Modularizing Early Architectural Assumptions in Scenario-Based Requirements

Dimitri Van Landuyt and Wouter Joosen

iMinds-DistriNet, KU Leuven
Celestijnenlaan 200A,
B-3001 Leuven, Belgium
{dimitri.vanlanduyt,wouter.joosen}@cs.kuleuven.be

Abstract. Early architectural assumptions (EAAs) are initial assumptions about the architectural solution that are made already during requirements elicitation. Such EAAs are inherently present when applying requirements engineering methods and techniques situated at the transition to architecture, for example those adhering to the Twin Peaks model to software engineering.

In the current state-of-the-art, early architectural assumptions (EAAs) are documented implicitly, and they are tangled within and scattered across heterogeneous requirement artifacts. This makes it hard to distinguish EAAs from actual requirements, analyze their relevance, and bring them in relation to architectural decisions taken in later development stages. As a consequence, early development activities in the transition to architecture are hindered by the lack of explicit support for EAAs.

In this paper, we address this problem in the context of scenario-based requirements (use cases and quality attribute scenarios). We present a system meta-model for EAAs, and provide an aspect-oriented requirements language that allows the instantiation of EAAs in terms of use case-level pointcuts. We employ our prototype implementation of above-mentioned techniques to evaluate and illustrate the benefits of making EAAs explicit in the early stages of development, specifically in terms of modularity and requirements navigability.

1 Introduction

It is widely accepted that in practical software development, there is seldomly a clear-cut dichotomy between requirements engineering and architectural design, but that, in order to achieve an easier transition and faster convergence to the final architecture, these activities are often intertwined. This is called the Twin Peaks model to software engineering [18].

One consequence of this is that requirements —which exist in the problem space– are inherently affected or based upon initial architectural assumptions — solution-space elements. These early assumptions are architectural in nature, in the sense that they exist in different architectural views: some assumptions are about the structural decomposition of the architecture, some about the system's behavior, some about the deployment configuration, etc. It is in the intrinsic

S. Gnesi and A. Rensink (Eds.): FASE 2014, LNCS 8411, pp. 170–184, 2014.

nature of software engineering activities in the transition to architecture that they rely on some early-defined, pre-assumed system abstractions. We call these *early architectural assumptions* (EAAs). In earlier work [29], we have argued that making such early assumptions is indeed an effective technique to converge more quickly to a suitable architectural solution.

In the current state-of-the-art, early architectural assumptions (EAAs) are documented implicitly. Specifically in the context of scenario-based requirements, they are embedded in natural language descriptions, and it takes substantial analytic effort to (i) identify and understand them, (ii) distinguish them from actual stakeholder requirements or technical constraints, and (iii) understand their often subtle interactions. Furthermore, we have seen that EAAs have a crosscutting impact on the different requirements: they are scattered across scenarios and tangled with each other within these scenarios. This inherent characteristic of EAAs has been pointed out by Nusebeih who argues that crosscutting influences between requirements can mostly be observed "*when elements of a solution also begin to be explored*" [19].

For these reasons, implicit early architectural assumptions have a negative impact on the development activities at the transition from requirements to architecture. Specifically, activities involving change impact analysis and trade-off analysis such as the Architecture Trade-off Analysis Method (ATAM) [12]) and Attribute-Driven Design (ADD) [32,2] are hindered by badly modularized and implicit EAAs [30] as they increase the required analytic effort to understand and manipulate the requirement body.

In this paper, we address these problems in the context of scenario-based requirements (use cases [9] and quality attribute scenarios [2]). Specifically, we create an EAA model which acts as a knowledge repository of explicit early-defined system abstractions. This EAA model is promoted as a key enabler to achieve a smooth transition to architecture. To increase the robustness and expressivity of EAA definitions and to address their crosscutting nature, we introduce and employ an expressive use case-level pointcut language called AspectU+.

Additionally, we present prototype implementation of these techniques, and we leverage this prototype to evaluate the presented solutions. Specifically, we present detailed development scenarios from a realistic case study, a car crash management system (CMS) [13,11], and we focus on illustrating the benefits of having an explicit EAA model in the transition to architecture.

This paper is structured as follows: Sect. 2 presents the problem statement. Subsequently, Sect. 3 presents a set of techniques for modularizing EAAs in the context of scenario-based requirements. Then, Sect. 4 evaluates our results. Sect. 5 discusses related work and finally, Sect. 6 concludes this paper.

2 Problem Statement

We motivate our work in the context of a realistic case study, a crisis management system (CMS) for resolving car crashes (e.g. by dispatching tow trucks, ambulance, rescue workers, ...), the requirements of which are scenario-based in

nature. QAS 1 presents a (shortened) performance quality attribute scenario related to the calculation or revision of car crash resolution strategies. Specifically, it prescribes how the system should react in the case when too much new information arrives, giving rise to many strategy (re-)calculations in parallel; i.e. by shifting some of the calculation requests to a replica instance of the strategy calculation component.

