SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2013; **43**:305–332
Published online 26 February 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2113

Mastering crosscutting architectural decisions with aspects

Claudio Sant'Anna[1,*,†], Alessandro Garcia[2], Thais Batista[3] and Awais Rashid[4]

[1]*Federal University of Bahia, Computer Science Department, Brazil*
[2]*Pontifical Catholic University of Rio Janeiro, Informatics Department, Brazil*
[3]*Federal University of Rio Grande do Norte, Computer Science Department, Brazil*
[4]*Lancaster University, Computing Department – InfoLab21, UK*

SUMMARY

When reflecting upon driving system requirements such as security and availability, software architects often face decisions that have a broadly scoped impact on the software architecture. These decisions are the core of the architecting process because they typically have implications intertwined in a multitude of architectural elements and across multiple views. Without a modular representation and management of those crucial choices, architects cannot properly communicate, assess and reason about their crosscutting effects. The result is a number of architectural breakdowns, such as misinformed architectural evaluation, time-consuming trade-off analysis and unmanageable traceability. This paper presents an architectural documentation approach in which aspects are exploited as a natural way to capture widely-scoped design decisions in a modular fashion. The approach consists of a simple high-level notation to describe crosscutting decisions, and a supplementary language that allows architects to formally define how such architectural decisions affect the final architectural decomposition according to different views. On the basis of two case studies, we have systematically assessed to what extent our approach: (i) supports the description of heterogeneous forms of crosscutting architecture decisions, (ii) improves the support for architecture modularity analysis, and (iii) enhances upstream and downstream traceability of crosscutting architectural decisions. Copyright © 2012 John Wiley & Sons, Ltd.

Received 19 February 2011; Revised 6 January 2012; Accepted 10 January 2012

KEY WORDS: architectural decisions; modularity; aspect-oriented software development; early aspects

1. INTRODUCTION

Explicit representations of software architectures and associated design decisions are fundamental to tame the growing complexity of software systems [1, 2]. Architects strive to develop evolvable and reusable architectural decisions especially for systems in volatile business domains such as banking, telecommunications and mobile applications. To be reusable and evolvable, the specification of architecturally relevant design choices must be modular. This serves a twofold purpose. If the specification of architectural decisions is modular one can reason in isolation about their rationale and implications to the final design decomposition. This is termed modular reasoning [3] about architectural decisions. At the same time, the various choices need to relate to each other in a systematic and coherent fashion to realize the intended architecture. Effective representation and specification of such relationships makes it possible to reason about the multiple architectural decisions as a whole — using the modular reasoning outcomes as a basis. We refer to this global reasoning as compositional reasoning [4] about architecturally-relevant choices.

*Correspondence to: Claudio Sant'Anna, Computer Science Department, Federal University of Bahia, Brazil.
†E-mail: santanna@dcc.ufba.br

In fact, existing software architecture description approaches are already geared towards supporting such modular and compositional reasoning. For instance, the '4 + 1' view model [5] separates an architecture into logical, process, physical and development views, derived from the various stakeholders' perspectives. This makes it possible for an architect to modularly reason about each of the views. A fifth view, the scenario or use case view, shows how elements in the other views work together thus supporting compositional reasoning. In addition, architectural styles and patterns [6] are based on the recognition of the effectiveness of specific organizational principles and structures. This helps one to undertake compositional reasoning about the elements deployed using a particular architectural pattern or style. However, it is well known that only describing final architectural decompositions are not sufficient to support time-effective evolution and well-informed reuse of software architecture [2]. In fact, several authors (e.g. [2, 7]) motivate the need for explicit documentation to support conveying of change, implications, rationale, options and facilitating traceability.

Architectural decisions and their rationale encompass critical structural and behavioural implications for the various architectural elements and the architecture we wish to reason about. Making such decisions explicit is critical to enable designers to trace which requirements and emerging architectural concerns influenced the definition of the coarse-grained modularity units and their relationships. Even though it is important to document architectural decisions in a systematic fashion, it is not a trivial task to modularly describe them with existing notations [2, 7, 8]. The experience of others [8, 9] and our own [10–12] show that it is particularly challenging to capture architecturally-relevant choices exerting a broadly scoped impact on the architecture. The problem is that the decisions are typically associated with day-to-day software design concerns, such as security, error handling, availability, and performance. Unfortunately, both modular and compositional reasoning are not fully supported by existing techniques to describe design decisions with such a global design impact [2, 7, 8].

Take, for instance, the '4+1' view of a software architecture that addresses several broadly scoped properties, such as availability. When attempting to understand the availability-specific architectural decisions and their implications, an architect needs to reason across the various views, that is the logical, process, physical and development views. This is because those availability decisions are likely to relate to multiple elements across those views. This is particularly challenging as availability-specific architectural decisions often lead to the addition of new elements within a view. Because these implications associated with a single architectural concern (availability, in this case) are scattered and tangled across various architecture elements (Figure 1(a)) and views themselves, it is difficult to undertake modular reasoning about particular decisions. Compositional reasoning is even more challenging as one needs to understand the combined implications of various architectural decisions spanning a multitude of elements across logical, process, physical and development views. This implies that, in addition to systematic documentation of a decision, it is important to capture the additional structure it introduces into the various elements in the views.

In this context, this paper presents an architecture documentation approach to providing such modular and compositional reasoning support, which is based on the use of aspect-oriented composition mechanisms [13, 14]. Figure 1 provides a pictorial representation of our proposal to separately record crosscutting choices. Our documentation technique exploits the notion of aspects (Figure 1(b)) to modularize and compose crosscutting architecture decisions that would otherwise be interspersed with other relevant decisions' descriptions (Figure 1(a)). The colored dots in Figure 1(b) represent well-defined points in the architecture representation (or *join points*) related to crosscutting decisions associated with specific concerns — for example, availability, security and performance — now modularized into single *aspects*. Our technique consists of a simple template to define crosscutting decisions as aspects. The template specifications modularly describe the broadly influencing concerns that otherwise would be scattered and tangled over the architecture description and its multiple views. The template is general and agnostic to different architectural representations.

Such aspectual templates are supplemented with a language that allows architects to formally describe how such architectural decisions affect the final architectural decomposition according to different views. This formal description supports architects to undertake modular reasoning

DOI: 10.1002/spe

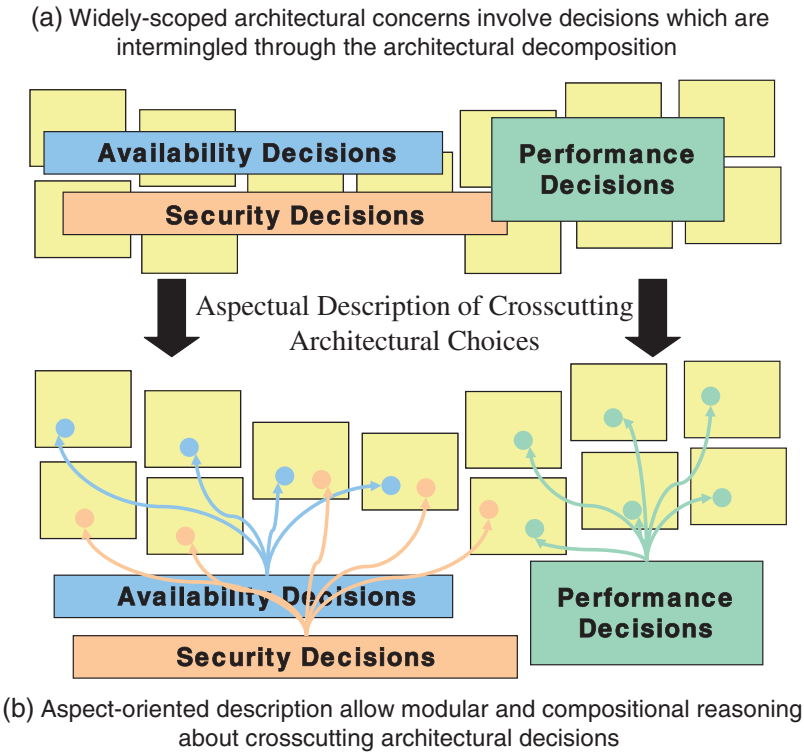


Figure 1. Supporting improved modularity and composition of crosscutting architectural decisions (a) with an aspect-oriented documentation approach (b).

about decisions' implications, such as the evaluation of its effects on the system modularity both through early design stages and system maintenance steps. More importantly, by systematically exposing the semantics of a decision's compositional relationship with architectural elements, we can support an architect to undertake compositional reasoning about the combined implications of various architectural decisions. In this paper, we also perform a multidimensional assessment of the proposed aspect-oriented architecture documentation technique based on two case studies from different application domains. We evaluate to what extent the use of aspect-oriented composition mechanisms improves or not the ability of software architects on the: (i) explicit modularization of widely-scoped architecture decisions, (ii) better-informed analysis of architecture alternatives in terms of relevant modularity attributes, and (iii) upstream and downstream traceability of crosscutting architecture concerns.

