural or topological features (number of connections, etc.), as in many other proposals;
- Situations in the external context – i.e. something which happens on the outside, including human intervention – and therefore unpredictable by its own nature;
- *Active decisions* – i.e., 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* – i.e. 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.

*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 – i.e. what we have called structured Architectural Knowledge (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.

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 which is also used in Figure 1 and Figure 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 Architectural Design Decision (an ADD), and it deals with the system itself; but a change in the architecture defines an evolution decision, i.e. 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 which 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, i.e. 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 which 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, it 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 done 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 evolution decision (ED) is considered, and the corresponding 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 the 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 above. 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 in order to be able to apply it. Therefore, that process must at least comply with the following requirements:

- **Structured Architectural Knowledge**: the approach consists in exploiting this Architectural Knowledge, hence it must be available. As already noted, our proposal takes advantage of the structure of this knowledge – therefore structured Architectural Knowledge (SAK) must be assumed. The elements of this knowledge must include architectural design decisions (ADD) 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, 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 which stores structured Architectural Knowledge from requirements to architecture, and which is able to handle this information and process by means of a MDD-based approach.

4 PROOF OF CONCEPT: ATRIUM FOR EVOLUTION

In order 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 software architecture, 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 which fulfils three requirements. ATRIUM does comply with these requirements, and therefore it is able to use the ideas in our proposal. Specifically:

- ATRIUM provides explicit support for Architectural Knowledge, as explained in [44]. In fact, it has been extended to provide a very complex structure of Architectural Knowledge in a simple way, using just three AK relationships [41]. Therefore, it does support structured Architectural Knowledge, and in fact it 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 a model-driven development process [45], and in fact it 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 Architectural Knowledge using MDD-based techniques [44]. Hence, this support can also be extended to support evolution.

In the rest of the article, and for the sake of clarity, we will assume 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 which fulfils the noted requirements.

4.2 Describing ATRIUM

Figure 3 shows, using 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:
Figure 3 Describing ATRIUM

- **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 KAOS [47] and the NFR 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 Quality requirements (SQuaRE, [49]). The latter is used as 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 describes 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 build 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 (M2M, [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 metamodel used to describe the proto-architecture, because the architect only has to describe the needed transformations to instantiate the architectural metamodel he deems appropriate. However, the necessary transformation to generate PRISMA architectural models [53] have been already defined.

4.3 Architectural Knowledge in ATRIUM

As presented in [44][41], 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, i.e., 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 behaviour to meet the requirements, both functional and non-functional. For this reason, two key attributes are included while they are described: DesignDecision (DD) and DesignRational (DR). A seamless transition is performed from requirements to operationalizations by means of the contribution relationship, in order to specify an Operationalization contributes to/preserves 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 software architecture by exploiting the network of AK. In order to provide ATRIUM with this facility, several relationships were defined in its metamodel, 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’s consider A and B operationalizations, describing different design decisions. Having a constrains 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’s consider A and B operationalizations, describing different design decisions. Having an inhibits relationship from A to B, means that if A’s design decision is made, it hinders B’s design decision to be made.

- **excludes** is a binary and unidirectional relationship with stronger negative semantics than inhibits. Let’s consider A and B operationalizations, describing different design decisions. Having an excludes relationship from A to B, means that if A’s design decision is made, it prevents B’s design decision to be 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 Architectural metamodel. 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 Metamodel.

It is worth noting which 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 design decisions and design rationales that were analysed 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:

- **Evolution conditions.** The condition which 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 times 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 architectural design decisions (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 the 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 metamodel, as shown in Figure 5, as a special case of DesignAsset.

- **Evolutionary step.** It describes an action, i.e. 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 to 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 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 Metamodel. As it is defined as an extension it does not only inherits 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 Metamodel, it could be also defined in the Goal Metamodel. 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 which can be traversed and reused.

5 Example: Evolving a Cloud Architecture

In order 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 Architectural Knowledge 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 which 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 fulfil a real-world necessity. This is relevant because that is the kind of situations which 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.

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” which 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.

![Diagram of Radio Station System](image)

**Figure 6** Radio Station System at runtime: version 1.0

Every time that a listener enters the station, he searches for a certain program – i.e. 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 “software-as-a-service” (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 due to 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.

Both the “canonical cloud architecture” and our specific example, as presented above 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, 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 in order to ensure that no request (or response) gets 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 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. Using the popular Amazon AWS platform, 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 – e.g. the billing subsystem has been deliberately omitted. However, it must be present in any cloud-based application: every service in the cloud costs
Of course we would assume its function, i.e. 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 a lot of 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 not 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 due to 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 streaming service (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 criticises 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) to 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 above) 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, 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.

In order to simplify the presentation, we have chosen the same graphical format than 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, while the previous one just provided generic terms for any decision.

First, the *evolution condition*, requiring the architecture to evolve, is detected (see previous subsection). The first time this condition is evaluated, it leads to *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 FCFS 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 architectural design decision (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 labelled 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 (EA.3) a different approach. 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) gets inhibited, and the 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 gets 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 (EA.4) a related solution: to use a relay system. It is still true, as considered above, that many listeners would like to access the same program. Then, instead of connecting all 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. A 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 below, this defines the final step in this specific evolution, ultimately leading to the final architecture (see Figure 8 in section 5.5).

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 have 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 Architectural Knowledge; 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 Architectural Knowledge.

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 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.

5.5 Final situation: result of applying the evolution style

Though 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 (see 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: first, the original architecture (Figure 8) has evolved from a client-server style to an hybrid architecture, which includes a P2P 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 a lot of 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 needs not to 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 Architectural Knowledge. The model of the latter includes all the relevant elements as defined in the ATRIUM metamodel (and as already exposed in section 4.4), independently 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, considered 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:

- Four DesignAssets, one to describe every alternative branch (DA.1-4), apart from those which 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 Morpheus toolkit; the only issue is that these operations must be applied dynamically, during the system’s runtime.

Please note that ATRIUM covers the 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 the responsible to be in charge of performing these actions.

In summary, the example shows how our management and use of the AK makes 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

Software Architecture 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, the use of this evolution style in the context of a cloud-based application example is described in some detail and provides 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 software architectures 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, that would to 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 not a 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 and CSD2007-022 by Program CONSOLIDER INGENIO 2010) from the Spanish Ministry of Science and Innovation. Prof. Perry is supported in part by NSF CISE Grants IIS-0438967 and CCF-0820251.

REFERENCES