QAS 1 Performance. Strategy calculation and revision

[..] the rate of newly-arriving information exceeds the throughput of strategy revisions and initial strategy calculations, especially in the case that many strategy calculations are executed in parallel, for example in case of multiple related accidents (e.g. a pile-up).

- **Source:** Witness, Service Provider or Information Service
- **Stimulus**: New car crashes reported by Witness, new information received from the Information Service or Service Provider.
- **Artifact:** (sub-)system responsible for strategy (re-)calculation
- **Environment:** Normal execution mode.
- **Response:** 'Overload' mode: requests are forwarded to a next instance of the sub-system:
  - when processing new information, the CMS assesses whether this new information gives rise to a strategy revision. In overload mode, the request is sent to the replica instance.
  - when entering a new dossier, the initial strategy is selected by the Coordinator. In overload mode, the request is sent to the replica instance.
- **Response Measure:**
  - the system goes into 'overload' mode when the response time > 500 ms for a given strategy (re-)calculation job.

Upon analysis, it is clear that this scenario has been written with a number of early architectural assumptions (EAAs) in mind: (i) the algorithms used to calculate strategies will not be light-weight (and thus it is realistic to imagine a situation of system overload); (ii) initial strategy calculation and strategy revision will be done by one and the same component; (iii) this component will be easily replicated, and there will at least be two replicas in total; and (iv) the system or its middleware will have the ability to monitor the throughput of strategy calculation jobs, and redirect requests dynamically.

These assumptions are fundamentally different from other requirement artifacts such as use cases, domain models, glossaries, and most importantly, they are architectural in nature: the first assumption is about the performance characteristics of the selected algorithms while the second is an assumption about the structural decomposition of the system. The third is an assumption about the deployment configuration of the CMS, while the fourth is about the selected technologies and platforms on which the CMS is to be built.

Due to the inherently implicit nature of these EAAs (embedded in requirements), it is hard to assess (without additional analysis) (i) that these are in fact assumptions and not constraints or requirements imposed by the stakeholders, (ii) whether or not these EAAs are made after thoughtful (architectural) analysis, and if so, (iii) what the underlying rationale was, and (iv) whether these

EAAs are contradicted or reinforced by other assumptions (perhaps documented in different scenarios).

The software architect —when handed these requirements— must assess whether these EAAs are desirable, realistic and technically feasible. Perhaps the architect decides to reject some assumptions, and this in turn might lead to consistency problems, as it is unclear what the impact of that decision might be on the other requirements, on the other architectural assumptions and decisions. This is worsened by the large number of inherent yet subtle interdependencies between (the EAAs and) the different architectural drivers, often crosscutting in nature. To illustrate this, we extend the example of QAS 1 with the *"Revise current strategy"* use case, which involves the revision of a currently executing car crash resolution strategy (presented in Use Case 1). When comparing the description of the strategy revision algorithm in step 2 of Use Case 1 with QAS 1, it becomes clear that some of the assumptions made during the creation of QAS 1 have been influenced directly by this use case (e.g. the first EAA highlighted above about the complexity of the strategy calculation and revision algorithms).

Use Case 1. The '*Revise current strategy*' use case

- **Id/name:** Revise current strategy
- **Primary actor:** Coordinator, CMS
- **Basic Flow:**
  1. The Coordinator indicates that he wants to revise the current strategy for an ongoing car crash [..] by selecting an ongoing car crash dossier.
  2. The CMS calculates and proposes a number of car crash resolution strategies and indicates for each of these strategies, the price, the expected time duration and the risks associated with these. The CMS takes into account possible dependencies between individual missions, missions that have already been executed, the availability of the external service providers and emergency services already at the scene.
  3. The Coordinator selects the desired strategy.
  4. The CMS registers the revised strategy in the car crash dossier.
  5. The CMS informs the involved external service providers [..] about their new or updated missions.
- **Alternative Scenario:**

  3B. The CMS only comes up with one strategy, continue with step 4.

Problem Statement. A more detailed problem statement has been derived and presented in earlier work [29]. This problem statement is summarized below:

1. EAAs are documented **implicitly** in scenario-based requirements;
2. EAA definitions are scattered across scenario-based requirements and tangled with other EAA definitions: **bad modularity** of EAAs;
3. EAAs typically have a **crosscutting** influence on other requirements.

3 Modularizing EAAs in Scenario-Based Requirements

In this section, we provide a set of techniques to modularize early architectural assumptions (EAAs) in the context of scenario-based requirements (use cases

and quality attribute scenarios). Sect. 3.1 presents our system meta-model for expressing EAAs. Then, Sect. 3.2 discusses our technique to instantiate EAAs in terms of use-case level pointcuts. Subsequently, Sect. 3.3 describes the role of EAAs in the quality attribute scenario authoring process.

