*

*International Journal of Information Technology& Decision Making*


World Scientific
www.worldscientific.com

# ANTIPATTERNS FOR ARCHITECTURAL KNOWLEDGE MANAGEMENT

ELENA NAVARRO

*Laboratory of User Interface and Software Engineering, Computing Systems Department,*
*University of Castilla-La Mancha,*
*Albacete, 02071, Spain*
*elena.navarro@uclm.es*

CARLOS E. CUESTA

*School of Computer Science & Engineering,*
*Rey Juan Carlos University*
*Mostoles, 2893328933 Madrid, Spain*
*carlos.cuesta@urjc.es*

DEWAYNE E. PERRY

*Department of Electrical and Computer Engineering,*
*The University of Texas at Austin,*
*Austin, TX 78712, USA*
*perry@mail.utexas.edu*

PASCUAL GONZÁLEZ

*Laboratory of User Interface and Software Engineering, Computing Systems Department,*
*University of Castilla-La Mancha,*
*Albacete, 02071, Spain*
*pascual.gonzalez@uclm.es*

Received 28 July 2011
Revised 15 January 2013
Communicated by (xxxxxxx)

Recent research on Software Architecture has recovered its original emphasis on keeping track of design decisions and their rationales during software development, compiling them under the name of Architectural Knowledge (AK). This knowledge is composed of explicit, atomic decision assets, which relate to each other creating a decision network structure. We argue that relationships in these networks of AK contain valuable information, in particular when they describe negative semantics. We use reusable knowledge, in the form of antipatterns, to exploit and manage these negative semantic relationships systematically. After examining and classifying the kinds of AK relationships, we describe a method that enriches this network by means of antipattern structures. To show the feasibility and suitability of this approach, we provide a proof-of-concept by applying it to an existing process, ATRIUM. A concrete example illustrates our approach in which we use the Excessive Dynamic Allocation performance antipattern against the classic Gas Station metaphor. Results of the use of the presented approach into three different projects with different complexities show both the feasibility and applicability of our method. The combination of this model-driven support and explicit AK makes it possible to go beyond traceability to a more proactive AK management system that may additionally trigger modifications in the final architecture.

*Keywords*: software architecture, architectural knowledge, antipatterns, requirements engineering, model-driven development.

## 1 Introduction

In a typical software development process, the rationale for the architecture and the intent behind the design decisions are lost in the pursuit of that final system artifact, code. If by chance the architecture and design is documented, the documents themselves are usually out of date or incorrect – in other words, probably at least as harmful as they are helpful.

Given that architecture rationale [72] 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 claim that the rationale and intent, instead of being lost during the development process, can be captured as a byproduct of architecting and designing the system. Every decision is explicitly specified and included in the process of architecting the system and each decision becomes a part of a hyper-textual document that represents the network of architectural assets with their complex relationships. The advantage of such an approach is that it provides traceability and, optionally, even backtracking for design decisions.

The resulting structure, the *architectural rationale*, can be considered just documentation, the transcription of the thought process followed during design. However, in recent research it has acquired a more active role. It can be seen as a computational structure, composed of small assets, or artifacts, of design knowledge, tracing back to some requirements and forward towards an implementation. As such, it can be considered 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; we talk about *Design Decisions* (DDs) and *Design Rationales* (DRs), which comprise a concrete decision in the architecture process, and the reasoning behind it. This conception is consistent with the existing literature on design; indeed, design rationale is a topic with a long tradition on its own [53]. The purpose of research on AK is to incorporate this derivation of the architectural design in the architectural process – that is, include that derivation via design decisions and rationales as part of the architecture description. When only the final architecture is described, this derivation is lost and becomes *unrepresented design knowledge* [86].

From the AK perspective, architecture is therefore better defined as "a set of design decisions" [34]. But it is indeed, not a set, but a *network*. DDs are related to some others, both forward and backwards, defining an intertwined chain of decisions. When the rationale is conceived as a network of AK assets, the network itself can subsume much of its structure. Moreover, these *AK relationships* (AKRs) can be either *positive* or *negative.* A positive AKR between two DDs determines that both of them should be made for the software architecture to be correctly specified, as one depends on the other. On the other hand, a negative AKR between two DDs establishes that there is a conflict between them, and that a compromise has to be achieved to resolve the conflict in the architecture. However, when trying to provide automatic (or at least, semi-automatic) support for this process, positive relationships are easy to deduce and, hence, to process. However, negative relationships are not easy to deduce and process. A decision *not taken*, or

inhibited due to some other decision, is not visible (there are no *traces* of it) in the final architecture. This *unrepresented knowledge* [86], which cannot be deduced from the final design, must be introduced manually. This is perhaps the most important reason to introduce the distinction between positive and negative AK relationships.

The main contribution presented in this work is the use of standard "negative" design knowledge, namely *antipatterns*, as a source of information that helps to identify potential *negative* relationships within the AK structure and suggest possible solutions. As a proof of concept, we have implemented this support in an existing model-driven methodology, ATRIUM [52], that already has the support for DDs [61]. This enables us to offer two secondary contributions: (i) the extension of ATRIUM with a new activity named *Analysis* that embodies the exploitation of antipatterns for AK management; and (ii) the definition of new AK relationships that are necessary both to change the AK to a network of AK and to weave antipatterns into this network.

This paper is structured as follows. Section 2 describes the proposal presented in this work: the automation of the process of detecting negative relationships by exploiting antipatterns. Next, Section 3 describes the current approaches to AKRs, discussing their coverage and benefits, while Section 4 presents the related work about the exploitation of anti-patterns. Section 5 demonstrates, by means of a proof of concept, that our approach can be put into practice, describing its application by using the well-known gas station example in Section 5.4. Section 6 describes the most relevant results obtained from its application into three different projects of different complexities. Finally, Section 7 presents our conclusions and suggestions for further research.

## 2 Problem Statement: Antipatterns for AK Management

As stated in the previous section, we can classify AKRs into two kinds: positive and negative AKRs. The former makes it possible to specify obligation semantics within the AK network. These kinds of relationships are specified to describe when some DDs need, constrain, etc., other DDs. The latter group implies negation semantics, such that specific conditions, situations, etc., cannot occur in the system-to-be. Usually, negative AKRs are used either to describe choices (DDs) which, if taken, would mean bad decisions for the system-to-be, or sets of DDs whose combination would result in architectural problems. Although both of them are very noteworthy for the design and the maintenance of software systems, the negative relationships are especially relevant as they can lead to faults, inconsistencies, conflicts or other design problems in the final system. Therefore, their early detection can avoid problems in terms of the quality of the final system.

Usually, the detection of the positive relationships emerges naturally as part of the process of the description of the software architecture. However, the negative relationships must usually be detected by the architect without any automated support, relying on his previous architectural experience, or on his knowledge of the current system and its history. This happens because software development is a *constructive design* process, that is, for many decisions to be made it is mandatory that other ones already have been taken. However, when a decision is *not taken*, something is just *not*

*done*, and hence the corresponding negative relationship does not emerge naturally from the final result. Therefore this knowledge of what was not done is lost, unless we explicitly capture it.

Manual analysis can become cumbersome and error-prone, especially when it is carried out by an inexpert architect. Therefore, the introduction of techniques that help in the automatic (or at least semi-automatic) detection of these negative AKRs would provide an initial advantage for creating higher quality specifications of software architectures, just a first step to bridge the gap between initial development and the evolution of a system.

In order to meet this challenge, we propose the use of antipatterns. Brown et al. [9] defined an antipattern as "a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences." As can be observed, antipatterns describe very useful knowledge about ineffective software practices in terms of software development processes, software architecture, project management, etc. This is not a new concept in Software Engineering but has been applied over last decades, almost simultaneously with the introduction of *design patterns* [28] or *software architecture patterns* [11]. Unlike antipatterns, these do describe good practices in software development as they document good solutions to recurring problems. When software patterns are formally identified and specified, they enable us to reason about the architecture because they describe how and why the software architecture is as it is. This has motivated some proposals that have emerged in the AK management arena that promote the use of patterns as a way to detect and document DDs [30] automatically. However, as far as we know, there are no approaches that promote the use of antipatterns in the context of AK management, despite their utility to document bad DDs or bad combinations of DDs.

As can be observed in Fig. 1, whenever an antipattern is detected, it involves one or more artifacts. It can also determine that either a bad decision was made during its specification or an improper combination of DDs led to such a condition. For instance, during the requirements stage, the antipattern *Functional Decomposition of Use Cases* (UCs) [22] can be detected if the architect made the decision of specifying a set of all too simple UCs instead of one unique and more comprehensive UC that can be more easily traceable and assignable during the software development process. *One-Lane Bridge* [78] is another example of an antipattern that can be found when the software architecture is being specified. It indicates that concurrency problems can arise whenever the architect decides that only one, or only a few, processes can proceed to execute in a concurrent way causing the other ones to wait. At the code or design level we can also find this relationship between bad decisions and antipatterns, as for example when the *blob* antipattern [9] is found. In this case, developers assign most of the responsibility for the system to a unique class. Therefore, we can establish a direct relationship between antipatterns and artifacts that we have represented as *involves* in Fig. 1. But we can also establish a direct relationship between antipatterns and bad DDs, described as *affects* in

Fig. 1, because the detection of an antipattern may suggest or cause a change in the decision network.

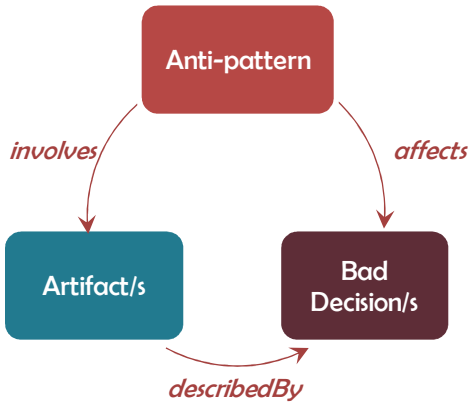



Fig. 1. Elements affected/involved by an antipattern

Most of the current approaches in AK management, such as [35][59][82], intend to provide solutions to maintain the relationship between the artifacts produced, the software development process, and the DDs that lead to its specification in their current form (shown as *describedBy* in Fig. 1). This relationship can be used during the software lifecycle to evaluate in advance the impact of changes; to help novice analysts to understand why the system is the way it is; etc. Therefore, they constitute a valuable asset during both the software development process and the maintenance stage.

Our approach intends to exploit *describedBy*, *involves* and *affects* relationships to help the architect in the process of determining when bad DDs were made during the process and when negative relationships among them can arise. With this goal, our approach can be described as follows:

- Let us consider an approach for AK management where clear relationships between DDs and artifacts are defined – that is, a similar notation to *describedBy* exists;
- Apply the following process: (1) apply some of the existing algorithms for detection of antipatterns; (2) whenever an antipattern involves an artifact, mark its related DDs as potentially bad DDs and establish a suspicion on the relationships among them; (3) analyze the potentially bad DDs and resolve possible conflicts, taking into account the antipattern they are related to.

It should be noted that this process cannot be fully automated, as several steps could require human intervention. However, our purpose is to provide as much automation as possible. This article focuses mainly on steps (2) and (3). The step (1), dealing with detection, will be based on the state-of-the-art research, and will make use of existing tools, but it is not the main concern of this paper. On the other hand, the step (3), related to the analysis of the situation, should not be automatic, as the architect has to evaluate what the problem actually is. The architect must always be the one who takes the final decision. However, we can provide semi-automatic support for this step: the architect

does have an automatic warning and helpful information to detect and solve a situation that perhaps he would not have come across otherwise.

As can be observed, the proposal is not related to any existing approach to AK management and our primary requirement is that we should provide notations for AK relationships and clearly identify DDs.

A proof of concept, described in Section 5, has been carried out to validate our proposal by modifying ATRIUM [52] to incorporate the above described process. In order to improve the understandability of our approach, first Section 3 presents a discussion about AKRs discussing their benefits and drawbacks and next Section 4 provides a brief background on antipatterns.

## 3 AKRs: Networking Architectural Knowledge

The most natural way to think about a decision is to consider it in isolation, separately from the rest of the system by a process of abstraction. It is also the easiest way to document it, particularly when provided with a template defining its basic attributes. A set of such isolated decisions would provide basic knowledge about a system's design, as they operate on the same substrate, namely the system *architecture*. However, such a view is necessarily incomplete and partial. Design decisions are connected, because they refer to each other, interact with, and affect each other. In fact, there is a complex fabric of relationships surrounding them: even the simplest model of DDs in the existing literature provides some structure. First, every decision is chosen on purpose: it can be *traced* back to some goal it achieves, or the requirement(s) it satisfies. Second, every decision is *implemented* by some design artifact, some architectural element. Apart from these two reifying relationships, any decision relates to other decisions at the same level in different ways.

Design Decisions and Rationales (DDs & DRs) can be correctly considered as the basic assets that together comprise our design knowledge. As already noted, this is consistent with the existing practice on design rationale [53]. However, to provide a coherent system-wide rationale these assets have to be complemented with the information about their mutual inter-relations and connections. Therefore, their Architectural Knowledge Relationships (AKRs) present themselves as another invaluable basic asset, and transform the *set* of design decisions into a *network* of design knowledge. This point has already been noticed by other authors, such as [84].