This paper is organized as follows. Section 2 discusses the importance of exploring the notion of aspects to document broadly scoped architectural decisions and uses a running case study to illustrate the crosscutting nature of decisions associated with recurring architectural concerns. Section 3 describes a general-purpose template that we propose to explicitly communicate crosscutting architectural decisions. Section 3.2 focuses on the composition of such architectural decisions. Section 4 presents the evaluation of the proposed template in the light of a second case study. Section 5 discusses the benefits and drawbacks of aspectizing architectural decisions. It also discusses related work. Section 6 presents future trends in this context, and Section 7 presents our final remarks.

2. CROSSCUTTING ARCHITECTURAL DECISIONS

Software is no longer engineered using a rigid separation of development stages. With the increasing adoption of iterative and agile methodologies, gone are the times when a strict separation between

DOI: 10.1002/spe

requirements engineering, architecture, design, implementation and evolution was perceived as good practice. Key architectural decisions may be taken as early as requirements engineering. Section 2.1 discusses the influence of broadly scoped concerns derived from requirements specifications on early design decisions. Examples of such crosscutting architectural decisions are illustrated through a case study. Section 2.2 discusses typical design impairments caused by the lack of modular and compositional reasoning for such crosscutting decisions.

2.1. From requirements to architecture: the running case study

Broadly scoped concerns, whether functional or nonfunctional, for example availability, security, performance, informational retrieval, etc., identified during requirements engineering have important architectural implications. Aspect-oriented requirements engineering techniques [15] make it possible to systematically identify, modularize, represent, and compose such broadly scoped concerns [16]. Such techniques, therefore, make it possible to modularly reason about such concerns and undertake compositional analysis for early identification of trade-offs among them. These broadly scoped concerns and their interferences provide early insights into the various architectural decisions facing an architect.

Although the explicit handling and representation of architectural decisions are of paramount importance, they are not trivial tasks. Many decisions associated with relevant architectural concerns are crosscutting by their very nature and, as a result, they need to be treated as such. They cut through the primary modularities of the architecture description, which is often consisted of one or more views. An architectural concern can affect several elements in an architecture description, such as components and their interfaces, relationships, processes, and also the decisions associated with other concerns.

To illustrate these problems, we rely on examples taken from the architecture of a context-sensitive tour guide system. This application was developed at Lancaster University to support visitors in configuring their own tours around the historical and cultural attractions in the city [17]. Using a handheld device, the visitors can retrieve information about the various attractions and obtain route guidance from their current location to the site of their choice. The system also supports generation of custom tours based on the preferences of a particular user and includes access to hotel, restaurant and theatre reservation services. The system has already undergone a systematic analysis of various crosscutting features [18] based on which the software architecture was subsequently derived. In the following, we discuss the crosscutting nature of architectural decisions in such context-sensitive tour guide system.

Figure 2 shows a partial description of the software architecture for this application based on a component-and-connector view [9] and on additional views from the '4 + 1' view model [5]. The upper left depicts a structural diagram with the component-and-connector view. The visitors use a Navigator to navigate through a tour, to create a customized tour, and to update information about the navigation preferences. The Navigator component contacts the Information Retrieval component to recover information from the system. The Navigator also contacts the ExternalServices component to connect the visitor to external services. The LocationManager provides the identification of the current location of a visitor. This identification is used by the Information-Retrieval component that provides tourist information according to his/her current location. The TouristInfoManager allows the tourist centre to update information in the system.

In this structural perspective of the tour guide architecture, it is also clear that the decisions with respect to the availability requirement affect several points of the architecture specification. Although availability-specific choices are somewhat localized in the Replication Manager component, they largely impact on the definition of several interfaces and components, which do not have the primary purpose of addressing availability issues. Availability-related decisions crosscut multiple components, including InformationRetrieval, LocationManager, and TouristInfo-Manager. Because availability support requires the replication of critical components and the replica consistency management, specific components and interfaces need to be created and added to those affected components. The crosscutting phenomenon also involves other concerns, such as security and performance.

DOI: 10.1002/spe

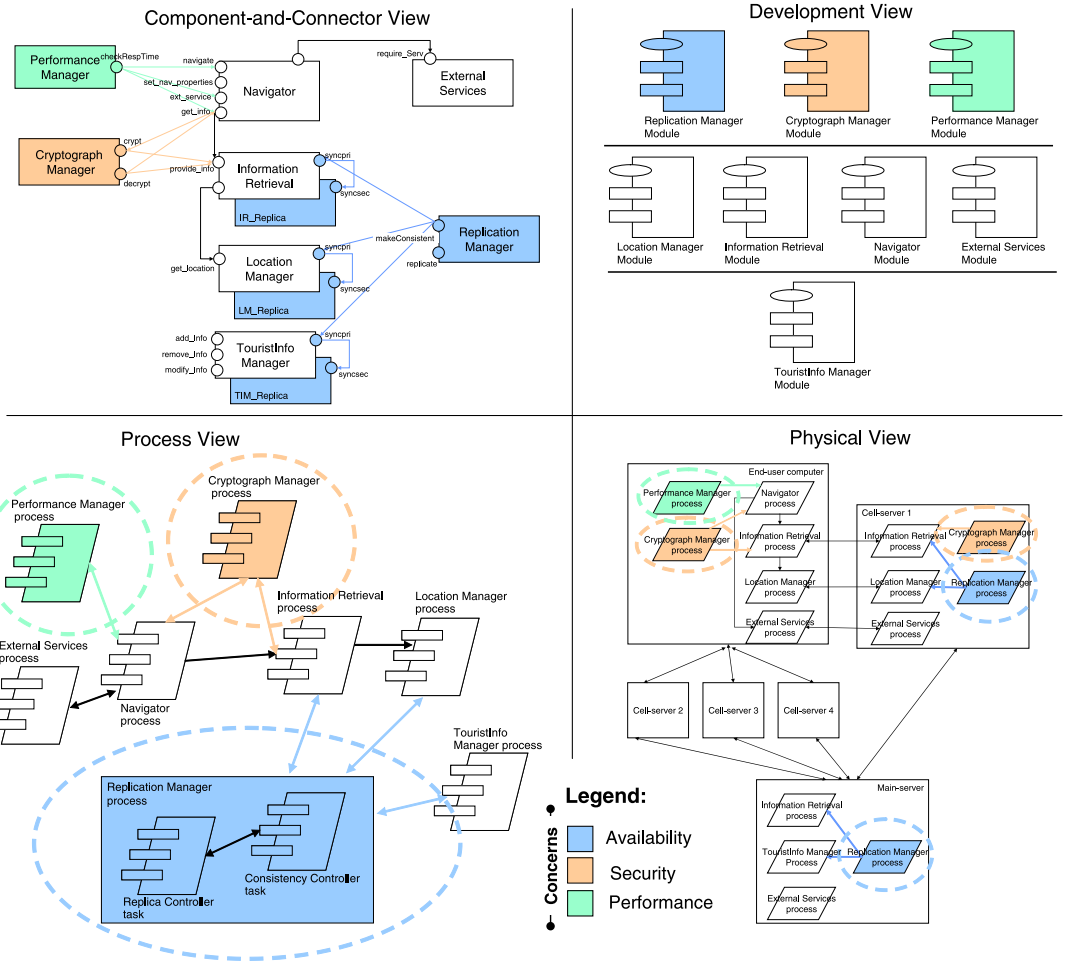


Figure 2. Tangling and scattering in an architecture description.

The crosscutting manifestation leads to two major problems at the architectural level, the so-called scattering and tangling. Architectural scattering is the manifestation of architectural decisions, which belong to one specific concern, in several architectural units encapsulating architectural decisions referred to other architectural concerns. For example, the replication-related interfaces are scattered over multiple architectural components, such as LocationManager, InformationRetrieval and TouristInfoManager components (upper left of Figure 2). Architectural tangling is the mix of multiple concerns together in the same architectural elements. For instance, tangling is evident in the InformationRetrieval component because it is realizing an availability-related interface in addition to its primary functionality of providing information.