3.1 System Meta-model for EAAs

As the goal is to define a system meta-model suitable for documenting EAAs in quality attribute scenarios, we have analyzed the quality attribute scenario creation guidelines presented in [2]. Specifically, we selected the most common system concepts used in quality attribute scenarios and we removed synonyms, in order to keep the resulting meta-model minimal and light-weight. The result is presented in Fig. 1.

EAAs in quality attribute scenarios typically refer to functional requirements and fragments of functionality, which are in our case described as use cases. To capture and document these semantic interdependencies, we instantiate EAAs in terms of use case steps (or collections thereof). As use case modeling in general aligns best to a behavioral architectural view, the key behavioral concepts (`Message`, `Request`, `Response` and `Event`) are instantiated directly from use case steps. Based on these elements, the remaining concepts (`Interaction`, `Function`, `Subsystem`, `Channel` and `Node`) can be derived. Table 1 below provides a detailed description of the meta-model concepts and provides syntax to instantiate EAAs in terms of use case (fragments).

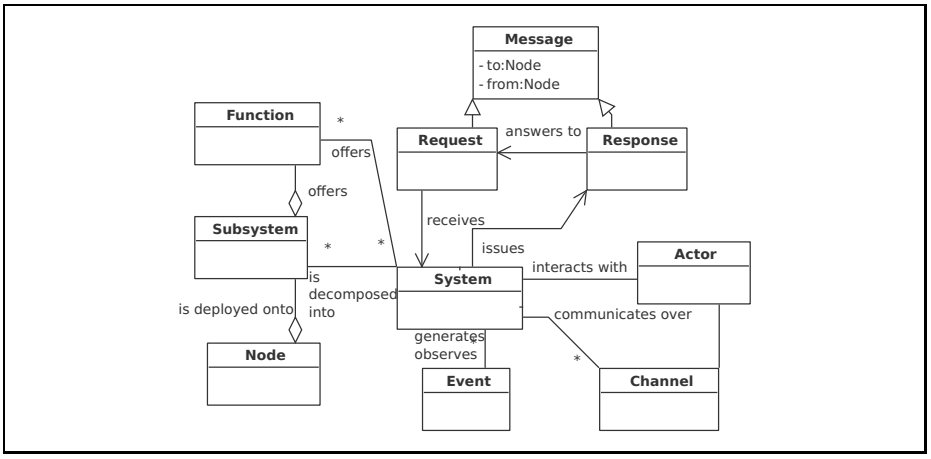


Fig. 1. The meta-model for expressing early architectural assumptions (EAAs)

Table 1. Detailed description of the meta-model concepts and Java-like constructor syntax for instantiating EAAs

Message. In general, every use case step corresponds to a message. It represents a trigger or an action relevant to the system. A message is always directed, from a one `Subsystem` to another. The following syntax is used to instantiate messages:

```
Message(UseCaseStep step);
```

Request and Response. Conceptually, requests and responses are refinements of the more generic message. The messages sent by any actor other than the system are considered requests, while the use case steps directed from the system are called responses. The constructor syntax to instantiate these elements is as follows:

```
Request(UseCaseStep step); // pre: (!step.getActor().getName().equals("System"))
Response(UseCaseStep step); // pre: (step.getActor().getName().equals("System"))
```

Event. In essence, an event is a certain path through a use case, or a subset thereof. The necessary precondition is that an event consists of consecutive use case steps. The constructor syntax for instantiating an `Event` is presented below:

```
Event(UseCaseStep[] steps); // pre: (steps.size()>1 and steps are subsequent)
```

Interaction. An interaction is defined as a single couple of consecutive `Request-Response`s instances. Therefore, the constructor for an `Interaction` is as follows:

```
Interaction(Request req, Response resp); // pre: resp.answersTo(req)
```

Function. The concept of a function overlaps slightly with that of an `Event`, in the sense that a function realizes or executes a certain `Event` path. It represents how the system reacts or behaves, given a certain trigger or request. To instantiate a concrete function, the following syntax is used:

```
Function(Event event); // pre: event starts with a Request, and the consecutive steps
conceptually belong together
```

Subsystem. Similarly, a subsystem is defined by referring to the `Function`(s) offered by it. It is possible to formulate the assumption that several related `Function`s are offered by one single `Subsystem`, and therefore to instantiate a specific subsystem, the following syntax is offered:

```
Subsystem(Function[] functions);
```

Channel. Channels are defined by one or more interactions between the system and one or more actors. Potentially, many different interactions may occur over a single channel, or separate channels (e.g. a control channel or a data channel). It is possible to merge several interactions over one channel. To instantiate a `Channel`, the following syntax is offered:

```
Channel(Interaction[] interaction); // pre: the same actors are involved in each
interaction
```