Some authors [34][35] describe the network of decisions using a single (dependency) relationship, or perhaps several assimilated ones. Even this simple layout is useful and much more so than a simple "set" of decisions, as many details depend just on topology. However, these uniform links lose an essential feature: the direction, or type, of the semantic relationship. Indeed, there are *positive* and *negative* relationships, respectively exposing synergies and divergences within the design; it is obvious that their influence extends to the entire architecture.

Many authors recognize the inherent complexity of managing and combining "knowledge assets", and thus they advocate an *ontological approach* [1][23][25][44] [48]

to provide a solid basis for this reasoning. These ontologies are able to identify categories of DDs (such as *ontocrises*, *diacrises* and *pericrises*, dealing respectively with concepts, concrete features or system-wide constraints), and to enumerate the basic properties that describe the knowledge contained in an asset. And of course, they can also help to define the basic relationships between assets, and even a more complete metamodel of DDs.

However, most of the work in this area, even that inspired by an ontological approach, has focused on the description of the (internal) *structure* of DDs. This includes, apart from most of the citations in the previous paragraph, many others [35][82][88]. Some of them are specifically concerned with documentation, which is their reason for providing a template [30]; some of them support their definition on a strong empirical basis [24]; and even some of them do both [86]; but for most of them, relationships play a secondary role, if any role at all.

For example, Tang et al. [82] provide a structural model for their basic asset, *the architectural rationale*. This is defined as the composition of a quantitative and a qualitative rationale, plus scenarios describing special cases; and possibly some aggregations, defining an *alternate* DD for this, which in turn could be again a composite. Also, there are trace relationships to several views. Considered in isolation, such a complex structure is indeed useful, particularly for documentation, as it provides the required support for a full description of a decision, the rationale behind it and even potential alternatives. However, this model does not seem to scale well: first, every single decision is complex, including cases and alternates; and second, there is not support to relate a decision to any other, except for aggregation itself. Therefore, the resulting "global" rationale is defined as an unstructured discourse, which does not provide an organization for architectural knowledge.

The exploitation of a *network* of design assets improves the situation significantly because it captures and structures much of the inherent complexity of those inter-related design assets. Of course, a "conventional" set of complex DDs, in which all relational information is captured by means of attributes, would be able to describe exactly the same information; i.e., there are equivalent descriptions which need not to take the form of a network. However, the "networked" version can be considered a more flexible representation: while it is just making explicit some implicit relationships, in doing so it also makes much easier to handle and manage them.

As already mentioned, some authors have explored the issue of AKRs in some detail. Most of them identify only a few relationships, and these proposals converge mainly in two of them. The first one is *constrains*, which expresses the self-evident positive implication; and the second is *alternative*, probably the most cited one, which expresses variability. It provides the support to express a choice and refer to otherwise unrepresented, *vaporized* design knowledge [86]. It is also very useful in the context of product lines [77]. Apart from these, however, there are a number of additional proposals; some of them use also a significant *semantic* perspective. We will examine them more carefully in the following.

Table 1 summarizes most of the AKRs described by current research (column R*elationship*), providing their terminological equivalences (column *Synonyms*) and the references where they have been defined or used (column *References*). As can be observed, Kruchten [44] provides the most complete reference about this topic. He not only provides the definition for most existing AKRs, but also the way they relate to each other, emphasizing their ontological foundations. Other references, such as [1][25][45], basically use the same ontological framework.

Of course, these equivalences are not always direct mappings, and therefore they do not actually define strict "synonyms". In Table 1, two terms appear in the same row when they essentially describe the same information, i.e. when one of them can be derived from the other. Also, as aforementioned, they are not mutually exclusive: *bound to*, for instance, is defined in terms of *constrains*; this is also the case of *enables* and *comprises*, which are described respectively as weaker and stronger versions of it.

Table 1. Evaluating proposed AKRs

| Relationship | Synonyms | References |
|---|---|---|
| Alternative | Alternate DD | [44][23][25][82] |
| Bound To | | [44][25] |
| Comprises | Made Of | [44][25][82] |
| Constrains | Implies, Refines | [44][23][25][34] |
| Enables | | [44][25] |
| Forbids | Excludes | [44][25] |
| Not Complies | | [44][25] |
| Related To | | [44][25][30] |
| Overrides | | [44][25] |
| Conflicts with | (*) | [44][25] |
| ModelElementBinding | (*) | [41] |
| Traces From/To | Addresses, Implements (*) | [44][23][25][30][82][83] |
| Depends on | (**) | [34][45] |
| Subsumes | (**) | [44][25] |

Although the names of these relationships must be understood in terms of this particular context, it is obvious that some of them can be considered within a wider scope; these have been marked with a star (*) in Table 1. These can be seen as "extended" relationships, which should not be considered strictly AKRs, as they would be able to relate elements which are not DDs.

This refers, in particular, to *conflicts with* relationship, along with *traces from/to* and *ModelElementBinding*. In a restricted sense, they obviously refer to marking conflicting decisions and keeping track in a chain of decisions. All of them can also be conceived as derived relationships (see Table 1) in the context of DDs, but there is also a general meaning out of this scope. In particular, the second relationship (*traces from*), can be read, in a general context, as the traditional traceability relationship (*trace*), which describes the history of every element in the architecture, and which is a basic element also for AK [61]. Meanwhile, the first one (*conflicts with*) could also refer to well-known

schemas of conflict between requirements, which combined with *trace* could also refer to some derived decisions.

Another comment must be made about relationships which have been marked with a double star (**). These are in a similar situation, but *generalization* is the issue here. Both *depends* and *subsumes* can be considered as generic versions of the rest. As in many other contexts, dependency can be considered the basic relationship, by definition, so that every other inherits from it. Then every AKR is also a dependency [35]. On the other hand, *subsumption* is usually considered as the target relationship for ontologies [2], acting as the transitive closure for ontological relationships. In fact, it could be used to "flatten" a structure with several AKRs into a single-relationship model to apply basic analysis, provided that the model can be considered formalized enough.

Considering all of the above, we can see that AKRs provide a particularly rich framework to capture design knowledge. As already stated, this enables us to capture or represent a richer set of AK with a smaller set of DDs. Also, the conceptual closeness between some of them makes it possible to apply simple analysis techniques. The potential of exploiting all this information remains still unexplored. To provide a fair comparison, just consider that Tang *et al.* use only one relationship (*traces*, indeed) in [83]. But as they extract every consequence, using Bayesian Networks analysis, they obtain quite complex and significant results. In summary, though not very frequently applied yet, AKRs provide a very useful framework to define, use and reuse architectural knowledge.

Therefore, there are several additional arguments to support our preference for the networked version. First, it is true that an attribute-based presentation can include the same information; but in that case, the role of many arguments would be just to replace the relationship itself. Even worse, *transitive* relationships would be much harder to manage, and, if these were introduced as attributes, it would be much easier to introduce redundancies, which would ultimately cause the appearance of anomalies in the structure. These would be similar to classic data anomalies, i.e. insertion, deletion and updating issues. For instance, consider that a certain decision A implies ("constrains") a decision B, and eventually decision A is removed, but B is not. In a networked representation, it would be obvious that B is a dangling thread, while in an attribute-based version this could pass unnoticed.

Second, some AKRs must assume ontological features, which can be independent from DDs themselves, and therefore are more conveniently separated. And third, the *structure* of the network itself becomes significant. Indeed, the connections capture some essential facts, not only about the architecture itself, but also about the design and construction process. We are not only able to include information about decisions which were discarded during this process (the classic "unrepresented design knowledge"), but we are even able to capture the *sequence* of decisions which led to a certain choice. The AK network is a dynamic structure – it evolves as the architecture requires, while leaving a trace of the design process. For instance, let's consider a decision A related to some component B. Now, if A is rejected, component B can be removed, but A still remains in

the network – related to a new decision C, which inhibits A. Growing this way, the resulting structure describes the design decision process, rather than just the architecture.

This conclusion is not surprising, as it is something that often happens in the context of knowledge representation, which appears as a related area indeed. Therefore, a *network* of very simple DDs is able to capture perhaps even *more* information than a poorly structured set of complex DDs.

Finally, there is an additional reason, and it has to do with performing *analysis*. When AKRs are provided in an explicit, relational form, it is quite simple to use network theory [5] to examine and analyze the resulting structure. Therefore, it will be much easier to identify *key* decisions in the proto-architecture, by detecting them as *hubs* in the network; and independent branches of the development would be also easily located by identifying the DDs which act as *bridges* for them. However, we will not deal with this topic further, as it is out of the scope of this article, and it will be considered for future work.

Of course, all of this reasoning should not be confused with decisions about relationships or connectors in the target architecture, which are also described in, e.g. [31]; obviously, these are just described using regular DDs and do not affect the topic.

## 4 Background on Antipatterns

A pattern is a solution to a recurring problem [28]. They are high-quality experiences distilled into a form that facilitates their reuse and application in the design of different types of software. Antipatterns are similar in their definition to patterns as they also document solutions to recurring problems [9]. However, they differ from patterns in that their use produces negative consequences for the system. For instance, it has been empirically validated that they have a high impact on change and fault-proneness in object-oriented systems [39]. For this reason, the detection of antipatterns and the application of their related solutions constitute a valuable asset in terms of quality of the final product. However, they have not received as much attention as they deserve from either academia or industry.

Most of the existing research has focused on the identification and specification of antipatterns. In the literature about antipatterns, the work by Brown et al. [9] is especially relevant. They have identified antipatterns that can be detected not only in the architecture and the design of systems but also in project management. In addition to this catalogue of antipatterns, other research [49] [79] has identified performance antipatterns as their main concerns because of its direct impact on the quality of the software product. These approaches have focused on identifying problematic situations, in terms of software architecture or design, that diminish response time, increase processing time, etc. Security has been another concern for which antipatterns have been defined. Kis [37] describes two antipatterns that clearly identify the impact of business contexts on the security policies to be implemented. Antipatterns have been also described related to testability. Baudry et al. [7] have defined a set of antipatterns and how they should be dealt with in order to reduce testing effort. In addition, some research has been done to detect antipatterns in earlier stages of software development, such as those presented by

El-Attar and Miller [22]. They describe some antipatterns that can be detected in use case models, by using the Object Constraint Language (OCL) for their description and later identification.

The exploitation of the antipatterns has been carried out from different perspectives. Most of the research has been related to the definition of techniques and tools for their identification. For instance, Moha *et al*. describe in [51] how formal concepts and metrics can be used to detect the *Blob* antipattern. Similarly, the methodology COMPAS [54] has been proposed to evaluate distributed component-oriented applications in terms of performance by detecting antipatterns for specific technologies, such as Enterprise Java Beans. COMPAS exploits a Model Driven Architecture approach to describe the necessary models for each step of the process: monitoring, modeling and prediction. Parsons and Murphy [64] describe a framework that detects performance antipatterns in Java EE architectures by monitoring component-based system to build a performance model used to detect EJB-specific performance antipatterns. Another related proposal is the Bayesian Detection Expert (BDE, [38]) that detects antipatterns by building Bayesian Belief Networks using the information extracted from the code by using the Goal Question Metric (GQM, [6]) methodology. SPARSE [76] and its extension [75] describe the detection of project management antipatterns from a different perspective that takes into account the fact that this kind of antipatterns do not emerge in isolation but jointly. With this aim, SPARSE defines project management antipatterns by means of ontologies for their detection and analysis.

Some approaches go one step further, by dealing with antipatterns as one of the inputs for refactoring processes. Work presented by Cortellesa et al. [13][14][15][16] is especially relevant in this area. In a preliminary work, Cortellessa *et al*. [16] exploit antipatterns as part of a process that evaluates performance by using interpretation matrices and proposes architectural alternatives to improve the results. Later on, they present a proposal [15] that exploits model-driven techniques for the detection and resolution of antipatterns. Finally, they have also presented an alternative [14] that exploits logical predicates for the detection of antipatterns and a process [13] that guides the software designers in the analysis of performances measures, model entities and performance antipatterns to classify the level of guiltiness of each detected antipattern. In the same context, Martens and Koziolek [49] first explore the design space by using metaheuristic search techniques, and second evaluate the results to detect antipatterns. These antipatterns can be eliminated using the solutions attached to their descriptions. More recently, Trubiani and Koziolek [85] present an approach to automatically detect and solve specific software performance antipatterns of the Palladio Component Model (PCM).

Antipatterns have drawn criticism for lack of formalism in their specifications. In order to solve such problems, research has been done that provides such techniques for their specifications. For example, Ballis et al. [3] have proposed a new visual language to describe antipatterns (and patterns). It has been defined by extending UML with some new graphical elements so that antipatterns can be specified in a more rigorous way.

Dietrich *et al.* [21] have defined an infrastructure using social networks and semantic web technology that, although initially developed to exploit patterns, can be used to describe antipatterns as well. More recently, Stamelos [81] has proposed the exploitation of Bayesian Belief Networks, Ontologies, Design Structure Matrices, and Social Networks as proper tools to formally represent Software Project Management anti-patterns.