2.2. *Early design impairments*

As previously mentioned, there are some architectural concerns that bring deeper problems to the software architects; they can even crosscut other architectural views in addition to the structural view, as it is the case for the availability concern. The availability-specific decisions are scattered and tangled within elements of other concerns over the four architectural views. Availability requires not only the inclusion of components, interfaces, and connectors (component-and-connector view), but also the definition of two separate threads to manage both replication and consistency (process view), the conception of the management layer together with other supplementary managers (development view), and the distribution of replication elements through different servers (physical view).

Traditional architectural approaches such as 4 + 1 view model [5], the architecture tradeoff analysis method (ATAM) [19], tactics [20, 21], and architectural styles or patterns [6] have different

DOI: 10.1002/spe

complementary purposes. However, they are not aimed at supporting the separate handling of crosscutting architectural decisions as exemplified in Figure 2. It brings in turn a number of substantial pitfalls, such as the following.

2.2.1. Hindering of modular and compositional reasoning. Tangling and scattering of decisions hinder both modular and compositional reasoning at the architectural stage. The architects are unable to reason about an architectural concern while looking only at its description, including its core decisions and structural and behavioural implications. Hence, its analysis inevitably forces architects to consider all the architectural artefacts in an *ad hoc* manner. For example, the architects treating the availability and security concerns in Figure 2 need to consult the definitions and decisions associated with all other architectural concerns across all the different views, leading to an expanded or global reasoning rather than a modular reasoning.

2.2.2. Losing essential information. Many of the concerns in the requirements specification entail crosscutting architectural decisions. The mapping of those concerns to the respective decisions is awkward because the developers do not have proper ways to easily check whether and how the requirements are met in the software architecture. With traditional approaches, software architects are not able to locally express the structural and behavioural, physical, and developmental implications of a given architectural decision in several architectural elements and views. The result is that important information is irrecoverable just because of the lack of support for properly specifying them. Not only the final choices can be lost, but also the crosscutting rationale and competing options the architects considered. Traceability also becomes unmanageable. For example, the association of availability-specific requirements with their architectural implications (Figure 2) is cumbersome and far from being trivial. This obstacle makes it difficult to assess the goodness of the software architecture even in the presence of a well-informed requirements engineering process.

2.2.3. Inaccuracy on architecture modularity analysis. Current architecture analysis methods are not able to quantify the interplay of key architecture concerns and architecture modularity properties [11, 12]. For instance, existing architecture metrics fail to inform that security-related decisions have a wide impact on several component interfaces and architectural coupling [22]. The underlying problem is that such measurement approaches cannot rely on artefacts that identify the architectural elements related to each concern, thereby causing a number of false negatives and false positives in architecture assessment processes [12].

2.2.4. Decreasing evolvability. Architecture degeneration is becoming very common in an age where software systems are always changing. Architecture artefacts are often key deliverables in the evolution process. As a consequence, the architects have additional work to answer recurring questions: What happens if we decide to change security-related components of our system? Has this decision been affected by which architectural concerns? Because a complex architecture probably reflects thousands of crosscutting decisions, finding the answers for these questions is naturally time consuming, especially when the original architects are no longer available.

2.2.5. Reducing reuse possibilities. Tangling and scattering are two of the main anti-reuse factors in the software lifecycle. The lack of a clear separation of concerns generates undesirable burdens on architectural reuse. For example, software architects may want to recycle, or at least remember, a comprehensive list of decisions and the rationale associated with an architectural concern in posterior projects. It would be certainly beneficial to empower software architects to reuse successful crosscutting architectural choices from previous projects.

3. CAPTURING ARCHITECTURAL DECISIONS AS ASPECTS

In light of the aforementioned problems, we conjecture that crosscutting architectural decisions should be handled as separate architectural aspects. The idea is to have proper abstractions to enable

DOI: 10.1002/spe

their representation as first-class elements, and also provide the means to facilitate their further composition. Aspects were originally conceived to address crosscutting concerns at the programming level [14]. It is then natural to believe that the key for capturing crosscutting architectural decisions is exploiting some aspect-oriented concepts [13] at the architectural level.

3.1. Aspectual templates

Architectural aspects are units of modularity to capture the decisions associated with broadly scoped concerns, letting the architects represent all the implications in a single place. We represent architectural aspects as templates, called *aspectual templates*. Figures 3 and 4 show aspectual templates with essential information to capture crosscutting decisions: (i) name of the architectural aspect; (ii) architectural decisions, such as the inclusion of components, interfaces, relationships, processes, and so forth, which were made with the sole purpose of contemplating issues related to the architectural aspect; (iii) composition rules to describe how the crosscutting decisions with respect to this architectural aspect affects other architectural elements and alternatively other aspects; and (iv) a reasoning section that captures the rationale behind those decisions. The architectural elements placed in the second compartment of an aspectual template should be grouped by the architectural views used in the original architectural description, for instance, component-and-connector view, process view, physical and so forth (see Figures 3 and 4).

The crosscutting decisions affect several architecture elements, which are named architectural joint points. An architectural joint point is an element of interest in the architecture description through which two or more architectural decisions may be composed. Examples of join points are: a component, an interface, a process, an architectural aspect, or even an architectural decision. Architectural composition rules support the composition specification and enable compositional

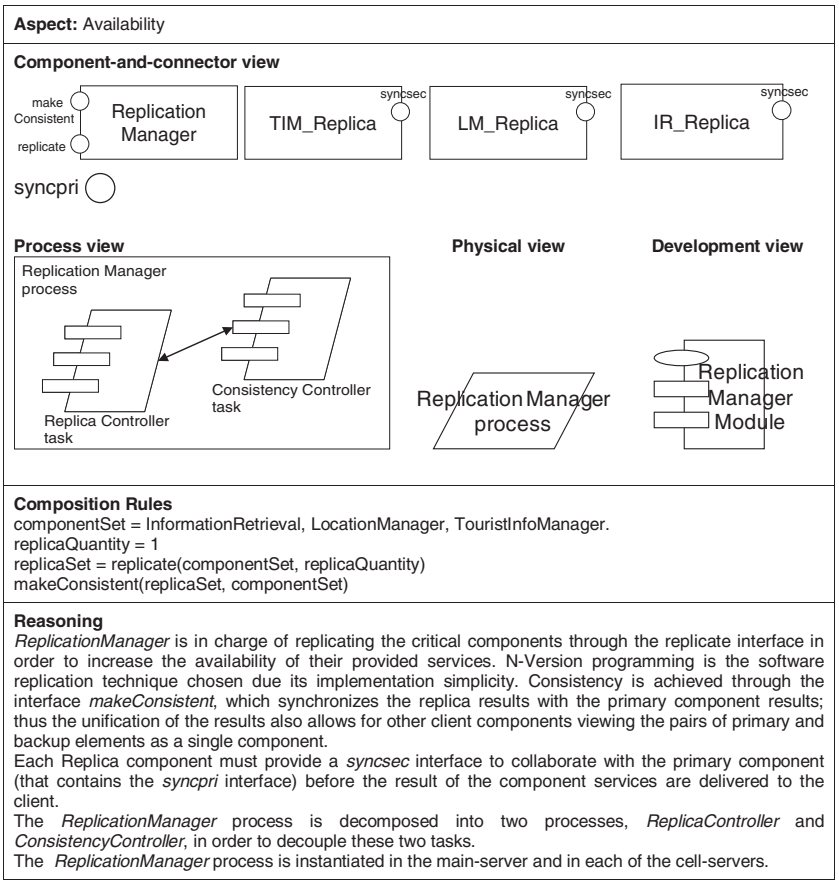


Figure 3. Modularizing and composing architectural aspects: availability.