Node. Finally, a node represents a physical machine. We refer to a node by referring to one or more subsystems deployed to it. `Node`s are instantiated as follows:

```
Node(Subsystem[] subsystems); // pre: subsystems.size >=1
```

3.2 EAA Instantiation in Terms of Use Case-Level Pointcuts

A straightforward strategy to instantiate the early architectural assumptions (EAAs) would involve referring directly to specific use case steps. However, as this would introduce tight coupling between the EAA model and the use case

model (due to their often crosscutting interrelations), this strategy would not adequately address Problem #3 defined in Sect. 2. Therefore, we present an alternative instantiation strategy that involves referring to use case steps (or collections thereof) by means of pointcut expressions.

First, we introduce a use case-level pointcut specification language (PSL) called AspectU+. This PSL allows capturing the crosscutting nature of the EAAs in a more expressive manner. Then, we define some EAAs from the running example of Sect. 2 in terms of AspectU+ pointcuts.

AspectU+. Sillito et al. [24] have already proposed a use case-level PSL called AspectU, which is primarily meant for composition of functional aspects to use case models. However, this PSL is limited in the sense that it does not fully exploit the semantics and structural conventions behind use case modeling. For example, in AspectU it is not possible to match all steps in which a certain actor is involved, or the set of responses to a certain step (which includes main steps and steps from alternative scenarios). Nonetheless, these are useful constructs for defining EAAs.

Table 2. AspectU+ primitive pointcuts

Primitive pointcut	**description**
`usecase(`*ucname*`)`	selects all use case steps from the use cases whose names match *ucname* (incl. alternate scenarios)
`main(`*ucname*`)`	selects only the use case steps from the main scenario of the use cases whose names match *ucname*
`extension(`*ucname*`)`	selects only the use case steps from the alternate or extension scenarios of the use cases whose names match *ucname*)
`steps(`*id*`)`	selects specific steps that match the *id* expression
`actorsteps(`*id*`)`	selects the use case steps that are performed by the actors whose names match *id*
`responses(`*steps*`)`	selects the use case steps in response to the steps in *steps*
operators: `and` (`&&`), `or` (`||`), `not` (`!`)	(*prefix* notation)
define: `name` := *expression*	

Therefore, we have extended the AspectU PSL to serve our needs[1], and we call the resulting language AspectU+. Note that pointcut expressions specified in AspectU remain fully compatible to pointcut expressions in AspectU+, but not vice versa. Table 2 presents the syntax of the AspectU+ language. The rows of this table that are colored in grey present our extension to AspectU [24]. String matching in AspectU+ is done by means of regular expressions.

[1] This is a minimal extension, as we only extend AspectU for pragmatic reasons; i.e. to illustrate the feasibility of the method presented in this paper.

Below, we present one EAA from the motivating example of Sect. 2. Specifically, we model the `strategyRevision` event using the Java-like constructor syntax introduced in Sect. 3.1. To refer to use case steps or collections thereof, we provide an AspectU+ pointcut expression as the parameter to the `Event` constructor (placed between curly brackets). This pointcut refers to the use case whose name matches to the '`.*Revise.*strategy.*`' regular expression, and more specifically, the steps in the main flow of this use case (`main(``.*'')`). In the CMS use cases, this matches to steps 1–5 of the *"Revise current strategy"* use case (Use Case 1).

```
strategyRevision := Event(                                    1
  { &&(usecase(".*Revise.*strategy.*"),                       2
  main(".*")) });                                             3
```

3.3 Authoring Quality Attribute Scenarios with EAAs

During the quality attribute scenario elicitation and authoring process, the requirements engineer has to be aware of the centrally defined EAA model. This model will aid him in making explicit key assumptions about the system. Whenever such an assumption has to be made (for example, *initial strategy calculation and strategy revision will be done by one and the same component*), he first has to verify whether these assumed system elements —for example, the `strategyRevision` event— have already been defined in the EAA model. If so, he can simply refer to these elements. If not, he first has to introduce them in the EAA model, possibly by reusing or building upon already-existing model elements or pointcuts.

QAS 2 Performance. Strategy calculation

- [..] **Response:** 'Overload' mode: requests are forwarded to a next instance of the sub-system:
  - when processing new information (`processingNewData`), the CMS assesses whether this new information gives rise to a strategy revision (`strategyRevision`). In overload mode, the request is sent to the replica instance.
  - when entering a new dossier (`newDossierEntered`), the initial strategy is selected by the Coordinator (`strategyCalculation`). In overload mode, the request is sent to the replica instance.