Thus, while antipattern approaches have been applied in various areas, there is no research exploiting antipatterns in the AK management area. As far as we know, the research presented by van Lamsweerde [47] is the only one that presents some similarities to our approach. His work describes a formal, iterative method for security requirements specification that exploits anti-models, derived from the goal model of the system-to-be [47], to show how elements of the system-to-be could be threatened. These anti-models are elaborated as a refinement process from the initial anti-goals obtained by negating security goals (Confidentiality, Privacy, Integrity and Availability) until anti-requirements and vulnerabilities are determined, that is, anti-goals realizable by attacker agents or attackee agents, respectively. These anti-requirements are operationalized by describing the potential capabilities of their related attacker in order to generate countermeasures in the goal model. These countermeasures can determine a modification of the goal model - that is, the decisions in terms of the goals and requirements of the system-to-be must be made. This approach exhibits several advantages such as its applicability in early stages of development, or the exploitation of temporal logic to facilitate the generation and analysis of the anti-goal models. However, it does not deal with design decisions, and unfortunately, it cannot be generalized to other approaches.

## 5 ATRIUM: A Methodological Proof of Concept

In order to validate the approach presented in this paper, we have selected a methodology that allows us to deal with AK, and to manipulate its models in an easy way. With this goal in mind, we have selected ATRIUM. This methodology has been designed for the concurrent definition of requirements and software architecture, providing automatic/semi-automatic support for traceability throughout its application, and for the description and manipulation of AK at different abstraction levels [59]. As it follows a MDD (Model Driven Development [74]) paradigm, our approach can be easily put into practice. In the following section we briefly introduce ATRIUM, and present our approach in Sections 5.2 and 5.3. Finally, the Gas Station example is used in Section 5.4 to illustrate the proposal.

### 5.1 Describing ATRIUM extension

Fig. 2 shows, using SPEM 1.1 [62], 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 KAOS [20] and the NFR Framework [12]. This activity uses as input both an

informal description of the requirements stated by the stakeholders, and the 25010:2011 *Systems and software engineering - Systems and software Quality Requirements and Evaluation - System and software quality models (SQuaRE,* [33]). 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.

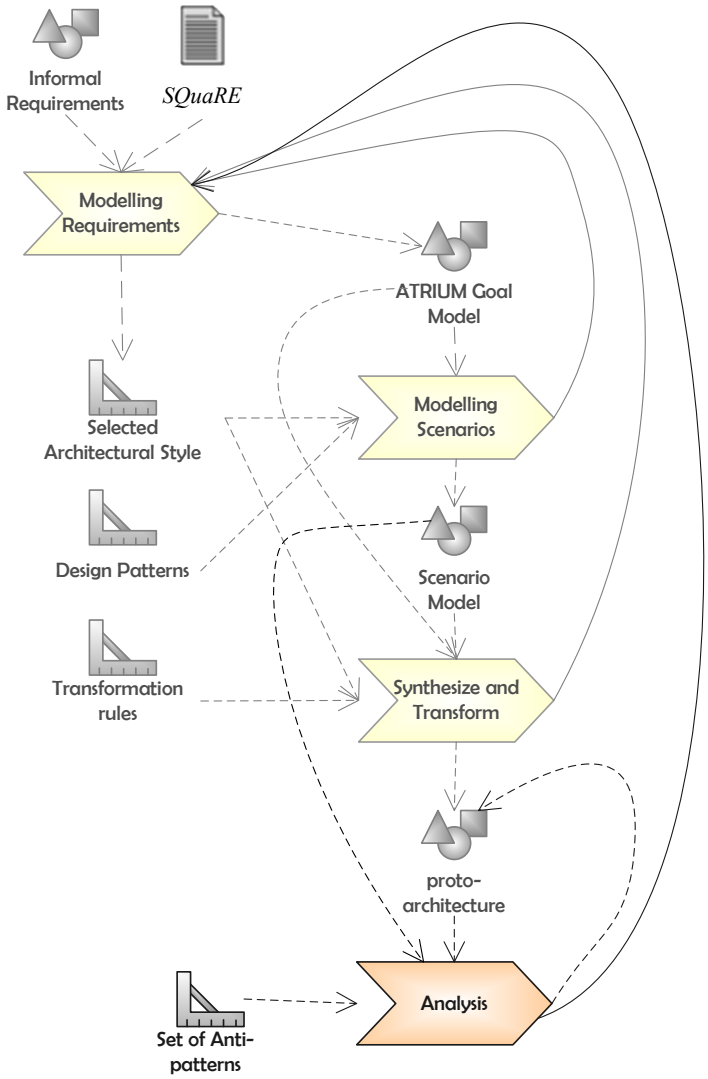



Fig. 2. ATRIUM and its extension: Analysis activity

- *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 [60]. 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 [59]. 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, [17]), specifically, using the QVT Relations language [63] 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. Currently, the set of transformations [56] to generate the proto-architecture instantiating the PRISMA architectural model [70] has been defined. This activity resembles other works, [18][65], that also generate the software architecture using M2M techniques.

We want to highlight that we have included in ATRIUM a new activity called *Analysis* (see Fig. 2). Its main goal is to facilitate the integration of the approach presented in this paper, that is, to evaluate the architectural knowledge specified in the proto-architecture regarding the set of antipatterns. This activity will be detailed in section 5.3. Another advantage of ATRIUM is that a supporting tool, called MORPHEUS [58], has been developed to put into practice its different activities, whose extension to support the ATRIUM *Analysis* activity is described in section 5.4.

Finally, it is worth noting that this extension of ATRIUM could seem similar to the use of *architectural tactics* as presented by Kim et al [40]. However, these authors exploit architectural tactics as a recommendation for the construction of the system, that is, they are used while the system is being built. However, the new activity of ATRIUM exploits antipatterns to analyze the already defined software architecture and determine which combination DDs carried out to problems of the system. Therefore, they are not identical but complementary.

### 5.2 AK relationships in ATRIUM extension

As presented in [59], the ATRIUM Goal Model is in charge of manipulating most of the AK. The building blocks of this model are *goal*, *requirement* and *operationalization* (see section 5.4 for an example of a Goal Model). *Goals* constitute expectations that the system should meet. *Requirements* are services that the system should provide or constraints on these services. The main difference between requirements and goals is that the former can be validated, but we cannot really describe a test to validate the latter because it is really something we expect about the system-to-be. Finally, *operationalizations* describe both the DD and the DR and how they satisfy the established requirements. Goals are refined into sub-goals and finally into requirements by using *AND* (*OR*) relationships to determine if all (at least one) of the sub-goals must

be satisfied to satisfy the root. A seamless transition is performed from requirements to operationalizations by means of the *contribution* relationship, in order to specify how solutions contribute positively and/or negatively to meet the corresponding requirements.

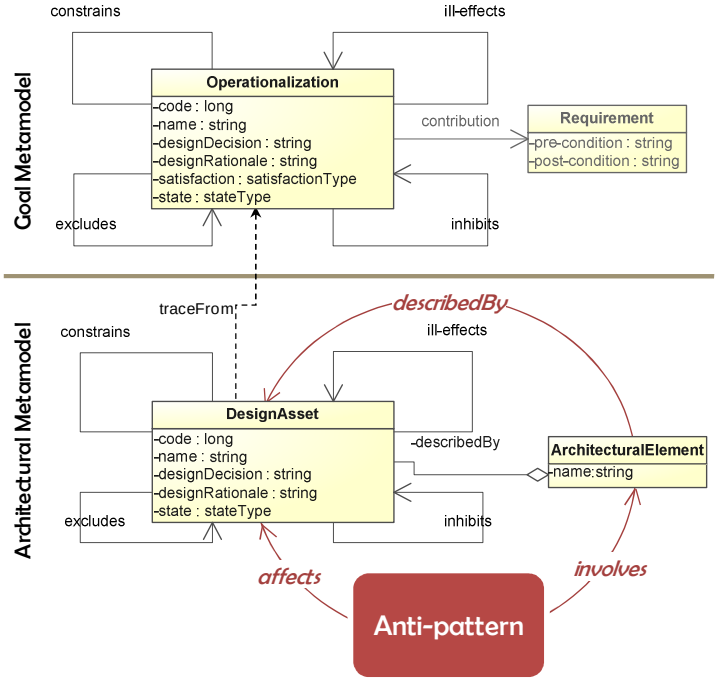



Fig. 3. Weaving antipatterns in ATRIUM Metamodels (partial view)

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 architect to describe the AK as a network. As shown in Fig. 3, these relationships were first defined on operationalizations, as they are in charge of describing both the DDs and the DRs, but they are also applicable to DesignAssets with identical semantics. An analysis was performed in [57], considering the existing relationships in other proposals, which finally led to the identification of the following relationships:

- *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.

It was shown in [57] how the selection of these relationships provides us with the necessary expressiveness to cover most of the existing approaches in the area. This analysis is summarized in Table 2. Essentially, it shows how every AKR, as proposed in the literature, can be translated or described in terms of some of these three "basic" relationships, or their combinations. Of course, semantic details are not always fully translated; but the relational description can be considered equivalent.

Table 2. Translating AKRs within ATRIUM

| Relationship | Constrains | Inhibits | Excludes | DescribedBy |
|---|---|---|---|---|
| Alternative | | Derived | | |
| Bound To | Derived | | | |
| Comprises | Derived | | | |
| Constrains | Equivalent | | | |
| Enables | | Opposite | | |
| Forbids | | | Equivalent | |
| Not Complies | Derived | Opposite | | |
| Related to | Derived | | | |
| Overrides | | Derived | | |
| Conflicts with | | Derived | | |
| ModelElementBinding | | | | Equivalent |
| Traces From/To | Derived + *Trace* | | | |
| Depends on | Derived | | | |
| Subsumes | Closure | | | |

As can be seen, *constrains* and *excludes* were included in the list. Most of the AKRs can be seen as constraints on pure dependency, and therefore, as already noted in the discussion of Table 1, they can be seen as deriving from the first one. This is the case of relationships such as *comprises*, *bound*, *related* or *not-complies* identified. The most interesting cases are *depends* (dependency is the simpler constraint) and *subsumes* (which can be formally defined as the transitive closure of a constraint).

The *traces from/to* relationship, which relates every element in the AK network with its predecessors and successors in a refinement graph, can also be considered as derived from *constrains* (the only direct dependency). In some cases, when some element is not a DD, it must be combined with *trace*, the conventional traceability relationship, which in ATRIUM is provided by the base MDD framework itself. Therefore, the *traces from* relationship is not directly included as it can be obtained by combining existing information.

The other included relationship is *inhibits*. It is perhaps more subtle, as it is intrinsically negative, but it is not a pure negation as *excludes*. This form could seem less intuitive than its negation (*enables*) but it is in fact more useful, as it expresses easily

relational concepts such as *conflicts* or *overrides*. Structurally, it also provides *alternative* branches (any choice implies inhibiting the other branch).

Fig. 3 includes another relationship, named *ill-effects*, which has been introduced in the Metamodels and, as far as we know, does not have a direct matching with other proposals, due to its different semantics. Let's consider two operationalizations, A and B, describing different design decisions. Having an *ill-effects* relationship from A to B, means that A determines that B should be analyzed because a problem has been detected in the specification. This relationship is used during the analysis of the proto-architecture whenever an antipattern is detected, as shown in section 5.3.

Fig. 3 also shows (part of) the Architectural metamodel (an extension of the PRISMA metamodel [70]) that is used to describe the proto-architecture generated as a result of the *Synthesis and transform* activity. It can be observed that every *ArchitecturalElement* is related to a set of *DesignAssets* that describe both its DDs and DRs by means of the *DescribedBy* relationship. These DesignAssets can be related by means of *constrains*, *excludes* and *inhibits* relationships in a similar way to the operationalizations in the Goal Metamodel.

We would like to point out 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 DDs and DRs that were analyzed during the specification of the system, that is, they describe all the *history* of the DDs and DRs considered during the design of the system. The latter describe the reasoning behind the *current* specification of the system, that is, why the system has its current specification. For instance, a requirement REQ_X could be related to two different operationalizations, OPE_Y and OPE_Z. Both of them would be analyzed by the architect, but only one of them, OPE_Y for instance, would be finally chosen. However, both of them would be kept in the Goal Model because they describe the reasoning carried out to evaluate which was the best alternative for the system. This is the reason why only the operationalization OPE_Y would have a trace relationship to a DesignAsset DA_Y. This DesignAsset would reflect that this was the decision made, and has a direct influence onto the current architectural specification. There would not be a DesignAsset for OPE_Z, because this choice was not made. Therefore, both kinds of entities help to maintain AK from different perspectives.

Given that the analysis of DDs and DRs is carried out during the ATRIUM Modeling Requirements activity, the re-introduction of negative relationships at the architectural level might seem confusing. However, the output of the process is a proto-architecture – that is, it must be refined in a later stage. During this refinement new relationships could be detected, thus making it necessary to provide the architect with such expressive power. In addition, the application of the MDD approach in ATRIUM allows us to trace relationships in an automatic way, by exploiting M2M transformations back to the goal model maintaining both models up-to-date. These M2M transformations provide us with another advantage, already presented in [59]. The DesignAssets are generated 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. In addition, the

necessary traceability relationships are also generated from artifacts and relationships of the Goal Model to artifacts and relations of the proto-architecture in an automatic way as was described in [56].

### 5.3 Exploiting Antipatterns in ATRIUM extension

As introduced in Section 2, the approach presented in this paper is the use of antipatterns in the process of detecting the likely negative relationships, and bad DDs. With this goal, every one of the identified concepts in Fig. 1 should be mapped on ATRIUM so that we can determine the feasibility of this approach.

Fig. 3 shows how our approach was put into practice in ATRIUM considering the goal metamodel and the architectural metamodel. In this case, ArchitecturalElement is the artifact that can be affected by an antipattern. In the architectural metamodel, the DDs are specified by means of DesignAssets which are related to the different ArchitecturalElements they have described. Our intention is to determine the *affects* relationship (see Fig. 3) – that is, to identify which ones were the DesignAssets specifying bad DDs, or which combination of them resulted in an antipattern in the proto-architecture.