DOI: 10.1002/spe

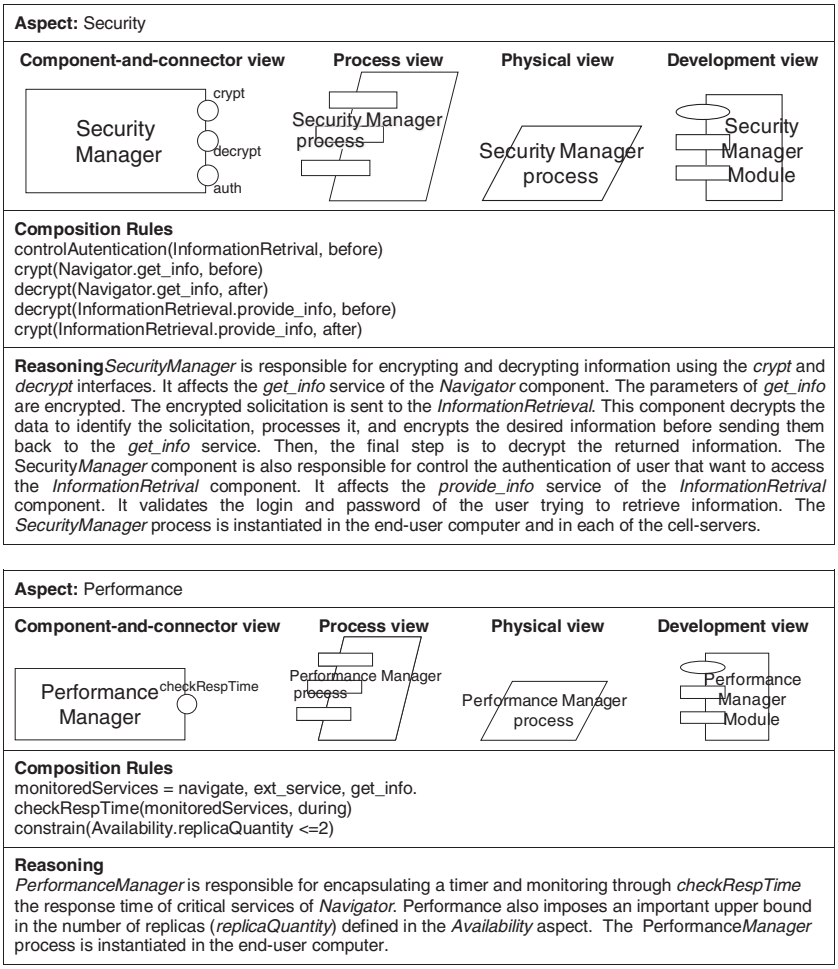


Figure 4. Architectural aspects: security and performance.

reasoning. They are a means of referring to collections of architectural join points and describing some architectural decisions to be applied at those join points.

Figures 3 and 4 show how to use the notion of architectural aspects to support the modular description of the availability, security and performance concerns in our running example. All the availability-specific decisions are clearly captured in the first template, including the creation of a ReplicationManager and system replicas, and the definition of two processes for controlling the system replicas and their global consistency. The rationale behind the availability decisions are reported in the reasoning section of the template. The reasons are related to structural and behavioural decisions and the composition decisions. In a similar way, the security-related and performance-related decisions are respectively isolated in the 1st and 2nd templates of Figure 4.

As a result, the template-based specification is a cohesive manner to describe those broadly influencing concerns that otherwise would be scattered and tangled over the architecture description and its multiple views. Notice that this approach is general and agnostic to different architectural representations that the software developers are relying on, whether graphical or textual, such as ADLs (architectural description languages), UML-based or XML-based notations. Our assumption here is that architectural representations define graphical or textual elements to represent software entities, such as components and processes, and the composition between them. Aspectual templates do not impose any restriction on what type of elements can be used to fill them. The software architect can, therefore, work with any architectural representation. The software architect can also use the templates in conjunction with any set of architectural views, and any existing notations for reflective design, where design rationale is extensively recorded [7]. In fact, the template can be used

DOI: 10.1002/spe

to describe all kinds of architectural decisions and rationale, including assumptions, constraints, positions, arguments, status, and the like.

3.2. Composing architectural decisions

Properly documenting the composition of architecture decisions is critical because architects make them in complex environments. The architects can use a high-level composition language to facilitate the registration and communication of broadly scoped choices and enhance compositional reasoning. Figures 3 and 4 show how to work with a high-level language to describe those choices as architectural composition rules. The naming of the composition rules is intuitive as it actually captures the architectural operation associated with the crosscutting decisions. The composition rules are domain dependent to provide a friendly description of the decisions. They are formalised by their translation to domain-agnostic low-level rules, called as mapping rules (Sections 3.2.1 and 3.2.2).

For example, the third composition rule in the first template (Figure 3), named replicate, captures the fact that a list of architectural components should be replicated because of availability purposes. Auxiliary declarations can be made to facilitate the quantification process, such as the use of *componentSet* and *replicaSet*. The first rule uses *componentSet* to quantify the architectural join points affected by the replicate decision. Those points are critical components to be duplicated with different implementation versions, namely InformationRetrieval, LocationManager, and TouristInfo Manager. The rule *makeConsistent* abstracts the process of including architectural elements to address the consistency of the primary components and their replicas.

To facilitate the composition of architectural decisions, the rules can pick out different types of architectural join points, such as interfaces or even rules defined in other architectural aspects. Figure 4 shows the crypt and decrypt decisions in the security aspect affect interfaces of Navigator and InformationRetrieval. Some behavioural information can also be part of the composition rules. For instance, the specification of the security aspect also includes 'when' the crypt and decrypt decisions should actuate over specific architectural elements, that is 'before' and 'after' requests of services of Navigator and InformationRetrieval.

The third rule of the performance aspect, named constraint, influences an availability rule that specifies the number of replicas. This rule represents a recurring scenario faced by software architects: several aspectual decisions affect each other. The aspectual templates promote composition interfaces that allow for the architect to make explicit the relationships and mutual influences of broadly scoped concerns, which are not easily captured in traditional architectural views. In fact, this architectural constraint involving performance and availability components was not explicitly represented by any of the views in Figure 2. Note that this feature allows that aspectual templates make explicit references to other aspectual templates. This follows an asymmetric aspect-oriented approach, which is similar to some aspect-oriented programming models, such as AspectJ, whose aspects are also allowed to make references to other aspects. This is an important feature because concerns influence other concerns. However, as already well understood by the aspect-oriented software development (AOSD) community, the architects should use this feature carefully, because it introduces coupling between aspectual templates. In fact, aspectual templates supporting tools must control the dependence between templates. The tools, for instance, should inform if an aspectual template is referenced by other aspectual templates. We could have opted for a symmetric approach where aspect interactions would be registered in a separated part of the architecture description. However, symmetric approaches have disadvantages as well. For instance, maintaining the specification of the concerns separated from their composition specification tends to be hard. It is often the case that the composition specification cannot be reasoned about if the architect does not read the affected concern specifications.

3.2.1. Mapping rules. As previously mentioned, architectural aspects can influence decisions made in several views. The architect may want now to review together the crosscutting decisions and the architectural views with the rest of the project team and the project stakeholders. Hence, once the architectural aspects have been defined, the actual effect of the decisions in the multiple views may

DOI: 10.1002/spe

need to be specified and analyzed. The next alternative step then would be to use underlying mechanisms to support the mapping of aspectual decisions in terms of elements of the other architectural views. Those mechanisms can rely on a language with mapping rules that simply translate the aspectual decisions in terms of the corresponding elements in the architectural views. Figure 5 shows how those mapping rules could be applied for mapping availability, security and performance decisions to elements of a component-and-connector view. A similar mapping process could be carried out for the other architectural views.

The mapping rules consist of a small set of reusable primitives that can be easily translated to elements of different architectural views and also a quantification mechanism over architectural elements. The Backus-Naur Form (BNF) description of the mapping rules is presented in Figure 6. The process of designing the set of mapping rules and producing its grammar follow two different and complementary threads: (i) we analysed different architectural description languages and some well-known traditional configuration languages such as those by Magee *et al.* [42, 43] and some of their constructs (create, add) to compose a configuration inspired us to define equivalent mapping rules; (ii) we conducted an analysis of some case studies, to identify their needs in terms of composition rules, and we defined new rules. On the basis of our findings from the two threads, we composed the final set of rules. Finally, we applied our rules in the two case studies presented in this paper, which have different needs in terms of architectural operations associated with crosscutting decisions. Our rules were enough to meet all the compositional needs of the different elements involved in these case studies.

The BNF assumes that no whitespace is necessary for proper interpretation of the rule. The item <element-name> is to be substituted with an architecture element's name declared in any view of the architecture description. The item <role-name> is to be substituted with a role's name specified by an architectural style. The item <value> is to be substituted with a number. The entries *architectural_elem* and *plural_architectural_elem* should be defined according the abstractions encompassed by the used architecture description approach. In the case of the component-and-connector view of the tour guide system, these two entries are defined as follows:

- *architectural_elem* :: = 'component' | 'interface' | 'constraint'
- *plural_architectural_elem* :: = 'components' | 'interfaces' | 'constraints'

The description of each mapping rule and the graphical representation of their effects are depicted as follows. Table I shows a summary of the foundational set of mapping rules.