QAS 2 presents the Response part of QAS 1 from Sect. 2 after re-factoring. We have introduced annotations (presented in a `typewriter` font and between brackets) that refer to the EAAs. For example, we introduced a direct reference to the `strategyRevision` event defined in Sect. 3.2. In our experience, EAAs are most common in the "*Stimulus*", "*Response*", and "*Response Measure*" fields of a quality attribute scenario.

Clearly, the main goal of our approach is not to re-factor existing quality attribute scenarios after the fact, but to support the authoring process itself. This is in line with the view that the EAA model will act as a central knowledge

repository during requirements engineering, in which and from which the relevant interrelations between the different requirement artifacts can be documented and derived.

4 Evaluation and Discussion

We have evaluated the presented approach in the context of the Crisis Management System (CMS) case study. In total, the case study requirements comprise (i) 13 detailed use cases, and (ii) in total, eighteen quality attribute scenarios, specifically 6 availability, 6 performance and 6 modifiability scenarios. While the use cases originate from the original case study [13], the quality attribute scenarios have been derived from textual software quality descriptions in [13]. Further details on the followed process to obtain these requirements can be found in [28]. Throughout Sect. 3, we have illustrated the method over a very small subset of this case study. Note that the requirements presented in this paper are in fact highly simplified versions of those from the case study.

First, Sect. 4.1 discusses our prototype implementation. Then, Sect. 4.2 evaluates our results in terms of the modularity of early architectural assumptions (EAAs). Finally, Sect. 4.3 discusses how the existence of an explicit EAA model improves the navigability of the requirements body.

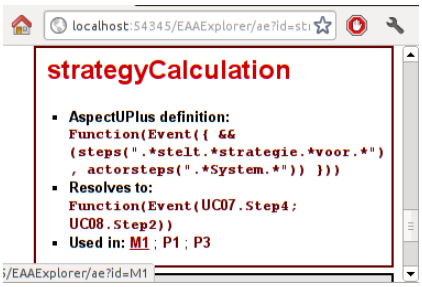
4.1 Prototype Implementation

We have developed a prototype implementing the proposed techniques[2] for modularizing EAAs in scenario-based requirements. The tool can be used for querying the EAA model and navigating the interdependencies between the different requirement artifacts. To this end, it provides a web front-end and the interrelations between EAAs and quality attribute scenarios (which we represented earlier as annotations) are shown by the prototype as hyperlinks. The prototype evaluates the pointcut expressions on demand. It offers a fully implemented pointcut parser and evaluator for the AspectU+ pointcut language. An EAA is presented by showing its AspectU+ pointcut expression, the concrete use case steps which are the concrete join points for that pointcut (as hyperlinks) and the quality attribute scenarios referring to the EAA (also as hyperlinks). The screenshot in Fig. 2(a) illustrates this for the running example of the `strategyCalculation` EAA. When displaying a specific use case, the tool automatically adds hyperlinks referring to the EAAs for which that use case offers join points.

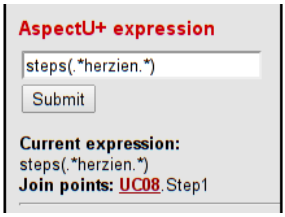
In addition, the tool offers an environment to create and test AspectU+ pointcut expressions, of which Fig. 2(b) presents a screenshot. During the construction of an AspectU+ expression, the expression is evaluated over the use case model and the matching use case-level join points are presented.

In future work, we plan to refine and integrate this tool into requirements engineering tools such as the UCEd [26] which offer a more rigorous approach

[2] The source code of this prototype and further implementation details can be found on `http://people.cs.kuleuven.be/~dimitri.vanlanduyt/eaa/`



(a) Screenshot of browsing the AspectU+ definition of the `strategyCalculation` EAA



(b) Screenshot of creating and testing AspectU+ expressions

to use case and domain modeling, and architecture creation tools such as the SEI's ArchE [25] to impose EAAs as actual architectural constraints during architectural design.

4.2 Modularity of EAAs

In this part of the evaluation, we focus on the modularity of early architectural assumptions (EAAs). Specifically, we have applied scattering metrics [7] (i.e. based of the number of recurring EAA definitions) on the requirements of the CMS, and we compare the cases with and without an explicit EAA model.