In order to put our approach into practice, ATRIUM was modified by defining a new activity called *Analysis*. As can be seen in Fig. 2, this activity has two inputs: the generated proto-architecture from the previous activity and the set of antipatterns to be detected in the proto-architecture. This set can be determined according to the specific needs of the system-to be – that is, they can be antipatterns related to security, performance, etc., depending on the specific goals that were defined in the goal model. Having as input this set of antipatterns (*apSet*) and the proto-architecture to be analyzed (*am*), the following *DDChecker algorithm* is applied:

Algorithm 1 *DDChecker* algorithm

```
DDChecker
Step 1 Select the set of antipatterns apSet to be checked.
For each Antipattern ap included in apSet:
Step 2 Detect apI instances of the Antipattern ap in the
       Architectural Model am.
Step 3 For each apI instance of the Antipattern ap:
       Step 3.a Create a new DesignAsset daCreated and set
                its attributes:
                daCreated.designDecision =
                      "Modify current network of DD";
                daCreated.designRationale=
                      "Detected Antipattern"+ ap.name;
       Step 3.b For each ArchitecturalElement ae affected
                by the apI instance of the Antipattern ap:
```

i. Relate each of its DesignAssets *da* to the DesignAsset *daCreated* by means of *ill-effects* relationships:
   da.ill-effects=daCreated
ii. Set the attribute *state* of each of its DesignAssets *da* to dirty:
   da.sate=dirty

Therefore, the output of the DDChecker algorithm is a new version of the proto-architecture where the likely bad DDs are marked and the necessary relationships are included. Once the DDChecker algorithm has been applied, and as part of the activity *Analysis*, the architect evaluates the DesignAssets set to *dirty* to determine which ones are the sources of the problem. Therefore, it is the architect who ultimately takes the decision about whether the foreseen *ill-effects* of some antipattern actually occur in the software architecture description, using as input the information generated by the DDChecker algorithm and the specified DDs and DRs. We cannot consider such an algorithm as an "oracle" able to detect any conceivable antipattern, but just as a tool that the architect can use in the process of detecting likely antipatterns and their effects on the architecture. Unfortunately, this process cannot be fully automatic, as the detected antipattern could even contradict the involved DDs and DRs.

As a result of the analysis carried out, the architect modifies the network of AK by rewriting the DDs and DRs or the DesignAsset *daCreated* to describe which decision is made to avoid the problem and creating the necessary *excludes* and/or *inhibits* relationships between these DesignAssets and the *dirty* DesignAssets. Once the new network of AK is obtained, the architect modifies the architecture so that it follows the recommendations described by the network. Section 5.4 describes, by means of an example, how the Analysis activity is performed.

However, several issues should be taken into account about the DDChecker algorithm related to both how the described steps should be performed and how they are supported which are described in Sections 5.3.1, 5.3.2 and 5.3.3.

### *5.3.1 DDChecker: step 1*

It is worth noting that the set of antipatterns to be detected (apSet) is described as a list of codes that identifies the antipatterns to be detected. These codes are used as indexes to retrieve the information of the different antipatterns. They have been described in a catalogue where each antipattern is specified by means of a code, name, description, name of the file that describes the rules for its detection, and the different refactoring solutions that can be applied to eliminate or palliate the antipattern. Moreover, the antipatterns have been structured according to their kind as performance, security, etc. This facilitates the selection of the proper antipatterns framework to be used, as explained in the following section. As can be observed in the following, this catalogue has been encoded as an xml file:

```
<?xml version="1.0" encoding="ASCII"?>
<AntipatternMM:AntipatternCatalogue        xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:AntipatternMM="http://es.uclm/AntipatternMM"
xsi:schemaLocation="http://es.uclm/AntipatternMM
AntipatternMM.ecore" name="ATRIUMCatalogue">
  <kind name="Performance">
    <antipattern code="P001"
      name="Excessive Dynamic Allocation"
      description="The overhead for dynamic allocation
         increases as the number of calls increases"
      rule_file_name="excessive-dynamic-allocation.clp">
      <refactoring="To recycle objects rather than create
         new ones each time they are needed. This approach
         pre-allocates a pool of objects and stores them in
         a collection. New instances of the object are
         requested from the pool, and unneeded instances
         are returned to it. This approach is useful for
         systems that continually need many short-lived
         objects (like the call processing application).
         You pay for pre-allocating the objects at system
         initialization but reduce the run-time overhead to
         simply passing a pointer to the pre-allocated
         object."/>
      <refactoring="To share objects rather than create new
         ones."/>
    </antipattern>
…
  </kind>
…
</AntipatternMM:AntipatternCatalogue>
```

It has to be emphasized that the architect is the one who makes the final decision about what antipatterns should be checked and dealt with. However, the architect should always include in this set those antipatterns that have been already detected in previous iterations of the analysis. In this way, the architect will follow similar guidelines to those applied in testing, when regression tests are re-run to detect whether old bugs have come back and/or the new developed code collides with the previously existing code. In this case, the architect will try to detect whether the previously detected antipatterns have been eliminated and/or the introduced changes have reintroduced previously eliminated antipatterns.

Finally, another relevant issue to be considered regarding to the antipatterns selection is the importance of the order in its detection and resolution. As software design greatly depends on the requirements prioritization for its final description, the software architecture specification will depend on which order antipatterns are detected and solved. Consequently, the architect should detect and resolve first those antipatterns that are a greater threat to the system. As far as we know, it remains an open issue the classification of antipatterns regarding their impact. The only related work is that presented by Settas et al. [76], already mentioned in Section 4, that analyses synergies and divergences among project management antipatterns by using ontologies. Therefore, any proposal in this sense would be of interest to help the architect in this task.

### *5.3.2 DDChecker: step 2*

A critical issue when the DDChecker algorithm is applied is how to detect the antipattern instances. Unfortunately, there is no generic framework that can be used to detect every antipattern that could be relevant; therefore, this detection must execute all the adequate frameworks, depending on the kinds of antipatterns that the architect wants to detect. This was the reason why the antipatterns catalogue, described in Section 5.3.1, has been structured in terms of kinds of antipatterns.

In order to explain how this step can be performed, *performance* antipatterns are used in the following as an example. As aforementioned in Section 4, these are the kind of antipatterns more widely studied and analyzed in the literature. Several proposals have emerged in recent years trying to detect this kind of antipatterns, but unfortunately there is a lot still left to be explored, at least in terms of automation. Among the existing approaches, we have selected the framework called *Performance Booster* (PB) proposed by Xu [87] for several reasons. The main one is that it is a rule-based system that can be easily extended with new rules to describe new kinds of antipatterns. Moreover, in order to be selected, this framework must satisfy a pre-condition and two post-conditions. The pre-condition requires that the framework uses LQNs models as input, because of two reasons: first, LQN models are widely used for performance analysis and LQN models, and second, we have already established the necessary mappings between the architectural models. The two post-conditions establish that the framework identifies clearly the detected antipattern and which architectural elements are affected by that antipattern. PB is mainly made up of two components:

(1) *Layered Queuing Network Solver* (LQNS, [27]). This component is in charge of solving the performance model, specified as a kind of extended queueing network called a Layered Queuing Network (LQN), to extract the necessary performances measures. Xu [87] recommends the exploitation of the *LQNs models* for this goal, because they have the advantage of representing resource and bottleneck aspects of software servers, and their solution process scales up well for large systems. Moreover, other interesting approaches, such as that described by Cortellessa et al. [16], have used the LQNS to detect performance antipatterns.

(2) *Inference Engine*. This component carries out the detection of the instances of each antipattern selected in the previous step of the process (see section 5.3.1). It is worth noting that not only Xu's but also other works, such as [14], have promoted the use of a rule-based approach for the detection of antipatterns. Xu [87] recommends the use of Jess [8], because of the facilities it provides. For each antipattern defined in the catalogue, a file with its corresponding rules has been defined, that is used by Jess to carry out its detection. In this way, new antipatterns can be detected simply by defining new rules. A detailed explanation about how to define these rules is presented in [87].

Moreover, Xu exploits an additional component named PUMA [73] to transform the design model specified using UML into a LQN performance model. Therefore, we also suggest using PUMA when an UML-based design model is used, to put into practice the work presented in this paper. However, since ATRIUM has been used for this proof of concept, ATRIUM Scenario models and PRISMA Architectural models [56] have had to be used as input for the generation of the performance model. This led us to the development of the ATRIUM2LQN component by using the model-to-text transformation language, XPAND. No more details are provided about this issue, although interested readers are referred to previous work [4][42] in this area, to select the toolset more appropriate depending on their architectural model.

Therefore, when the step 2 of the DDChecker algorithm is carried out to detect the instances of performance antipatterns, the following tasks are performed:

(1) The ATRIUM Scenario model and the PRISMA Architectural model are transformed to a LQN model, using the ATRIUM2LQN component.

(2) The LQN model is used as input to the LQN Solver to simulate the execution of the system and generate the necessary performance measures related to the architectural elements.

(3) The Inference Engine is executed using as input both the results of the previous task and the name of the file where the rules to detect the antipattern *apI* have been described. It generates a list *aeSet* of ArchitecturalElements for each instance of the detected antipattern that will be used as input for the following step (Section 5.3.3).

Moreover, we would like to emphasize that there are other approaches, such as [13][15][64], which offer other alternatives for the detection of performance antipatterns. Interested readers are referred to these proposals to select the most appropriate one according to the architectural model they are using. In the following, we will not provide more details about this issue, as the specific method of detecting performance antipatterns is out of the scope of this paper.

### *5.3.3 DDChecker: step 3*

As previously mentioned, the step 3 of the DDChecker algorithm focuses mainly on the change of the architectural model by modifying the state of the DesignAssets affected by the instances of the selected Antipatterns, and establishing the necessary relationships. Therefore, for every detected instance of antipattern *apI* two main tasks are performed:

(1) A new DesignAsset, *daCreated*, is created. It is in charge of notifying that the network of AK must be modified. It lets the architect know both which DDs should be reviewed and which antipatterns they are related to. With this intent, the DD is established as "Modify current network of DD", and the related DR with the information of the detected antipattern ("Detected Antipattern" + ap.name).

(2) As Fig. 3 shows, every time an antipattern is detected, it involves a set of ArchitecturalElements whose DDs are probably the source of a problem in the proto-architecture. This is why every one of these DesignAssets should be analyzed by the architect to evaluate if they are affected by an antipattern. With this intent, the previous step described in Section 5.3.2 returns the set of ArchitecturalElements *aeSet* that can be affected by the instance of antipattern *apI* being dealt with. Every DesignAsset related to an ArchitecturalElement contained in the set *aeSet* that describes a decision is marked as *dirty* in order to advise the architect about a likely problem. These DesignAssets also have *ill-effects* relationships with the *daCreated* just created, to let the architect know that they are affected by the instance of antipattern *apI*. As all the DesignAssets affected by the same instance of the antipattern will be related to the same DesignAsset *daCreated*, the architect would be able to jointly analyze them, in order to determine which one of them or which combination is the source of the problem.

It is worth noting that the evaluation of the dirty DesignAssets can be performed not only at the architectural level but also at the requirements level. As shown in Fig. 3, DesignAsset has a *traceFrom* relationship with Operationalization. This relationship can be exploited to perform the decision process at the requirements level. This facility is very helpful, as it allows the architect to analyze *dirty* DesignAssets in the context where they were described – that is, considering the goals and requirements they are related to. With this purpose in mind, the application of M2M transformations is again a valuable resource. Similarly to the way that the proto-architecture is generated, a M2M transformation can be applied to modify the state of the operationalizations whose related DesignAssets are dirty. This is why the process of ATRIUM iterates from the *Analysis* activity to the *Define Goals* activity. However, it is out of the scope of this paper to provide more details about how these M2M transformations are specified.

### 5.4 The Gas Station Example in ATRIUM extension

Once the AKRs defined in ATRIUM has been described in section 5.2 and the process for exploiting Antipatterns in the detection of negative AKRs has been specified in section 5.3, in the following it is presented by means of an example how the proposal is put into practice. As was described in Section 4, although we can find antipatterns almost in any stage and context of software development, perhaps in the area of performance is where most researchers have focused their efforts and more formally antipatterns have been described. In terms of performance, *dynamic allocation* is perhaps one of the most expensive processes at run time. This is why several antipatterns have been proposed that try to avoid this problematic situation.

Smith and Williams identified the antipattern *excessive dynamic allocation* [80] that has been graphically illustrated in Fig. 4. It shows a typical behavior of several software systems (specially certain graphical applications) which is particularly inefficient but often remains unnoticed. To illustrate clearly the poor performance of this practice, the authors use the metaphor of a gas station. Incidentally, this is also a well-known metaphor (in a different context) within the field of software architecture [55].