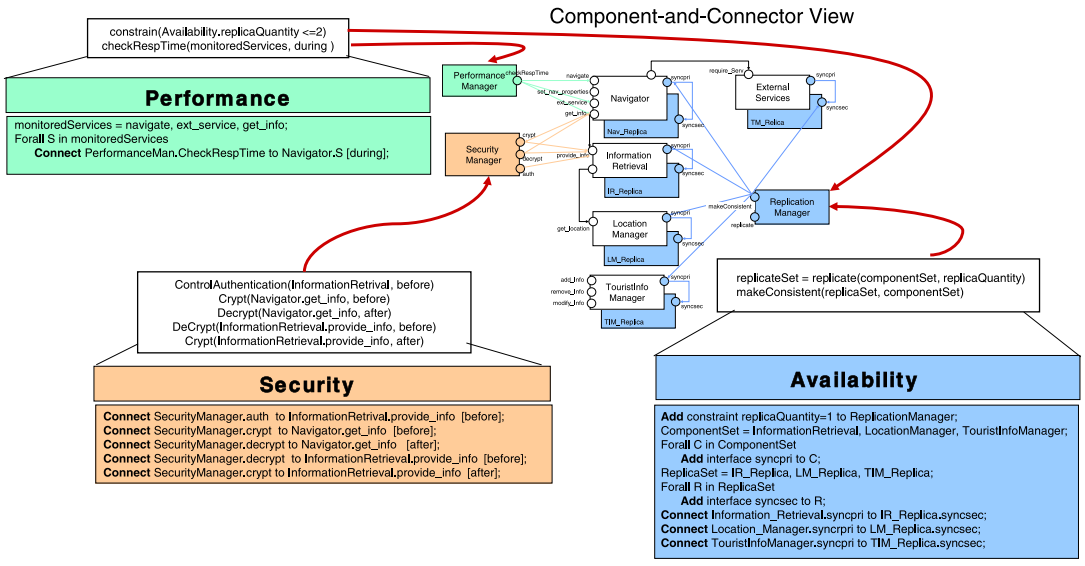


Figure 5. Mapping architectural aspects to architectural views: effects of architectural aspects in the component-and-connector view.

DOI: 10.1002/spe

rules ::= {*rule*}

rule ::= primitive | *forall_statement* | *assignment_statement*

primitive ::= create_ primitive | *add_ primitive* | *modify_ primitive* | *remove_ primitive* | *split_ primitive* | *unify_ primitive* | *connect_ primitive* | *disconnect_ primitive* | *play_ primitive*

add_primitive ::= "Add" *architectural_elem* <elem-name>["=" <value>] "to" <elem-name> ";"

modify_ primitive ::= "Modify" *architectural_elem* <elem-name> "=" <value> | <elem-name> <elem-type> "to" <elem-type> ";"

remove_ primitive :: = "Remove" *architectural_elem* <elem-name> "from" <elem-name> ";"

split_ primitive ::= "Split" *architectural_elem* <elem-name> "into" <elem-name> "," <elem-name> {"," <elem-name>} ";"

unify_primitive ::= "Unify" *plural_architectural_elem* <elem-name> "," <elem-name> {"," <elem-name>} "into" <elem-name> ";"

connect_ primitive ::= "Connect" <elem-name> "to" <elem-name> ";"

disconnect_ primitive ::= "Disconnect" <elem-name> "from" <elem-name> ";"

play_primitive ::= "Play" <elem-name> ", role" <role-name> ";"

forall_statement ::= "Forall" *variable* "in" *architecture_element_set rule_list* "end"

assignment_statement ::= architecture_element_set "=" (*all_statement* | <elem-name>) {"," (*all_statement* | <elem-name>)} ";"

all_statement ::= "All" *plural_architectural_elem* "in" (variable | <element-name>)

variable ::= **A..Z** {**A..Z** | **0..9**}

architecture_element_set ::= **a..z** {**a..z** | **A..Z** | **0..9**}

Figure 6. BNF description of the mapping rules.

Table I. Summary of the mapping rules.

MAPPING RULE	DESCRIPTION
Add *<architectural_elem> <elem_name1 [=value]>* ***to*** *<elem_name2>*	introduces an architectural element with name <elem_name1>, optionally set its value, to other architectural element with name <elem_name2>
Modify *[<architectural_elem> <elem_name1 = value] | [<elem_type1* ***to*** *<elem_type2>]*	changes an architectural element by setting a new value to <elem_name1> or modifying its type from <elem_type1> to <elem_type2>
Remove *<architectural_elem> <elem_name1>* ***from*** *< elem_name2>*	removes an architectural element with name <elem_name1> from other architectural element <elem_name2>
Split *<architectural_elem > <elem_name>* ***into*** *<elem_name_list>*	separates an architectural element with name <elem_name> into two or more elements defined in <elem_name_list>
Unify *<architectural_elem> <elem_name_list>* ***into*** *<elem_name>*	groups two or more architectural elements defined in <elem_name_list> in the architectural element <elem_name>
Connect *<elem_name1>* ***to*** *<elem_name2>*	defines a relationship between the elements <elem_name1> and <elem_name2>
Disconnect *<elem_name1>* ***from*** *<elem_name2>*	removes a relationship between the elements <elem_name1> and <elem_name2>
Play *<elem_name>,* ***role*** *<role_name>*	adds the responsibility denoted by the role <role_name> to the architectural element <elem_name>

DOI: 10.1002/spe

Add inserts an architectural element into another element. The value of the inserted architectural element is an optional argument. For instance, in the second occurrence of the **Add** rule for the Availability concern in Figure 5, the syncpri interface is inserted into component C. Composite elements, such as composite components, can be defined using this operator. For instance, Add component TIM_Europe to TouristInfo Manager transforms TouristInfoManager in a composite component that contains TIM_Europe, as illustrated in Figure 7.

Connect describes which elements are associated with each other. For example, it supports the description of how components' interfaces are bound. Specifically it describes the interconnection between operations of two different components. For instance, in Figure 5 syncpri interface of the Information_Retrieval component are connected to syncsec interface of the IR_Replica component. ADLs usually provide similar interconnection operations such as bind or connect.

Disconnect removes an association between two elements. For example, Disconnect InformationRetrieval.syncpri from IR_Replica.syncsec, removes the connection between the syncpri interface of InformationRetrieval and syncsec of IR_Replica.

Play assigns a new role to an architectural element. The role is specified by a previously defined architectural style that contains architectural element types and properties. The assignment of a role to an element implies that such an element receives all the syntactic and semantic properties of the original style. For instance, an exception_handling style may define two elements types: Exception_Handler and Exception_Propagator. Both define properties and interfaces. Exception_handler contains the handler interface and the Exception_Propagator contains the propag interface. In Figure 18 this style is instantiated using a sequence of Play rules. For instance, to the Distribution_Manager component is assigned the role of an Exception_Propagator and the role of an Exception_Handler is assigned to GUI_Elements. Figure 8 shows the effect of this rule to the Distribution_Manager component. Now it plays the role of Exception_Propagator and contains properties, exceptions and interfaces originally described to Exception_Propagator. The dotted rectangle is used to represent that the internal element is playing the role of the external one.

Split segregates the functionalities of an element in two or more elements. It disconnects all ports of the original element. Each new element inherits the properties of the original architectural element if no explicit parameter is specified. Otherwise, each element inherits only the specified properties. For instance, a security architectural concern represented by a SecurityManager component with some cryptograph and authentication operations can be split into two smaller components — CryptographManager and AuthenticationManager — each one with a specific responsibility. The interfaces of each one are specified as a parameter of the element. Figure 9 illustrates the Split component SecurityManager into CryptographManager(crypt, decrypt) and Authentication Manager(auth).

Figure 7. Add mapping rule example.

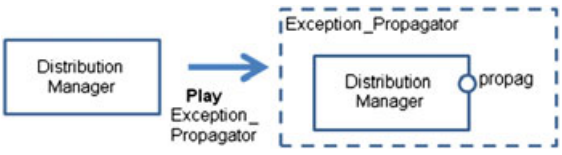


Figure 8. Play mapping rule example.

DOI: 10.1002/spe

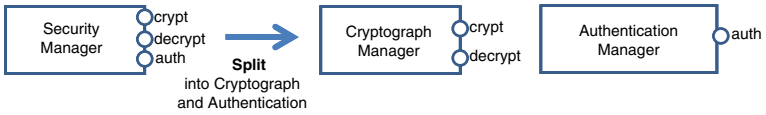


Figure 9. Split mapping rule example.

Unify allows the definition of a new element from other elements. The elements that are unified do not exist anymore. For instance, Unify component CryptographManager, Authentication-Manager into SecurityManager creates a component SecurityManager that combines the features of CryptographManager and AuthenticationManager, as illustrated in Figure 10.