To ensure comparability of both sets of quality attribute scenarios, we started with the 18 existing quality attribute scenarios of the CMS, and re-factored these incrementally by moving the EAAs one by one to the EAA model, reusing already-existing definitions wherever possible. Fig. 2 depicts this process. The X-axis shows the order in which quality attribute scenarios are selected and re-factored, while the Y-axis shows the number of distinct EAA definitions. The grey curve represents the case without an EAA model —thus scattering and

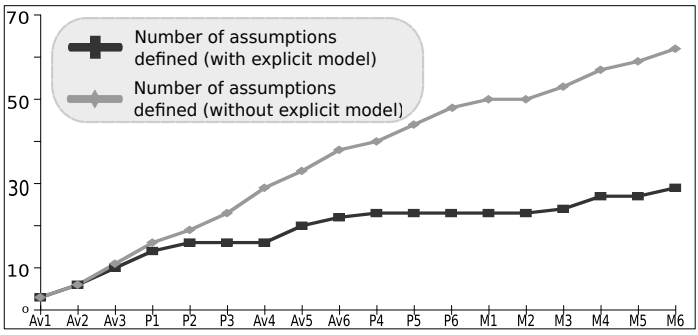


Fig. 2. The number of EAAs defined over time, with and without creating an EAA model

tangling EAAs within and across quality attribute scenarios— and the black curve represents the case with an explicitly defined EAA model, or in other words, the black curve plots the size of the EAA model.

The main observation from this graph is that the number of EAAs grows linearly with the number of quality attribute scenarios. However, when specifying the EAAs in a central EAA model, the number of EAAs definitions grows sub-linearly. Initially, when covering only one quality attribute scenario (*Av1*), both approaches introduce the same amount of EAAs. However, as more quality attribute scenarios are covered, there is clearly a higher reuse of already-defined EAAs. This is an indication that duplication of the EAA definitions (scattering and tangling) is avoided and this is a consequence of the fact that many of the system-level assumptions suitable for one quality attribute scenario also proved suitable for others. In total, this reduction of duplication has led to the definition of 29 EAAs in the centrally defined model, as opposed to 62 EAA definitions scattered across and hard-coded within the 18 quality attribute scenarios, a reduction of 53%.

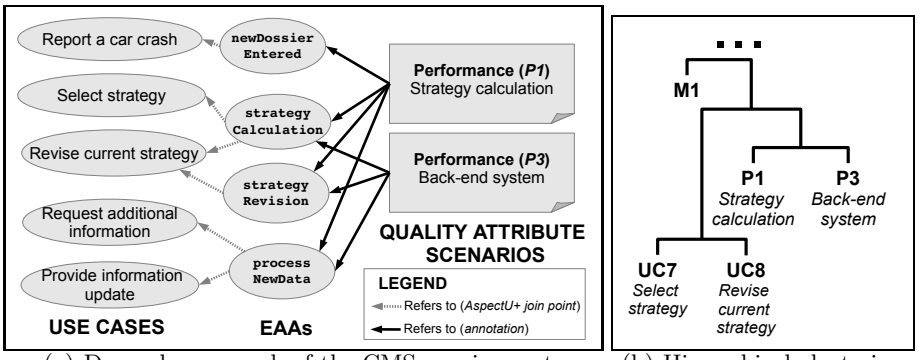
4.3 Requirement Navigability and the Transition to Architecture

We now illustrate how an explicit EAA model facilitates the navigation[3] of requirement bodies in the transition to architecture. Fig. 3(a) presents a dependency graph that has been derived directly from the EAA definitions. In addition to the requirements already presented in Sect. 2, it depicts three additional use cases and a second quality attribute scenario (*P3*) (about to the performance of the back-end system in the CMS). It shows the (otherwise implicit) relations between these requirements, made explicit on the one hand by means of explicit annotations, and on the other hand by means of AspectU+ pointcuts.

From this dependency graph, we have defined a distance function between two requirements that measures how closely related they are in terms of shared EAA references. These distance measurements can then be used to cluster the set of requirements. An example hierarchical clustering of the CMS requirements is presented in Fig. 3(b). This clustering is based on the most straightforward distance function counting the number of shared interdependencies between the requirements (the arrows in Fig. 3(a)). For example, *P1* and *P3* are clustered together because they share multiple EAA definitions.

Such a clustering can then serve as an important input for architecture creation and analysis approaches that rely on grouping requirements and architectural trade-offs, such as the Architecture Trade-off Analysis Method (ATAM) [12]) and the Attribute-Driven Design (ADD) [32,2] process. For example, when selecting *P1* as a key driver for the architecture, the architect can automatically be informed about its close relation to *P3*, and both drivers could be addressed together (for example, in a single ADD iteration).

[3] And as a direct consequence, the consistency management and traceability of requirements.



(a) Dependency graph of the CMS requirements (b) Hierarchical clustering

Fig. 3. Illustration of how explicit EAAs improve the navigability of the CMS requirements

5 Related Work

First, we discuss related work in the domain of Early Aspects (Aspect-Oriented Requirement and Architecture approaches). Then, we zoom in specifically on existing methods and techniques that involve documenting EAAs within use cases. Finally, we discuss related work in the domain of architectural knowledge management.