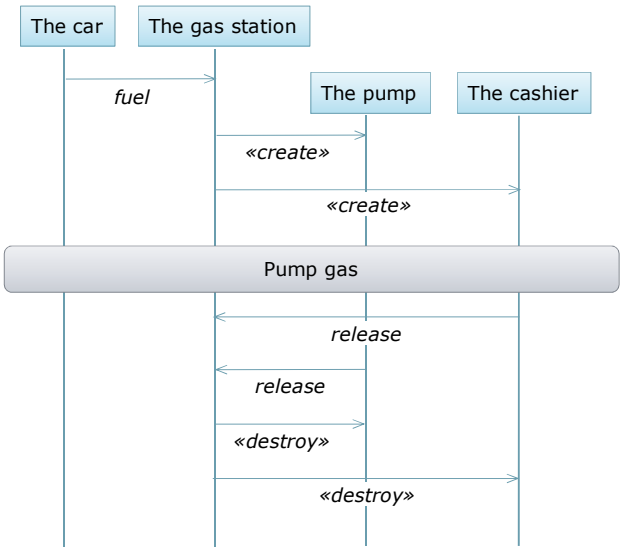



Fig. 4. Adaptation of the Excessive Dynamic Allocation Antipattern to the gas station case study

In this example, whenever a car needs gas, it pulls over to a gas station and asks for fueling. Then the gas station *creates* a pump with the gas, and a cashier to control the fueling. Once the tank is filled, the pump and the cashier notify the gas station they require to be released, and it *destroys* both of them. Obviously, this approach only works if the owner of the gas station is not worried about the money – i.e. the performance.

This simple antipattern has been detected, unfortunately, more frequently than it should be. For instance, it is frequently found in the context of Service Oriented Applications. According to our experience, especially when we have worked with junior developers, we have detected a tendency to think that resources and time are unlimited because they repeatedly create a new proxy or a new connection to the database every time a service is requested. This means the system has an increased response time as the number of requests increases, reducing the confidence of customers about the system. This is why these kinds of decisions should be detected as soon as possible.

In the following, we will use the example of the gas station to exemplify how ATRIUM has been used to put into practice the approach presented in this paper. With this aim, and according to the process described in Fig. 2, we first perform the modeling activity, having as a result a goal model where the goals, requirements and

operationalizations are described. Fig. 5 shows (part of) the goal model of a transport system. As can be observed, one of the primary goals (GOA.1) is that the car should be able to work. This goal has been refined in a requirement that establishes that the car must have a power source, being refined into two optional requirements to allow a car to have an electric engine or a gas engine. If the car has a gas engine it must be fueled, and thus, two alternative design decisions (operationalizations) could be made: first it is OPE1.2.1.1 that recommends searching for an existing gas pump and second, it is OPE1.2.1.2 that recommends building a gas pump. The last alternative was finally selected because it allows customers not to wait for a free gas pump.

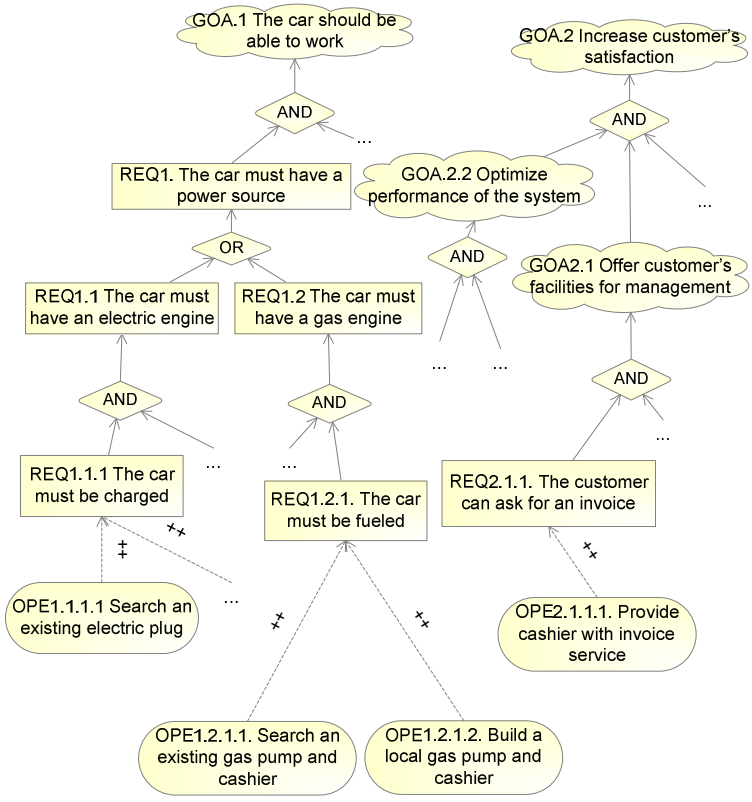



Fig. 5. ATRIUM Requirement Model (partial view)

Once the goal model has been defined, we can proceed to apply the next activity of ATRIUM, the *Modeling scenarios* activity (see Fig. 2). It establishes that we have to specify the scenarios associated to each operationalization finally selected to be applied in the final system, that is, to describe the scenario model. In this case, we could use a scenario similar to the one presented in Fig. 4, where every one of the architectural elements is identified along with its interactions.

The scenario model is used as input for the *Synthesis and Transform* ATRIUM activity to generate the proto-architecture, which does not only describe the architectural

elements but also their corresponding *describedBy* relationships with the DesignAssets that describe why they have been specified. These DesignAssets are generated as a trace from the operationalizations described in the goal model (the reader is referred to [59] for more details about how this generation is carried out). Fig. 6 shows the generated proto-architecture of the example of the gas station at run-time. Although it is not shown in the figure, *the gas station* has a relationship with the DesignAsset DA1.2.1.2, which is traced from the operationalization OPE1.2.1.2, that is, the one that establishes that a pump and a cashier should be created whenever a request is made. In a similar way, the others architectural elements generated using the scenario of the Fig. 4; that is, every pump, cashier, etc., also have this relationship.

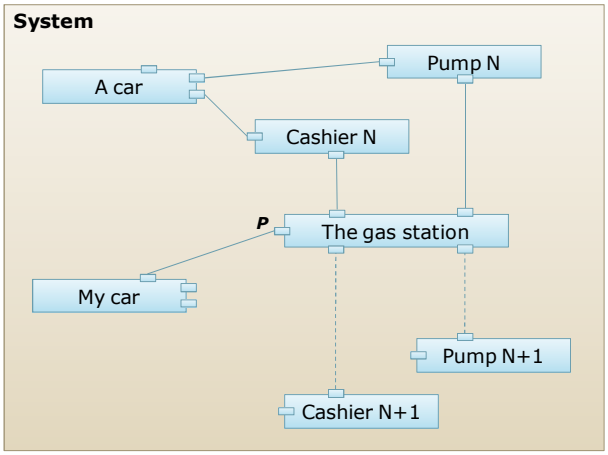



Fig. 6. The gas station system at run time

Once we have generated the proto-architecture, we can proceed with the ATRIUM *Analysis* activity. As was stated in the previous section, first we apply the *DDChecker algorithm* which is in charge of analyzing the proto-architecture to detect antipatterns, described in section 5.3. In order to run this algorithm, the first step was to select the set of antipatterns to be detected (apSet), as was stated in section 5.3.1. This set was described in an xml file, whose content is shown in the following.

```
<?xml version="1.0" encoding="UTF-8" >
<apSet>
   <antipattern code="P001" name="Excessive Dynamic
Allocation"/>
   <antipattern code="P002" name="Extensive Processing"/>
   <antipattern code="P003" name="Empty Semi Trucks"/>
   <antipattern code="P004" name="Circuitous Treasure
Hunt"/>
   <antipattern code="P005" name="Concurrent Processing
Systems"/>
```

```
   <antipattern code="P006" name="Blob"/>
   <antipattern code="P007" name="One-Lane Bridge"/>
   <antipattern code="P008" name="Ramp"/>
</apSet>
```

Regarding the example of Fig. 6, the process helped the architect to detect an excessive allocation problem, as every time a request is received on port *p*, a new *pump n+1* and a new *cashier n+1* are created instead of using some of the existing pumps and cashiers, as the one used by *a car*, that is, the system performance degrades as the number of cars gets very high. As we detected that this antipattern affects to the gas station, everyone of its related DesignAssets should be marked as dirty. This means that, as Fig. 7 illustrates, DA1.2.1.2, the one related to *the gas station* is marked as dirty (symbolized by means of forbidden sign in the figure), a new DesignAsset, DA1.2.1.3, is created to notify that the network should be modified to avoid the excessive dynamic allocation antipattern, and a ill-effects relationship is established between both DesignAssets.

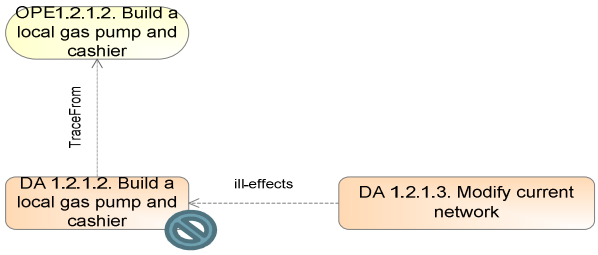



Fig. 7. Applying DDChecker on the existing DesignAsset

In the next step of the Analysis activity, the architect modifies this new network of AK to resolve the problem introduced by the antipattern. A likely solution could be the one described in Fig. 8. In this case, the architect has decided to establish a new way of managing the creation of pumps and cashiers, by providing the system with two options. First, cars must search for an existing pump whenever they need to be fueled. Second, whenever there are no free pumps for a car, a new pump will be created if their number is less than an established bound. It allows the system to fuel several cars simultaneously but avoiding being overloaded by the number of pumps. The architect also introduces an *excludes* relationship to avoid the previous decision DA1.2.1.2. As Fig. 8 shows two *constrains* relationships have been defined between DA1.2.1.3 and the DesignAssets DA1.2.1.1 and DA1.2.1.4, because whenever a new resource is created these two options have to be included in the system. An excludes relationship has been defined as well, as DA1.2.1.3 has been defined to avoid that DA1.2.1.2 is in the system as it means a problem for the specification of the system.

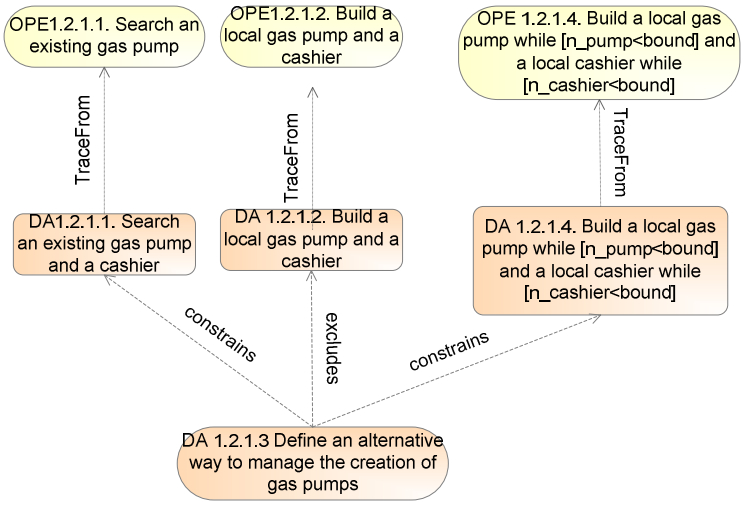



Fig. 8. Modifying the AK network

As can be observed in Fig. 8, every one of the DesignAssets is traced from an operationalization. These relationships could be exploited to perform this analysis at the requirements level, taking into account the requirements that determined the current specification of the system. These relationships, along with the new operationalizations, can be created by means of M2M transformations in an automatic way, as was described in [59].

In addition, we should point out that the identification of these new alternatives for the system will mean a new refinement of the system. This refinement will determine that the scenario model should be modified and the proto-architecture should be generated, but this issue is more related to ATRIUM itself, rather than to the goal of the approach presented here; this is why no more details are provided about it.

However, can the proto-architecture be considered free of antipatterns once it has been modified? Certainly not. This is why every time an antipattern has been detected and dealt with, the analysis should be applied again in order to determine if it has been solved and/or the modification has introduced a new antipattern. Just consider again the gas station example. Once the *DDChecker* algorithm is run again, the *unbalanced processing* antipattern [78] is detected, specifically the manifestation called *extensive processing*. This means that the modification just carried out has introduced a new antipattern.