Modify changes an architectural element by setting a new value to it or changes the type of an element. For instance, Modify constraint replicaQuantity = 1 sets a new value (1) to the replicaQuantity constraint. To change the type of an element from Tourist InfoManager to TouristInfoManager Europe the following rule is used: Modify Tourist InfoManager to TouristInfoManagerEurope. Figure 11 represents the effect of this rule:

Remove removes an architectural element from other architectural element. For example, Figure 12 represents the rule remove component TIM_Europe from TouristInfo Manager.

3.2.2. Evolving architectural decisions. In an evolutionary environment, it is challenging to document architectural changes through the architecture views in a way architects can easily understand [7]. The architect wants to clearly identify the architectural decisions that represent key changes associated to a concern without having to wade through the architecture views just to find a few key items that have changed. The mapping rules previously described can be used to document architectural decisions associated to the evolution of an architecture. With the use of these mapping rules, the architect can document changes made to architectural elements of the architecture because of changes in the concern captured by the template. Figure 13 illustrates the use of the **Split** rule to

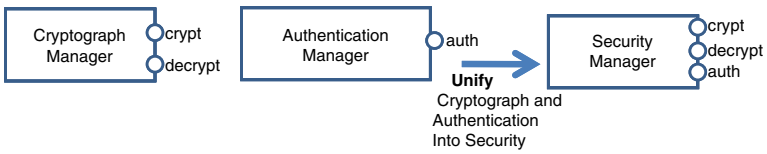


Figure 10. Unify mapping rule example.

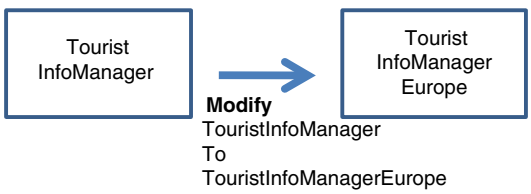


Figure 11. Modify mapping rule example.

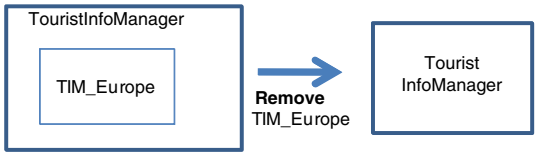


Figure 12. Remove mapping rule example.

DOI: 10.1002/spe

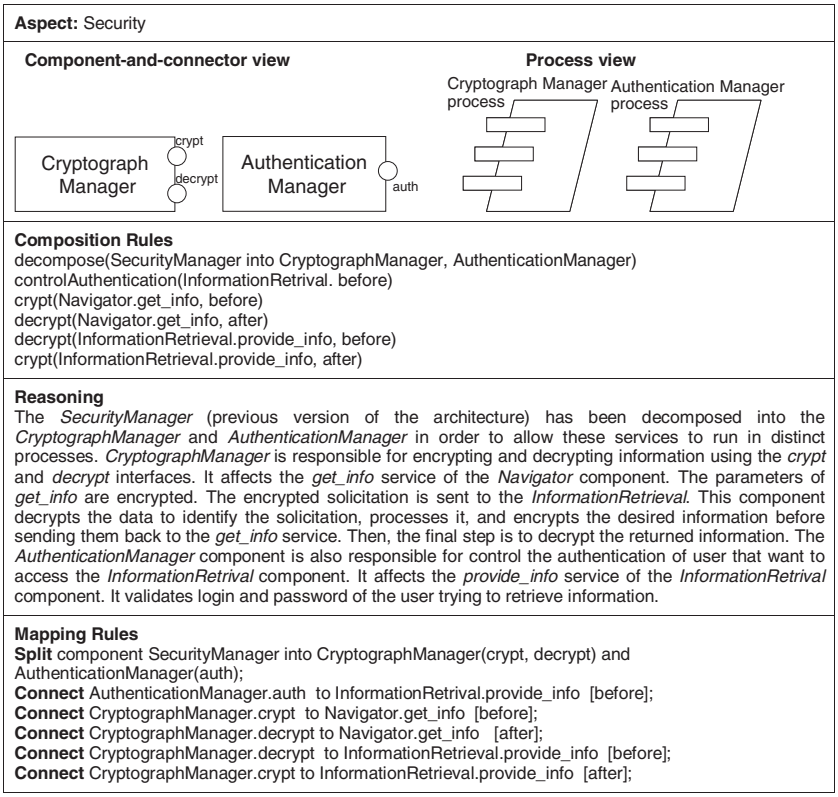


Figure 13. Security aspectual template: evolving the architecture.

document an evolution scenario that involves the security concern of the Tourist Guide architecture. The other mapping rules work similarly to the **Split** rule to document evolution scenarios.

In the example of Figure 13, the architects decided to run the cryptograph and authentication services in two distinct processes. Therefore, in the new version of the architecture, the SecurityManager component is decomposed in two components: CryptographManager and AuthenticationManager. The new Security aspectual template (Figure 13) documents the architectural decisions related to this change and its implications. The template shows the two new components, CryptographManager and AuthenticationManager, and no longer shows the SecurityManager component. An additional composition rule is included to indicate that the SecurityManager component was decomposed into two other components. The reasoning section of the template was updated to describe now the responsibilities of the CryptographManager and AuthenticationManager components and explicitly register the reason for decomposing the SecurityManager component into these two new components. Finally, the mapping rules section of the template includes now the **Split** rule:

Split component SecurityManager into CryptographManager(crypt, decrypt) and AuthenticationManager(auth).

4. CASE STUDY

To carry out an evaluation of the notion of architectural aspects, we undertook a second case study where we have used the proposed templates in the context of a system with an architecture blueprint largely different from the context-sensitive tour guide system (Section 2.1). We assessed the applicability of the aspectual templates and mapping rules while documenting the architectural decisions of a system called Health Watcher (HW) [23]. It supports the registration and management of complaints to the health public system, and several stakeholders are involved, including citizens,

DOI: 10.1002/spe

administrators, health agents, and so forth. On the basis of this case study, Section 4.1 discusses the usefulness of our approach for supporting concern-centric quantitative assessment of software architecture.

We have selected the Health Watcher system for several reasons. First, it is a real Web-based information system deployed in 2000 by the Public Health System in Recife, a city located in the north of Brazil [23]. After its initial deployment, a number of versions have been released in the last years. Second, this system has served as a kind of benchmark for the assessment of contemporary modularization techniques, such as AOSD [11, 12, 16, 23–26]. Third, modularity-driven requirements, such as reusability and maintainability, have been settled as one of the key priorities in the solution design. Figure 14 illustrates a graphical representation of the component-and-connector view of the HW architecture description based on UML 2.0 notation. The HW architecture follows the combination of the client–server style with a layered style [27]. Six main architectural concerns were considered in the HW system and they are described in Table II.

Figure 14 also shows how the architectural concerns are scattered and tangled over the architectural elements of the Health Watcher system. The gray boxes placed over or near a component or interface indicate that that element is related to the concerns the boxes represent. For instance, the box with the letter P on the superior left corner of the Transaction_Control component means that this component is part of the persistence concern. Similarly, the 'P' box near the UseTransactionControl required interface (in the Business_Rules component) indicates that this interface is related to the persistence component. The architects decided that the UseTransactionControl interface is related to the persistence concern because its only role is to require

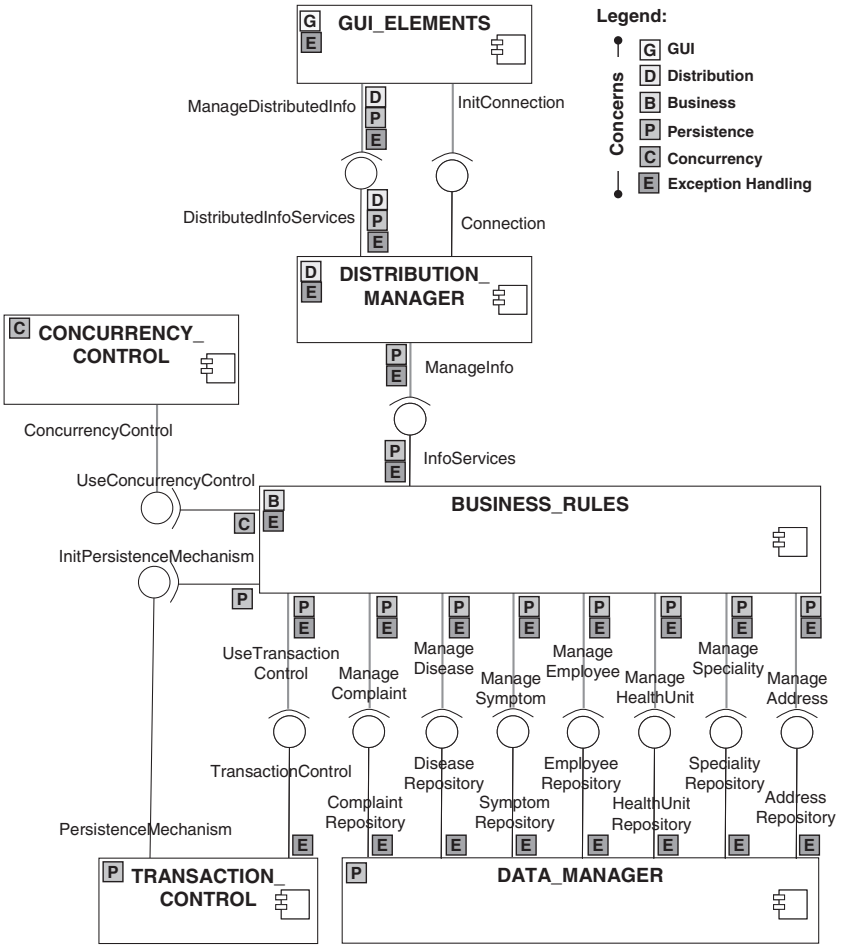


Figure 14. Health watcher architecture.