Early Aspects. Chitchyan et al. have presented their semantics-based composition approach of [4,31] which enables annotating requirements with semantic information obtained with natural language processing techniques. COMPASS further elaborates upon this approach by mapping the semantic compositions to initial AO architecture diagrams [3]. As such, this work traces key concepts in requirements specification towards architecture, facilitating consistency management between architecture and requirements.

The generic AORE model presented by Rashid et al. [20] supports the separation of crosscutting requirements and the identification of crosscutting influences between requirement-level aspects. As such, this enables the identification of critical trade-offs well before architectural design. In addition to this work, Moreira et al. [17] advocate a multi-dimensional approach to separation of concerns in requirements engineering that supports architectural trade-off analysis based on the notion of the *compositional intersection* between the stakeholder concerns.

Many additional AORE approaches focus on identifying and representing the crosscutting influences of non-functional requirements on system functionalities [16,23,27].

Although these approaches are related in terms of the techniques used, the key difference is that these do not explicitly disambiguate between requirements and assumptions. In future work, we will further explore the instantiation of our techniques in the context of these AORE approaches.

Documenting EAAs in Use Cases. Many use case templates have been proposed in literature, some of which explicitly propose fields for referring to related non-functional requirements [15,6]. This is generally considered bad practice among use case modeling experts [5] as it leads to scattered and confusing cross-references between heterogeneous requirements.

In Jacobson's use case-driven approach to AOSD [10], non-functional requirements are addressed by introducing *infrastructure* use cases which are functional realizations of the non-functional requirement (e.g. *log-in* use cases for authentication). As a consequence, the higher-level representation of this requirement is lost and the requirements engineer is forced to commit to a certain realization of the non-functional requirement.

Another common technique involves defining project glossaries of key concepts (which might correspond to EAAs). The main disadvantage of these techniques is a consequence of their disconnect from the other requirement artifacts.

Architectural Assumptions. In their seminal work about architectural mismatch [8], Garlan et al. have demonstrated that implicit architectural assumptions are one major factor impeding effective reuse and evolution of code- and architecture-level artifacts. This is related to *architectural decay* [21]. Roeller et al. [22] propose an approach for retro-actively discovering such architectural assumptions in existing software systems and documenting them. Zschaler et al. [33] have extensively studied *aspect assumptions*. These are all instances of architectural assumptions. However, the main difference to the assumptions targeted in this paper is that we focus on *early* architectural assumptions; i.e. those assumptions made about the system during requirement elicitation and in the transition phases to architecture.
Architectural knowledge management [1,14] focuses on making explicit key architectural assumptions. Again, these approaches focus on *late architectural assumptions* (solution-space assumptions). Given the potentially large impact of EAAs on the architecture and its creation processes, it is nonetheless important to also investigate architectural knowledge management in the context of EAAs. To our knowledge, no other approaches exist with this explicit focus.

6 Conclusion

In the development activities at the transition from requirements to architecture, the requirements engineer often makes *early architectural assumptions* (EAAs); i.e. initial assumptions about key properties or characteristics of the envisioned architectural solution. The inherently implicit nature of EAAs and the lack of modularization thereof has been shown to hinder key architectural design activities [29].

To address this, we have presented a set of techniques to modularize EAAs in the context of scenario-based requirements. These EAAs are stored in a central knowledge repository —the EAA model— which we consider to be a missing link in the transition from requirements to architecture. The crosscutting nature

of EAAs is tackled by employing aspect-oriented requirements engineering techniques and we define EAAs in terms of requirement-level pointcut expressions.

This work addresses some of the research challenges in the domain of architectural knowledge management. As a general trend, the focus shifts from purely documenting the architectural solutions (end products) to documenting the architectural creation processes themselves (i.e., the intermediate results, design decisions, design rationale, etc), and this well before actual architectural design decisions may have been taken.

Acknowledgements. This research is partially funded by the Research Fund KU Leuven.