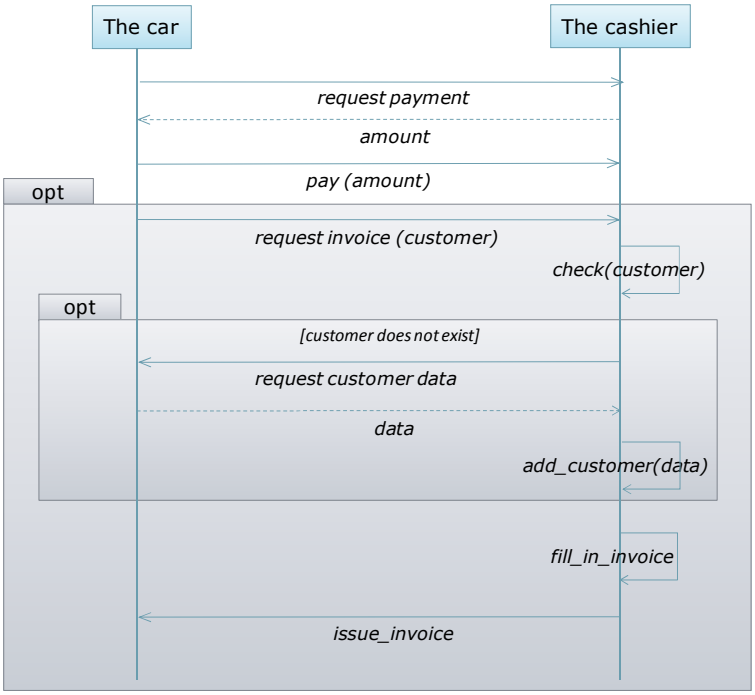


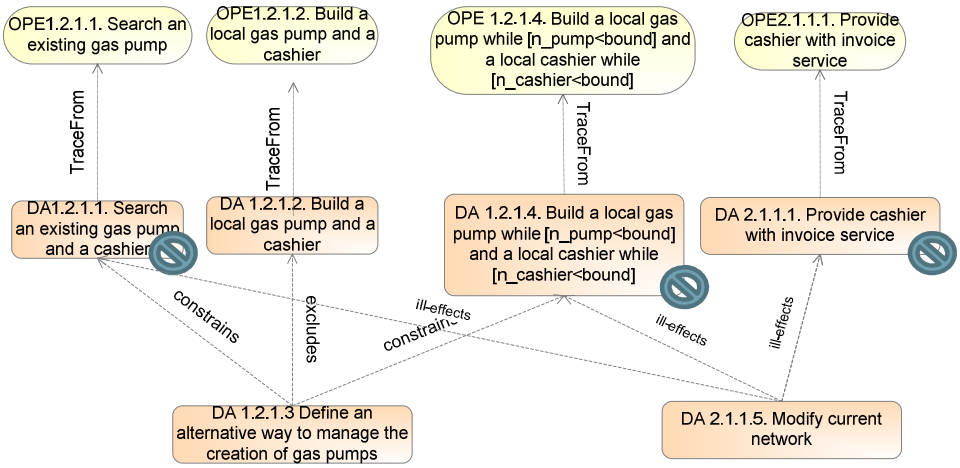

Fig. 9. Scenario to request an invoice: adaptation of the unbalanced processing antipattern

To explain where this new antipattern appears, it is better to focus on one of the requirements included in Fig. 5; specifically REQ2.1.1. “The customer can ask for an invoice”, whose operationalization is OPE2.1.1.1. “Provide cashier with invoice service”. This operationalization is related to the scenario depicted in Fig. 9, which describes how the customer and the cashier collaborate throughout the payment process, considering that he (the customer) can ask for an invoice. As can be seen, this means that a new customer must be added to the system if it does not already exist within it. Obviously, this can be a heavyweight task. Note that this situation emerged when the system was modified. The initial description of the proto-architecture was defined to create a new cashier whenever a new customer requests the service, so that the customer did not have to wait for payment. However, with the new proto-architecture description, whenever a customer requests the payment he has to wait for one of the existing cashiers in order to be attended, and in case he needs an invoice, he can monopolize that cashier for a long time, causing the other customers to wait. This is particularly problematic and disturbing, as one of the goals to be achieved is the customer satisfaction by shortening waits (see Fig. 5).

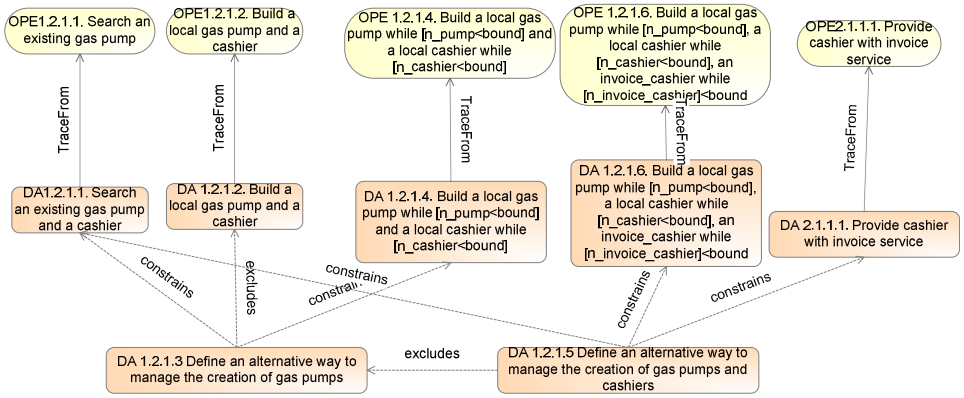
As a result of applying the DDChecker algorithm, the network of AK must be modified again. As can be observed in Fig. 10 (a) all the DA related to the cashier are marked as *dirty* except for DA1.2.1.2, that has not been marked because it is a decision that was not finally made in the system (as the *excludes* relationship and its attribute state

indicate). Thanks to the information contained in these decisions, when the architect is trying to solve the new problem, he is unlikely to make the same mistakes as before. For instance, he could decide to create a local gas pump and its associated cashier, whenever it is requested; but then he realizes that it would lead to a problem, specifically, to the *dynamic allocation* antipattern described in the DA1.2.1.3. This is exactly the reason why AK must be preserved; and here it actually serves its main purpose.

Instead of that, the architect decides to follow the recommendation of the *unbalanced* antipattern, that is, to create alternative paths for those steps that slow down the process. Specifically, the alternative was to create *two* kinds of cashiers: (1) those in charge of carrying out the payment, either in cash or money, and (2) those designed to issue invoices. This means that the proto-architecture and its network of architectural knowledge must be modified again to reflect the new decision made by the architect. As can be observed in Fig. 10 (b), DA1.2.1.5 is rewritten to describe the decision just made. It is related with DA1.2.1.3 by means of an *excludes* relationship, to specify that it is replacing this old decision, which was not finally made in the system. In addition, the architect changes the attribute state of DA1.2.1.3 to describe that this decision is not applicable to the system anymore.




(a) after DDChecker algorithm




(b) after architect evaluation

Fig. 10. Reevaluating the existing DesignAssets: (a) after DDChecker; (b) after architect evaluation

This process will be iterated over and over again until no more antipatterns are detected, or the existing ones can be considered as "acceptable" for the system development team, according to their expectations and available resources.

## 6 Validation

A very important step performed once our approach was defined was its validation by applying it to different projects. For this purpose, we selected three different projects in which we have been involved. The selected systems were the following:

- *Power plant system.* This system was initially developed to evaluate two alternative methods for deriving a software architecture specification from requirements [36]. The main goal of this system is to monitor the performance of a power plant to detect and to remedy faults in its stem condenser or its cooling circuit. This system was simple enough to validate the preliminary ideas we had.
- *MORPHEUS*. This system has been developed [58] as a novel tool that takes advantage of meta-modelling and modelling to offer flexibility and customization by providing analysts with a graphical environment for the specification and verification of the different ATRIUM models as well as the necessary transformations.
- *EFTCoR*. The proposal has been also validated in a real case study associated to the European project EFTCoR (Environmental Friendly and cost-effective Technology for Coating Removal) [29] and the national project DYNAMICA [19]. These projects aimed at designing a family of robots capable of performing maintenance operations for ship hulls. The system includes operations such as coating removal, cleaning and re-painting of the hull. Among the subsystems constituting the EFTCoR platform, our case study focused on the *Robotic Devices Control Unit* (RDCU), which interacts with other robotic devices to obtain the required information to control the different devices (positioning systems and cleaning tools) to be used for

maintenance tasks. The RDCU is in charge of commanding and controlling, in a coordinated way, the positioning of devices together with the tools attached to them. It is worth noting that the performance requirements were especially important for this system to facilitate a proper management of the robotic devices.

It is especially remarkable, as Table 3 shows, that they were selected because of the increasing complexity in terms of number of requirements provided, number of DesignAssets specified, number of components, and average number of operations provided per component.

Table 3. System Complexity of the systems used for the evaluation

| System | Requirements | DesignAssets | Components | Operations provided |
|---|---|---|---|---|
| Power plant | 39 | 52 | | 3,4 |
| Morpheus | 122 | 143 | | 22,5 |
| EFTCoR | 161 | 212 | | 14,7 |

We carried out the *Analysis* activity described in section 5 by using the specifications of these three systems. This led us to detect and solve the antipatterns identified in Table 4. This table also shows the number of detected instances of each antipattern along with the number of DAs and AKRs that had to be added. The *Analysis* activity was carried out in different iterations for each system:

Table 4. Results of the *Analysis* activity

| System | Antipattern | Instances | Added DAs | Added positive AKRs | Added negative AKRs |
|---|---|---|---|---|---|
| Power plant | - | - | - | - | - |
| Morpheus | Excessive Dynamic Allocation | 1 | 2 | 1 | 1 |
| | One-Lane Bridge | 1 | 2 | 2 | 3 |
| EFTCoR | Excessive Dynamic Allocation | 1 | 2 | 1 | 1 |
| | One-Lane Bridge | 2 | 3 | 2 | 3 |
| | Empty Semi Trucks | 4 | 5 | 4 | 6 |

- For the Power Plant system, just one iteration was carried out as no antipatterns were detected.
- For the MORPHEUS system, three iterations were performed. In the first iteration, two antipatterns were detected, and it was decided to solve the Excessive Dynamic Allocation antipattern, as both antipatterns have the same number of instances. As was already stated in section 5.4, the analysis activity was iterated over again to detect if the solution was appropriate. It was concluded that now, only the One-Lane Bridge remained in the system. This antipattern was solved, and in the final iteration it was concluded that none of the analyzed antipatterns were in the proto-architecture.

- For the EFTCoR system, four iterations were carried out in a similar way to the previous systems. The three antipatterns (Excessive Dynamic Allocation, One-Lane Bridge and Empty Semi Trucks) were detected at the first iteration. Then, similarly to the previous case, one of the antipatterns was solved in each new iteration, and it was checked again at the next iteration. Moreover, we decided to solve first, in each iteration, the antipattern with the highest number of instances. This decision was made because the number of instances is usually related to the number of components affected by the antipattern and, therefore, affected by the solution to be applied. Like in the previous system, at the end of the fourth iteration none of the analyzed antipatterns was in the proto-architecture.

It is important to note that the treatment of the detected antipatterns does not imply a notable increase of the AK networks regarding to their previous definition. Taking into account these results, it can be stated that, as a rule of thumb, the *minimum number* of DAs to be added to the network, in case of detection, will be 2: a DA to specify that an antipattern was detected and solved and an additional DA to specify the decision that solves the antipattern. We can also infer that the *maximum number* of DAs could become 1+*number of instances* of the detected antipattern, and this would only happen if the architect decides to specify a different solution for every instance of the detected antipattern. Indeed, the number of added DAs will be high only if we already had a very high number of instances. However, this would also mean that the system would have a high number of components and DAs so that, relatively, the number of new DAs would not be so significant. Moreover, we note that the results of the analyzed systems indicate that this situation is not very likely: the number of detected instances was always close to one. In this sense, we should also remark that despite the expertise of the architects who specified the analyzed systems they made errors in their specifications. Therefore, our approach helps to identify and solve them in early stages of the software development process.

The number of antipattern instances could at first seem reduced, but it has a clear justification; it should be taken into account that we are already working with software architecture specifications. When software architects have to specify large systems, they rely on *layering* techniques to manage their complexity. This implies that not only components but also their related decisions are layered, so that decisions leading to antipatterns will be usually confined to specific layers. We have noticed this situation in the analyzed systems Morpheus and EFTCoR.

We also note that a general rule can be determined regarding the number of AKRs relative to antipattern instances. The *minimum* number of AKRs will be 2: a positive and a negative AKR linking the DA that informs about the detected antipattern to the new decision and to the rejected decision, respectively. The *maximum* number of AKRs would be the number of DAs involved in each antipattern instance plus the number of antipattern instances, assuming again that a new DA is created for each instance. One might think that the number of relationships could be very high but, as noticed in Table 4,

we did not need a large number of relationships. Moreover, the use of an adequate tool that helps the architect to show and hide positive and negative relationships helps to manage properly the information added to the network.

Despite the previous arguments, it still might be questioned whether the advantages of maintaining this information is overcome by the overhead it imposes. In this sense, we consider especially relevant the results we have obtained in the analyzed projects, as it has helped us to understand why the systems were the way they were. This idea has been also emphasized by other experimental studies, such as that presented by Bratthall et al. [10]. They carried out an experiment with 17 subjects from both industry and academia, and concluded that most of the interviewed architects stated that by using AK 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 AK when they had to predict changes on unknown real-time systems. Therefore, there are compelling arguments for using the rationale while the SA is being changed.

Finally, we would like to highlight that few antipatterns were detected during the analysis of the systems. For instances, no antipatterns were detected for the Power plant system. In this sense, it is worth noting that the method we propose is not conceived for *synthezing* new architectures, but for *analyzing* those which have been already designed. This means that we cannot begin with some unstructured design, and start iterating over it, detecting antipatterns once and again until we reach a good design. In practice, this would probably result in marking every decision as *dirty*, and further progress would be inhibited. The whole system would be probably detected as the *blob* antipattern, which implies that a full redesign is required.

Instead of that, we consider that the proto-architecture to be examined is reasonably good already in the initial iteration. That is, we have tried to create or evolve a good design and we just want to check to see if there are some mistakes. In this case, every antipattern would appear almost in isolation, and therefore its detection should be relatively simple. But even then, if the system is complex and the elements are strongly related, a bad DD could affect a lot of the elements, giving the impression that "everything is dirty" again. In this kind of situation, however, those difficulties are just reflecting the actual complexity of the system; and our approach is still valid and still can be used to separate affected from unaffected elements.