DOI: 10.1002/spe

Table II. Health watcher architectural concerns.

Concern	Description
GUI	The GUI_Elements component provides a Web interface for the system
Distribution	The Distribution_Manager component externalizes the system services at the server side and support their distribution to the clients
Business	The Business_Rules component defines the business elements and rules
Persistence	The Data_Manager and Transaction_Control components address the persistency concern by storing the information manipulated by the system and providing transaction control.
Concurrency	The Concurrency_Control component provides control for avoiding inconsistency in the information manipulated by the system
Exception Handling	Exceptional events raised and handled by the components support forward error recovery.

the transaction control service, which is a persistence related service. A box near an interface also indicates that there is at least one operation in that interface that is related to that concern or raises or receive an exception related to that concern. For instance, there are three boxes near the DistributedInfoServices interface (in the Business_Rules component) because it contains at least: (i) one operation that raises exceptions ('E' box), (ii) one operation that raises persistence-specific exceptions ('P' box), and (iii) one operation that raises distribution-specific exceptions ('D' box). Similarly the ManageDistributedInfo is also related to error handling, persistence and distribution concerns, but instead of raising exceptions, it receives exceptions raised by the DistributedInfoServices interface.

As shown in Figure 14, the architectural decisions related to some of the concerns of interest in the HW system affect several points in the architecture. As stated before, this phenomenon hinders the modular reasoning about these architectural concerns. Figure 14 only shows the component-and-connector architectural view; however, the same problem also occurs in others views, such as the module view and physical view. In this way, we use the notion of architectural aspects to support the modular description of the concerns in HW architecture. Figures 15 and 16 present the aspectual templates for the persistence and exception handling concerns, respectively. All the persistence-specific decisions are captured in the template shown in Figure 15, including: (i) the creation of the Data_Manager component and its connection with the Business_Rules component; (ii) the creation of the Transaction_Control component; (iii) the creation of the InitPersistenceMechanism and UseTransactionControl interface in the Business_Rules component and their connection with the Transaction_Control component; and (iv) the creation of two persistence-specific exceptions (Transaction Exception and RepositoryException) and their assignment to the operations that raise or receive them. The rationale behind the persistence decisions are reported in the reasoning section of the template.

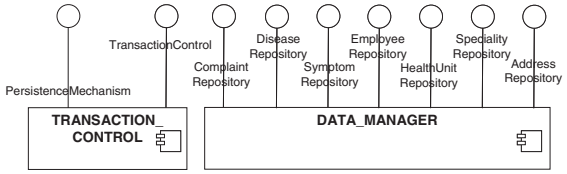
Figure 17 shows the mapping of the persistence architectural aspect (Figure 15) to the component-and-connector view by means of the mapping rules (Table I). The persist (Business_Maganer) high-level composition rule (Figure 15) means that the information manipulated by the Business_Rules component should be persisted. It is translated into a number of **Connect** mapping rules (lines 04–10), which represent the connection between the provided interfaces of the Data_Manager component to the required interfaces of the Business_Rules component. The controlTransaction (Business_Rules) rule captures the fact that the Business_Rules component should control the transaction while persisting the information it manipulates. This high-level rule is translated into two pairs of **Add** and **Connect** mapping rules. The first one (lines 13–14) represents the creation of the initPersistenceMechanism interface in the Business_Rules component and the connection of this interface to the PersistenceMechanism interface of the

DOI: 10.1002/spe

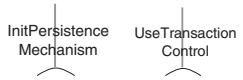
Aspect: Persistence

Component-and-connector view

Components



Interfaces



Exceptions

TransactionException
RepositoryException

Composition Rules

```
persist(Business_Rules);
controlTransaction(Business_Rules);

interfaceSet = All provided interfaces in Data_Manager, InfoServices, DistributedInfoServices;
raiseException(interfaceSet, RepositoryException);

interfaceSet = ManageComplaint, ManageDisease, ManageSymptom, ManageEmployee,
               ManageHealthUnit, ManageSpeciality, ManageAddress, ManageInfo,
               ManageDistributedInfo;
receiveException(interfaceSet, RepositoryException);

interfaceSet = TransactionControl, InfoServices, DistributedInfoServices;
raiseException(interfaceSet, TransactionException);

interfaceSet = UseTransactionControl, ManageInfo, ManageDistributedInfo;
receiveException(interfaceSet, TransactionException);
```

Reasoning

Data_Manager and *Transaction_Control* comprise the persistence services of the system. *Data_Manager* provide services, such as insert, update and search, for handling with persistent information manipulated by the system. This component depends on a specific persistence platform. *Transaction_Control* provides services to allow the transaction control for persisting information. These services – begin transaction, commit transaction and rollback transaction – are provided by the *TransactionControl* interface. This component is also in charge of the persistence services initialization via *the PersistenceMechanism* interface.

The *initPersistenceMechanism* is a required interface used to request the initialization of the persistence services. In the Health Watcher architecture it is realized by the *Business_Rules* component. Similarly, the *UseTransactionControl* interface requires transaction control services and is also realized by the *Business_Component*.

TransactionException and *RepositoryException* are persistence-related exceptional events raised by the *Transaction_Control* and *Data_Manager* components, respectively. They are received by the components that call the services of these components.

Figure 15. Aspectual template for the persistence architectural concern.

Transaction_Control component. The second pair of rules (lines 15–16) represents the creation of the UseTransactionControl interface in the Business_Rules component and the connection of this interface to the TransactionControl interface of the Transaction_Control component.

The following composition rules in Figure 15 are regarding the persistence-specific exceptional events raised or received by a number of interfaces. The raiseException(interfaceSet, RepositoryException) high-level rule (Figure 15) specifies which interfaces raise the RepositoryException exception: (i) all the provided interfaces in Data_Manager (ii) Info Services in Business_Rules, and (iii) DistributedInfoServices in Distribution_Manager. The Data_Manager component raises the exception, and the Business_Rules and Distribution_Manager components propagate that exception. This rule is mapped to two loop blocks of mapping rules which add the RepositoryException to every operation in the aforementioned interfaces (lines 19–23). In a similar way, the receiveException(interfaceSet,