References

1. Ali Babar, M., Dingsyr, T., Lago, P., Van Vliet, H.: Software Architecture Knowledge Management: Theory and Practice. Springer (2009)
2. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley (2003)
3. Chitchyan, R., Pinto, M., Rashid, A., Fuentes, L.: Compass: Composition-centric mapping of aspectual requirements to architecture. In: Rashid, A., Akşit, M. (eds.) Transactions on AOSD IV. LNCS, vol. 4640, pp. 3–53. Springer, Heidelberg (2007)
4. Chitchyan, R., Rashid, A., Rayson, P., Waters, R.: Semantics-based composition for aspect-oriented requirements engineering. In: Barry, B.M., de Moor, O. (eds.) AOSD. ACM ICPS, vol. 208, pp. 36–48. ACM (2007)
5. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley Professional (January 2000)
6. Coleman, D.: A use case template: Draft for discussion (1998)
7. Eaddy, M., Aho, A., Murphy, G.C.: Identifying, assigning, and quantifying crosscutting concerns. In: Proceedings of the First International ACoM Workshop, ACoM 2007, p. 2 (2007)
8. Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch, or, why it's hard to build systems out of existing parts. In: Proceedings of the 17th ICSE Conference, pp. 179–185 (April 1995)
9. Jacobson, I., Griss, M., Jonssson, P.: Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley (1997)
10. Jacobson, I., Ng, P.-W.: Aspect-Oriented Software Development with Use Cases, 1st edn. Addison-Wesley (December 2004)
11. Katz, S., Mezini, M., Kienzle, J. (eds.): Transactions on Aspect-Oriented Software Development VII. LNCS, vol. 6210. Springer, Heidelberg (2010)
12. Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., Carriere, J.: The architecture tradeoff analysis method. In: Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 1998, pp. 68–78 (1998)
13. Kienzle, J., Guelfi, N., Mustafiz, S.: Crisis Management Systems: A Case Study for Aspect-Oriented Modeling. Transactions on Aspect-Oriented Software Development 7, 1–22 (2010)
14. Kruchten, P., Lago, P., van Vliet, H.: Building up and reasoning about architectural knowledge. In: Hofmeister, C., Crnković, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214, pp. 43–58. Springer, Heidelberg (2006)

15. Malan, R., Bredemeyer, D.: Functional requirements and use cases: System has properties (2005)
16. Moreira, A., Araújo, J.A., Brito, I.: Crosscutting quality attributes for requirements engineering. In: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, SEKE 2002, pp. 167–174. ACM, New York (2002)
17. Moreira, A., Rashid, A., Araújo, J.: Multi-dimensional separation of concerns in requirements engineering. In: RE, pp. 285–296. IEEE Computer Society (2005)
18. Nuseibeh, B.: Weaving together requirements and architectures. IEEE Computer 34(3), 115–117 (2001)
19. Nuseibeh, B.: Crosscutting requirements. In: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, AOSD 2004, pp. 3–4. ACM, New York (2004)
20. Rashid, A., Moreira, A., Araújo, J.: Modularisation and composition of aspectual requirements. In: AOSD 2003: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, pp. 11–20. ACM, New York (2003)
21. Riaz, M., Sulayman, M., Naqvi, H.: Architectural decay during continuous software evolution and impact of 'design for change' on software architecture. In: Ślęzak, D., Kim, T.-h., Kiumi, A., Jiang, T., Verner, J., Abrahão, S. (eds.) ASEA 2009. CCIS, vol. 59, pp. 119–126. Springer, Heidelberg (2009)
22. Roeller, R., Lago, P., van Vliet, H.: Recovering architectural assumptions. Journal of Systems and Software 79(4), 552–573 (2006)
23. Rosenhainer, L.: Identifying crosscutting concerns in requirements specifications (2004)
24. Sillito, J., Dutchyn, C., Eisenberg, A.D., De Volder, K.: Use case level pointcuts. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 246–268. Springer, Heidelberg (2004)
25. U. Software Engineering Institute (SEI) (Carnegie Mellon). Arche, architecture expert design assistant, `http://www.sei.cmu.edu/architecture/tools/arche/`
26. Some, S.S.: Use cases based requirements validation with scenarios. In: IEEE International Conference on Requirements Engineering, pp. 465–466 (2005)
27. Tekinerdogan, B., Moreira, A., Araujo, J., Clements, P.: Presented papers: finding aspects in requirements with theme/doc (2004)
28. Van Landuyt, D., Truyen, E., Joosen, W.: Discovery of stable abstractions for aspect-oriented composition in the car crash management domain. In: Katz, S., Mezini, M., Kienzle, J. (eds.) Transactions on AOSD VII. LNCS, vol. 6210, pp. 375–422. Springer, Heidelberg (2010)
29. Van Landuyt, D., Truyen, E., Joosen, W.: Documenting early architectural assumptions in scenario-based requirements. In: Proceedings of the Joint 10th Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture (2012)
30. Van Landuyt, D., Truyen, E., Joosen, W.: On the modularity impact of architectural assumptions. In: Proceedings of the 2012 NEMARA Workshop, NEMARA 2012, pp. 13–16 (2012)
31. Weston, N., Chitchyan, R., Rashid, A.: Formal semantic conflict detection in aspect-oriented requirements. Requir. Eng. 14, 247–268 (2009)
32. Wojcik, R., Bachmann, F., Bass, L., Clements, P.C., Merson, P., Nord, R., Wood, W.G.: Attribute-driven design (add), version 2.0. Technical report, Software Engineering Institute (November 2006)
33. Zschaler, S., Rashid, A.: Aspect assumptions: a retrospective study of aspectj developers' assumptions about aspect usage. In: Proceedings of the Tenth International Conference on AOSD 2011, pp. 93–104. ACM (2011)