Also, this is handled by means of syntactic tools, but we have not considered *semantic* aspects. It is obvious that once several elements have been marked as *dirty*, it is the architect himself who must decide, considering the semantics of the affected elements, which ones are truly important for the system's design. For instance, if we have detected a performance antipattern, we can simply ignore those dirty elements which do not affect performance. The architect could even decide that the DAs involved in the detection of an antipattern should be kept because they contribute to achieve other quality attributes more important to the system. As previously noted, this is the main reason why the process is never considered to be fully automatic. It is always the architect who makes

the final decision; the automatic support has been designed to *assist* him by exposing possible problems.

To conclude, it is important to discuss the implications of the presented approach with regard to the quality of the development process of the previous systems, and more concretely regarding to its *effectiveness*. Revising the definition provided by the ISO/IEC 25010:2011 standard[33], we consider effectiveness as the degree to which *architects* can achieve *their goals* with *accuracy* and *completeness* in the context of *software architecture specification*. Basically these goals were:

(1) *To increase defect containment*. As the analysis activity is carried out during the early stages of the development process, the number of faults that would have otherwise escaped (and would have been found during subsequent phases) was reduced. In this sense, Table 4 provides clear evidence about the number of instances of antipatterns detected and ultimately eliminated from analyzed systems. Moreover, this reduction of faults has a direct impact on the reduction of cost as the cost of eliminating defects at this early stage of the development process is much lesser than during the coding stages. Therefore, this reduction of cost soon rewards the overhead of carrying out the analysis activity.

(2) *To improve the quality of the developed software*. As the detected antipatterns were related to performance, and this is a software quality attribute, the time devoted to their proper handling has a direct impact on the quality of the developed software. As suggested in our definition, these goals were achieved with accuracy and completeness, as the analysis activity provided us with all the relevant information to eliminate the antipatterns from the analyzed systems.

## 7 Conclusions and further Research

In conclusion, there are several consequences that can be extracted from the experience and results exposed in our approach. First, the degree of sophistication which is being achieved by Architectural Knowledge management is both illustrating and increasing its interest and relevance. Second, the importance of its inner relationships – and especially *negative* ones – is becoming clearer, as it defines a complex structure of previously unrepresented information. Third, our work shows the actual usefulness of *antipatterns* in this context – they might play a central role in AK, even more important than in more traditional applications. Fourth, the use of a model-driven engineering support makes it possible to be able to deal with these complex structures, emphasizing again the interest of combining model-driven and architecture-centric approaches. In fact, though ATRIUM is just provided as a proof-of-concept, it has an interest of its own. And finally, the combination of this model-driven support and explicit architectural knowledge makes possible to go *beyond traceability*, to the extent that AK management might trigger modifications in the final architecture. Even our simple example provides a clear perspective of the usefulness and applicability of these techniques as part of the basic toolset of a software architect.

These conclusions can be examined in some more detail. *First, about the relevance of Architectural Knowledge management itself.* It has evolved from the inspiration of initial efforts in Software Architecture, to the current emphasis on capturing unrepresented design knowledge and integrating it with architecture. Our approach provides a glimpse of the complexity of the resulting AK network, showing the need for automated support; but also highlights the usefulness of this structure –beyond "simple documentation"–, which could directly influence the structure of the final architecture itself.

*Second, the influence of relationships in AK and the structure they define*. What originally was a set of small-sized design decisions is now a complex decision network. In fact, once these relationships have been included, the network of AK can get as complex as the architecture itself; comparatively, even more so as the final architecture is just a part of the rationale. Now architecting becomes what we could define as *literate architecting*: instead of simply constructing the architecture, and losing valuable contextual knowledge in every decision, the process becomes that of *writing the architectural rationale*. Two complex structures are obtained: the architecture and the rationale; but being closely intertwined, they are represented as just *one* structure. The act of building the first is also the act of writing the second.

In fact, the resulting complexity of this structure can also be considered as an issue: if the rationale (i.e. the AK network) is more complex than the architecture itself, it is legitimate to question if it has become unmanageable. In summary, to what extent can our approach scale? There are three points to consider here. First, our approach is essentially *constructive*. This means that the information is provided as the development process itself happens: decisions must be taken before being included in the network. Therefore, we are not introducing artificial complexity of any kind. Second, our approach is providing *automatic or semi-automatic support* in every step. Hence, it should be easier to handle than existing approaches, which either lose this information or have to deal with it "by hand". Moreover, our solution uses a model-driven approach, and this should make automation easier: its models have been specifically designed to be automatically processed. Their use should have, at least in theory, two different consequences: first, the performance of these models should be reasonably good, considering their origin; and second, by providing what essentially is a "neutral" core model, they should be able to interact with many existing techniques and ADLs, even serving as a *bridge* between them.

Finally, when considering the *scalability* of this solution, we should consider that it is still an architectural approach. When dealing with large-scale systems, architecture practice usually relies in *layering*. The system is described as an abstract, prescriptive architecture at the top layer, and then every component is unfolded as an increasingly concrete sub-architecture in the next layers. These layers will be maintained at every level of abstraction; hence, our design decisions will be equally layered. Therefore, the AK network will be complex as a whole, but simple enough to manage at every layer.

*The third conclusion was about using antipatterns* as the way of semi-automating the detection and the management of *negative* relationships, whose importance has also been

emphasized. Our solution, based on *antipatterns*, is able to semi-automatically detect conflicting decisions, mark then as *dirty*, and trigger an analysis which begins by marking *ill-effects* relationships, and ends with the architect proposing a modification of the proto-architecture. As far as we know, our approach is one of the first in partially automating this part of the process; it does not provide a full automation, but it does provide automated support. Also, this approach gives antipatterns a significant role in the architectural process, analogous to some extent to the one already played by patterns [82].

*The fourth conclusion is about the technological context*. Our methodology (and even the proof-of-concept provided by ATRIUM and its associated tool, MORPHEUS), is another example of the benefits of combining architecture-centric and model driven approaches. Architecture provides the capability to adapt to different scales; a model-driven perspective provides flexibility and automation. Consequently, it is quite simple to extend, either in size or scope, this hybrid approach. For instance, our preliminary work, shown in [61], already provided the basic infrastructure for AK. We had just to extend this to include relationships and antipattern detection; the initial support was not modified.

Apart from the simpler automation, and the intrinsic flexibility just mentioned, the main merit of the hybrid approach, as already emphasized in [59], is to make explicit the *traceability* relationship, which becomes the "spine" of the architectural process. MDD needs every stage in the development to be explicitly described and modeled; traceability makes it possible to connect these descriptions. Hence, we obtain the proto-architecture as the final result of a complex process, in which every decision has been recorded and can be traced forward and backwards.

Now, within the current version of the MORPHEUS environment, relationships (including traceability) are easily composed with each other, and their consequences can be fully exploited – including such diverse concerns and questions as finding the scope of a requirement, the implementation of a decision, or using finer strategies for network analysis, as mentioned below.

We consider that the work presented in this paper is the first step towards a thorough analysis that could be called *meta-analysis*. The detection of anti-patterns is just one of the tasks that this meta-analysis should carry out, but other interesting and challenging facilities should also be provided, such as exploitation of a quality model and/or a metric model [46]. For instance, consider the importance of validating whether the architecture of the different products of a Software Product Line conform to a given reference architecture [50][71]. It could be of interest to execute during the meta-analysis that another process, similar to DDChecker, would determine if the proto-architecture has any problem with regards to the reference architecture. The implications of this new facility would have to be analyzed in order to determine how the inputs provided by those DDs and DRs involved in the definition of the reference architecture could be exploited as well. We consider that this meta-analysis should be a necessity for any proposal, hopefully in the near future.

Another direction of our work includes the definition of special decisions, which could provide a better structure to describe AND/OR relationships, similar to those found at requirements level, in the goal model. As already noted in section 2, we also intend to exploit the decision network using standard techniques for network analysis [5], which would lead to the identification of special nodes, critical decisions, or even separate process development branches. Another interesting future work will be the exploitation of data mining and knowledge discovery (DMKD, [68]) to extract useful information from the decision network. We also plan to provide more sophisticated methods to visualize the information contained in the structure, and to complete this knowledge with the definition, and even local implementation, of several adequate metrics, which will be also used for our detailed analysis. In summary, the AK structure will be further enriched and analyzed, and the consequences of its use during software development will be carefully examined.

Another interesting work in progress is related to the industrial exploitation of this knowledge. Indeed the architectural knowledge generated throughout different software projects is an important and essential resource for industrial competitiveness, its suitable flow throughout organizations will facilitate that this knowledge could transferred from one project to another. Therefore, we plan to study the implications that the introduction of knowledge management systems, such as [32] or [69], could have in terms of efficiency and/or effectiveness for software development, as well as the introduction of classification algorithms, such as [43][67], that help us to rank applicable solutions to solve antipatterns.

## Acknowledgments

This work has been funded in part by the Spanish Ministry of Science and Innovation under the National R&D&I Program, within Projects DESACO TIN2008-06596-C02-01, and *Agreement Technologies*, CONSOLIDER CSD2007-0022; by the grant PEII09-0054-9581 from the Junta de Comunidades de Castilla-La Mancha; and also in part by NSF CISE SRS Grant CCF-0820251.

## References

[1] A. Akerman, J. Tyree, Using Ontology to Support Development of Software Architectures, *IBM Systems J.*, 45(4), 2006, 813-826.

[2] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Patel-Schneider, *The Description Logic Handbook* (Cambridge University Press, Cambridge, 2003).

[3] D. Ballis, A. Baruzzo, M. Comini, A Minimalist Visual Notation for Design Patterns and Antipatterns, *Proc. 5th Int. Conf. on Information Technology: New Generations*, IEEE Computer Society, Los Alamitos, 2008, pp.51-56.

[4] S. Balsamo, A. Di Marco, P. Inverardi, M. Simeoni, Model-Based Performance Prediction in Software Development: A Survey, *IEEE Transactions on Software Engineering*, 30(5) (2004) 295-310.

[5] A.L. Barabási, M. Newman, D.J. Watts. *The Structure and Dynamics of Networks* (Princeton University Press, Princeton, 2006).

[6] R. Basili, D.M. Weiss, A methodology for collecting valid software engineering data, *IEEE Transactions on Software Engineering*, 10 (6) (1984) 728–738
[7] B. Baudry, Y. Le Traon, Measuring design testability of a UML class diagram, *Information & Software Technology*, 47(13) (2005) 859-879.
[8] M. Bernardo, V. Cortellessa, M. Flamminj, TwoEagles: A Model Transformation Tool from Architectural Descriptions to Queueing Networks, *Proc. 8th European Performance Engineering Workshop*, Springer, Berlin, 2011, pp. 265-279.
[9] W. Brown, R. Malveau, H. McCormick, T. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* (John Wiley & Sons, New York, 1998).
[10] L. Bratthall, E. Johansson, and B. Regnell, Is a Design Rationale Vital when Predicting Change Impact? – A Controlled Experiment on Software, *Proc. 2nd International Conference on Product Focused Software Process Improvement*, 2000, Springer, Berlin, pp. 126-139.
[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture, Volume 1, A System of Patterns* (Wiley, Chichester, 1996).
[12] L. Chung, B. A. Nixon, E. Yu, J. Mylopoulos, *Non-Functional Requirements in Software Engineering* (Kluwer Academic Publishing, Boston, 2000).
[13] V. Cortellessa, A. Martens, R. Reussner, C. Trubiani, A Process to Effectively Identify "Guilty" Performance Antipatterns, *Proc. 13th Int. Conf., Fundamental Approaches to Software Engineering*, LNCS 6013, Springer, Berlin, pp. 368-382.
[14] V. Cortellessa, A. Di Marco, C. Trubiani, Performance Antipatterns as Logical Predicates, *Proc. 15th IEEE International Conference on Engineering of Complex Computer Systems*, IEEE Computer Society, pp. 146-156.
[15] V. Cortellessa, A. Di Marco, R. Eramo, A. Pierantonio, C. Trubiani, Approaching the model-driven generation of feedback to remove software performance flaws. *Proc. 35th Euromicro Conference on Software Engineering and Advanced Applications*, IEEE Press, New York, 2009, pp. 162–169.
[16] V. Cortellessa, L. Frittella, A Framework for Automated Generation of Architectural Feedback from Software Performance Analysis, *Proc. 4th European Performance Engineering Workshop*, Springer, Berlin, 2007, pp. 171-1.
[17] K. Czarnecki, S. Helsen, Classification of Model Transformation Approaches, *IBM Systems Journal*, 45(3) (2006) 621-645.
[18] J. Cubo, C. Canal, E. Pimentel, Context-Aware Composition and Adaptation based on Model Transformation, *J. UCS*, 17(5) (2011) 777-806.
[19] CICYT TIC2003-07804-C05-01, DYNAMICA, DYNamic and Aspect-Oriented Modeling for Integrated Component-based Architectures, 2003- 2006.
[20] A. Dardenne, A. van Lamsweerde, S. Fickas, "Goal-directed Requirements Acquisition," *Science of Computer Programming*, 20(1-2) (1993) 3-50.
[21] J. Dietrich, N. Jones, J. Wright, Using social networking and semantic web technology in software engineering – Use cases, patterns, and a case study, *Journal of Systems and Software*, 81(12) (2008) 2183-2193.
[22] M. El-Attar, J. Miller, Matching Antipatterns to Improve the Quality of Use Case Models, *Proc. 14th IEEE Int. Requirements Engineering Conference*, IEEE Computer Society, Los Alamitos, pp. 99-108.
[23] A. Erfanian, F.S. Aliee, An Ontology-Driven Software Architecture Evaluation Method, *Proc. Workshop Sharing and Reusing Architectural Knowledge*, ACM Computing, ACM New York, 2008, pp. 79-86.
[24] D. Falessi, G. Cantone, P. Kruchten, Value-Based Design Decision Rationale Documentation: Principles and Empirical Feasibility Study, *Proc. 7th Working IEEE/IFIP Conf. Software Architecture*, IEEE Computer Society, Los Alamitos, 2008, pp. 189-198.
[25] R. Farenhorst, R.C de Boer, *Core Concepts of an Ontology of Architectural Design Decisions*, Technical Report IR-IMSE-002, Dept. Computer Science, Vrije Universiteit Amsterdam, 2006.

[26] G. Franks, *Performance analysis of distributed server systems, Ph.D. Thesis*, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 2000.
[27] E. Friedman-Hill, *Jess in Action, Rule-Based Systems in Java*, Manning Publications, 2003.
[28] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison Wesley, Boston, 1993).
[29] GROWTH G3RD-CT-00794: EFTCOR: Environmental Friendly and cost-effective Technology for Coating Removal. European Project, 5th Framework Program, 2003.
[30] N.B. Harrison, P. Avgeriou, U. Zdun, Using Patterns to Capture Architectural Decisions, *IEEE Software, 24(4)* (2007) 38-45.
[31] S.A. Hendrickson, S. Subramanian, A. van der Hoek, Multi-Tiered Design Rationale for Change Set Based Product Line Architectures, *Proc. 3rd Work. Sharing and Reusing Architectural Knowledge*, ACM Computing, ACM New York, 2008, pp. 41-44.
[32] C. Hirai, Y. Uchida, T. Fujinami, A Knowledge Management System for Dynamic Organizational Knowledge Circulation, *International Journal of Information Technology and Decision Making*, 6(3) (2007) 509-522.
[33] ISO/IEC ISO/IEC 25010:2011, Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models, 2011.
[34] A. Jansen, J. Bosch, Software Architecture as a Set of Architectural Design Decisions, *Proc 5th Working IEEE/IFIP Conf. Software Architecture*, IEEE Computer Society, Los Alamitos, 2005, pp. 109-120.
[35] A. Jansen, J. Bosch, P. Avgeriou, Documenting After the Fact: Recovering Architectural Design Decisions, *Journal of Systems and Software*, 81(4) (2008) 536-557.
[36] D. Jani, D. Vanderveken, D. Perry, Deriving Architecture Specifications from KAOS Specifications: a Research Case Study, *Proc. 2nd European Conference on Software Architecture*, Springer, Berlin, 2005, pp. 185-202.
[37] M. Kis, Information Security Antipatterns in Software Requirements Engineering, *Proc. 9th Conference on Pattern Language of Programs*, 2002.
[38] F. Khomh, S. Vaucher, Y.G. Guéhéneuc, H. A. Sahraoui, BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, 84(4) (2011) 559-572.
[39] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, *Empirical Software Engineering*, 17(3) (2012) 243-275.
[40] S. Kim, D. K. Kim, L. Lu, S. Park, Quality-driven architecture development using architectural tactics, *The Journal of Systems and Software*, 82 (2009) 1211–1231.
[41] P. Könemann, O. Zimmermann: Linking Design Decisions to Design Models in Model-Based Software Development, *Proc. 4th European Conference Software Architecture*, Springer, Berlin, 2010, pp. 246-262.
[42] H. Koziolek, Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8) (2010) 634-658.
[43] G. Kou, Y. Lu, Y. Peng, Y. Shi, Evaluation of Classification Algorithms Using MCDM and Rank Correlation, *International Journal of Information Technology and Decision Making*, 11(1) (2012) 197-225.
[44] P. Kruchten, An Ontology of Architectural Design Decisions, *Proc. 2nd Workshop of Soft. Variability Man., Groningen*, 2004, pp. 54-61.
[45] P. Kruchten, P. Lago, H. van Vliet, Building Up and Reasoning About Architectural Knowledge, *Proc. 2nd Intl. Conf. Quality of Software Architectures*, LNCS 4214, Springer, Berlin, 2006, pp. 43-58.
[46] O. Lamouchi, A. Ramdane-Cherif, N. Lévy, Evaluation Approach for Software Architecture, *Proc. International Conference on Software Engineering Research & Practice*, CSREA Press, 2009, pp. 320-326.

[47] A van Lamsweerde, Elaborating Security Requirements by Construction of Intentional Anti-Models, *Proc. 26th Int. Conf. on Software Engineering*, IEEE Comp. Society, Los Alamitos, 2004, pp.148-157
[48] T. Lenin Babu, M. Seetha Ramaiah, T.V. Prabhakar, D. Rambabu, ArchVoc – Towards an Ontology for Software Architecture, *Proc. 2nd Workshop Sharing and Reusing Architectural Knowledge*, IEEE Computer Society, Los Alamitos, 2007, pp. 5.
[49] A. Martens, and H. Koziolek, Performance-oriented Design Space Exploration, Components in a World of Mobile and Distributed Computing, *Proc. 30th Int. Work. on Component-Oriented Programming*, 2008.
[50] T. Mikkonen, A. Salminen: Towards a Reference Architecture for Mashups. *Proc. OTM Workshops: Workshop on Variability, Adaptation and Dynamism in Software Systems and Services*, 2011 pp. 647-656
[51] N. Moha, Y.-G. Guéhéneuc, On the Automatic Detection and Correction of Software Architectural Defects in Object-Oriented Designs, *Proc. 6th ECOOP Work. on Object-Oriented Reengineering*, 2005.
[52] F. Montero, E. Navarro, ATRIUM: Software Architecture Driven by Requirements, *Proc. 14th IEEE International Conference on Engineering of Complex Computer Systems*, IEEE Computer Society, Los Alamitos, 2009, pp.230-239.
[53] T.P. Moran, J.M. Carroll. Design Rationale: Concepts, Techniques and Use. Routledge, 1996.
[54] A. Mos, J. Murphy, Performance management in component-oriented systems using a Model Driven Architecture™ approach, *Proc. 6th Int. Enterprise Distributed Object Computing Conf.*, IEEE Computer Society, Los Alamitos, 2002, pp. 227-237.
[55] G. Naumovich, G.S. Avrunin, L.A. Clarke, L.J. Osterweil, Applying Static Analysis to Software Architectures, *Proc. 6th European Software Engineering Conference Software Engineering / 5th ACM SIGSOFT Symposium on Foundations of Software Engineering*, ACM Computing, ACM New York, 1997, pp. 77-93.
[56] E. Navarro, C. E. Cuesta, D. E. Perry, C. Roda, Using Model Transformation Techniques for the Superimposition of Architectural Styles, *Proc. 5th European Conference on Software Architecture*, LNCS 6903, Springer, Berlin, 2011, pp. 379–387.
[57] E. Navarro, C. E. Cuesta, D. E. Perry, Weaving a Network of Architectural Knowledge, *Proc. Joint Working IEEE/IFIP Conf. on Software Architecture 2009 & European Conference on Software Architecture*, IEEE Computer Society, Los Alamitos, 2009, pp. 241-244.
[58] E. Navarro, A. Gómez, P. Letelier, I. Ramos, MORPHEUS: a supporting tool for MDD, *Proc. 18th International Conference on Information Systems Development*, Springer, Berlin, 2009, pp. 255-267.
[59] E. Navarro, C. E. Cuesta, Automating the Trace of Architectural Design Decisions and Rationales Using a MDD Approach, *Proc. 2nd European Conference Software Architecture*, LNCS 5292, Springer, Berlin, 2008, pp. 114-130.
[60] E. Navarro, P. Letelier, I. Ramos, Requirements and Scenarios: playing Aspect Oriented Software Architectures, *Proc. 6th IEEE/IFIP Conf. on Software Architecture*, IEEE Computer Society, Los Alamitos, 2007, n. 23.
[61] E. Navarro, P. Letelier, J. Jaén, I. Ramos, Supporting the Automatic Generation of Proto-Architectures, *Proc. 1st European Conf. on Software Architecture*, LNCS 4758, Springer Verlag, Heidelberg, 2007, pp. 43-58, (Best poster award).
[62] OMG, Software Process Engineering Metamodel (SPEM), Version 1.1 formal/05-01-06, 2005.
[63] OMG document ptc/05-11-01, QVT, MOF Query/ Views/Transformations. Final adopted spec., 2005.
[64] T. Parsons, J. Murphy, Detecting performance antipatterns in component based enterprise systems, *Journal of Object Technology* 7(3) (2008) 55–90.
[65] O. Pastor, J. C. Molina, *Model-driven architecture in practice - a software production environment based on conceptual modeling* (Springer, Berlin, 2007).

[66] Y. Peng, G. Kou, G. Wang, W. Wu, Y. Shi, Ensemble of software defect predictors: An AHP-based evaluation method, *International Journal of Information Technology and Decision Making*, 10(1) (2011) 187-206.
[67] Y. Peng, G. Kou, G. Wang, H. Wang, F. I. S. Ko, Empirical Evaluation of Classifiers for Software Risk Management. *International Journal of Information Technology and Decision Making*, 8(4) (2009) 749-767.
[68] Y. Peng, G. Kou, Y. Shi, Z. Chen, A Descriptive Framework for the Field of Data Mining and Knowledge Discovery, *International Journal of Information Technology and Decision Making*, 7(4) (2008) 639-682.
[69] P. Peng-Kiat, K. L. Poh, Making decisions in an intelligent tutoring system, *International Journal of Information Technology and Decision Making*, 4(2) (2005) 207-233.
[70] J. Pérez, N. Ali, J. A. Carsí, I. Ramos, Designing Software Architectures with an Aspect-Oriented Architecture Description Language, *Proc. 3rd European Workshop on Soft. Architecture*, LNCS 4063, Springer, Berlin, 2006, pp. 123-138.
[71] D. E. Perry, Generic Architecture Descriptions for Product Lines, *Proc. Development and Evolution of Software Architectures for Product Families*, LNCS, 1429, Springer, Berlin, 1998, pp. 51-56.
[72] D. E. Perry and Alexander L Wolf, Foundations for the Study of Software Architecture, *ACM SIGSOFT Software Engineering Notes*, 17(4) (1992) 40-52.
[73] D. B. Petriu, C. M. Woodside, Software performance models from system scenarios, *Performance Evaluation*, 61(1) (2005) 65-89.
[74] B. Selic. The Pragmatics of Model-Driven Development, *IEEE Software*, 20(5) (2003) 19-25.
[75] D. Settas, A. Cerone, S. F., Enhancing ontology-based antipattern detection using Bayesian networks, *Expert Systems with Applications*, 39(10) (2012) 9041-9053.
[76] D. Settas, G. Meditskos, I. Stamelos, N. Bassiliades, SPARSE: A symptom-based antipattern retrieval knowledge-based system using Semantic Web technologies, *Expert Systems with Applications*, 38(6) (2011) 7633-7646.
[77] M. Sinnema, S. Deelstra, Classifying Variability Modeling Techniques, *Journal on Information and Software Technology*, 49(7) (2007) 717-739.
[78] C. U. Smith, L. G. Williams, New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot, *Proc. 28th Int. Computer Measurement Group Conf.*, Computer Measurement Group, Reno, 2002, pp. 667-674
[79] C. U. Smith, L. G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software* (Addison-Wesley, 2001).
[80] C. U. Smith, L. G. Williams: Software Performance AntiPatterns; Common Performance Problems and their Solutions, *Proc. 27th Int. Computer Measurement Group Conf.*, 2001, pp. 797-806.
[81] I. Stamelos, Software project management anti-patterns, *Journal of Systems and Software* 83(1) (2010) 52-59
[82] A. Tang, J. Han, Architecture Rationalization: A Methodology for Architecture Verifiability, Traceability and Completeness, *Proc. 12th IEEE Intl. Conf. Engineering of Computer Based Systems*, IEEE Computer Society, Los Alamitos, 2005, pp. 135-144.
[83] A. Tang, Y. Jin, J. Han, A. Nicholson, Predicting Change Impact in Architecture Design with Bayesian Belief Networks, *Proc. 5th Work. IEEE/IFIP Conf. Software Architecture*, IEEE Computer Society, Los Alamitos, 2005, pp. 67-76.
[84] M. T. Su, Capturing exploration to improve software architecture documentation. *Proc. 4th European Conference Software Architecture, Companion Volume*, ACM Computing, ACM New York, 2010, pp. 17-21.
[85] C. Trubiani, A. Koziolek, Detection and solution of software performance antipatterns in palladio architectural models, *Proc. Second Joint WOSP/SIPEW International Conference on Performance Engineering*, 2011, pp. 19-30.
[86] J. Tyree, A. Akerman, Architecture Decisions: Demystifying Architecture, *IEEE Software* 22(2) (2005) 19-27.

[87] J. Xu, Rule-based automatic software performance diagnosis and improvement, *Performance Evaluation*, 67(8) (2010) 585-611.
[88] L. Zhu, I. Gorton, UML Profiles for Design Decisions and Non-Functional Requirements, *Proc. 2nd Workshop Sharing and Reusing Architectural Knowledge*, IEEE CS, Los Alamitos, 2007, pp. 8.