DOI: 10.1002/spe

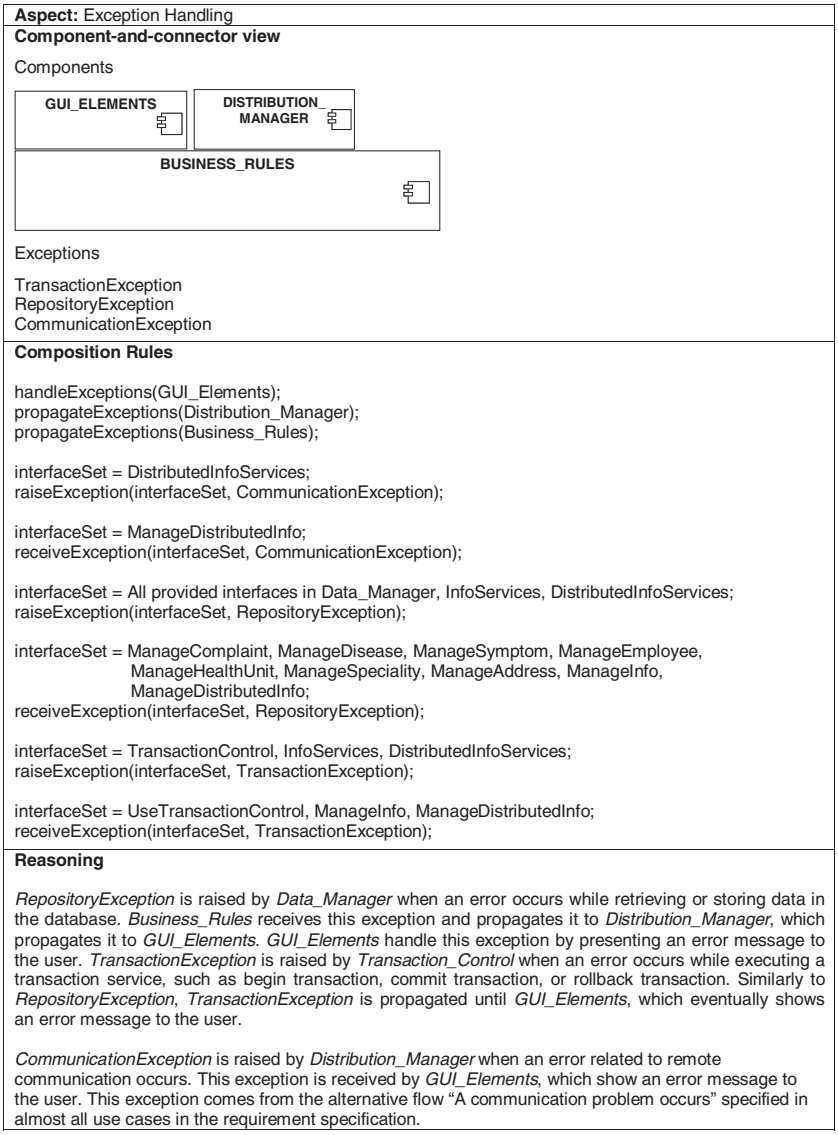


Figure 16. Aspectual template for the exception handling architectural concern.

RepositoryException) high-level rule specifies which interfaces receive the Repository Exception exception. It is translated to mapping rules that add the RepositoryException to (i) specific required interfaces in the Business_Rules component, (ii) ManageInfo in Distribution_Manager, and (iii) ManageDistributedInfo in GUI_Elements (lines 26–30). Likewise, the TransactionException is added to the interfaces that raise or receive it (lines 32–42). Note that adding an exception to a provided interface means that the interface raises the exception. On the other hand, adding an exception to a required interface means that the interface receives the exception from a provided interface connected to it.

All the decisions related to exception handling are captured in the template shown in Figure 16, including: (i) the creation of exceptions, such as TransactionException, Repository Exception, and CommunicationException; (ii) the attachment of the exceptions to the interfaces that raise or receive them; (ii) the fact that GUI_Elements handles exceptions; and (iv) the fact that Distribution_Manager and Business_Rules propagate exceptions. Figure 18 shows the mapping of the exception handling architectural aspect (Figure 16) to the component-and-connector view by means of the mapping rules (Section 3.2.1). The mapping

DOI: 10.1002/spe

```
01 // These mapping rules a related to the …
02
03 //… "persist(Business_Rules)" composition rule
04 Connect Data_Manager.DiseaseRepository to Business_Rules.ManageDisease;
05 Connect Data_Manager.SymptomRepository to Business_Rules.ManageSymptom;
06 Connect Data_Manager.EmployeeRepository to Business_Rules.ManageEmployee;
07 Connect Data_Manager.HealthUnitRepository to Business_Rules.ManageHealthUnit;
08 Connect Data_Manager.SpecialityRepository to Business_Rules.ManageSpeciality;
09 Connect Data_Manager.ComplaintRepository to Business_Rules.ManageComplaint;
10 Connect Data_Manager.AddressRepository to Business_Rules.ManageAddress;
11
12 //… "controlTransaction(Business_Rules)" composition rule
13 Add interface initPersistenceMechanism to Business_Rules;
14 Connect Transaction_Control.PersistenceMechanism to Business_Rules.initPersistenceMechanism;
15 Add interface UseTransactionControl to Business_Rules;
16 Connect Transaction_Control.TransactionControl to Business_Rules.UseTransactionControl;
17
18 //… "raiseException(interfaceSet, RepositoryException)" composition rule
19 interfaceSet = All provided interfaces in Data_Manager, InfoServices, DistributedInfoServices;
20 Forall I in interfaceSet
21    operationSet = All operations in I;
22    Forall O in operationSet
23       Add exception RepositoryException to O;
24
25 //… "receiveException(interfaceSet, RepositoryException)" composition rule
26 interfaceSet = ManageComplaint, ManageDisease, ManageSymptom, ManageEmployee,
                  ManageHealthUnit, ManageSpeciality, ManageAddress, ManageInfo,
                  ManageDistributedInfo;
27 Forall I in interfaceSet
28    operationSet = All operations in I;
29    Forall O in opeartionSet
30       Add exception RepositoryException to O;
31
32 //... "raiseException(TransactionControl, TransactionException)" composition rule
33   interfaceSet = TransactionControl, InfoServices, DistributedInfoServices;
34   Forall I in interfaceSet
35      operationSet = All operations in I;
36      Forall O in operationSet
37         Add exception TransactionException to O;
38
39 //… "receiveException(UseTransactionControl, TransactionException)" composition rule
40   interfaceSet = UseTransactionControl, ManageInfo, ManageDistributedInfo;
41   Forall I in interfaceSet
40      operationSet = All operations in I;
41      Forall O in operationSet
42         Add exception TransactionException to O;
```

Figure 17. Mapping the persistence architectural aspect to the component-and-connector view.

rules related to RepositoryException and TransactionException are omitted because they are identical to the ones shown for the persistence architectural aspect (Figure 17). The handle Exceptions(GUI_Elements) high-level composition rule (Figure 16) means that the GUI_Elements component handles the exceptions it receives. It is translated into the **Play** mapping rule (line 04), which indicates that GUI_Elements plays the role of exception handler. The propagateExceptions(Distribution_Manager) and propagate-Exceptions(Business_Rules) mean that Distribution_Manager and Business_Rules, respectively, propagate the exceptions they receive. Each of them is also translated to the **Play** mapping rule (lines 07–10), which specifies that they play the role of exception propagator. The next composition rules in the template (Figure 16) determine which interfaces raise or receive exceptions. As previously explained for the persistence architectural aspect, these composition rules are translated to blocks of the **Add** mapping rule (from line 17 on).

4.1. Concern-driven modularity analysis

Although typical architecture modularity problems are related to the inadequate modularization of concerns, most of the current quantitative assessment approaches do not explicitly consider concern as a measurement abstraction. A number of architecture quantitative assessment methods are targeted at guiding decisions related to modularity, without calibrating the measurement outcomes to the driving architectural concerns. A number of case studies have pointed out that detection of certain concern-related design flaws can be observed in early design stages [23–25].

One of the reasons for this limitation of current architecture measurement approaches was the lack of a systematic support for mapping and documenting of architectural concerns. As previously shown, the aspectual template approach provides a means for documenting the concerns in architecture description, that is registering the architecture elements related to each considered architectural

DOI: 10.1002/spe

```
01 // These mapping rules a related to the …
02
03 "… handleExceptions(GUI_Elements)" composition rule
04 Play GUI_Elements, role Exception Handler
05
06 "… propagateExceptions(Distribution_Manager)" composition rule
07 Play Distribution_Manager, role Exception Propagator
08
09 "… propagateExceptions(Business_Rules)" composition rule
10 Play Business_Rules, role Exception Propagator
11
12 "… raiseException(interfaceSet, CommunicationException)" composition rule
13 operationSet = All operations in DistributedInfoServices;
14 Forall O in operationSet
15    Add exception CommunicationException to O;
16
17 "… receiveException(interfaceSet, CommunicationException)" composition rule
18 operationSet = All operations in ManageDistributedInfo;
19 Forall O in operationSet
20    Add exception CommunicationException to O;
21 …
22 …
```

Figure 18. Mapping the exception handling architectural aspect to the component-and-connector view.

Table III. Suite of concern-driven architectural metrics.

Attribute	Metric	Definition
Concern Diffusion	Concern Diffusion over Architectural Components (CDAC)	It counts the number of architectural components which contributes to the realization of a certain concern.
	Concern Diffusion over Architectural Interfaces (CDAI)	It counts the number of interfaces which contributes to the realization of a certain concern.
	Concern Diffusion over Architectural Operations (CDAO)	It counts the number of operations which contributes to the realization of a certain concern.
Dependence Between Architectural Concerns	Component-level Interlacing Between Concerns (CIBC)	It counts the number of other concerns with which the assessed concerns share at least a component.
	Interface-level Interlacing Between Concerns (IIBC)	It counts the number of other concerns with which the assessed concerns share at least an interface.
	Operation-level Overlapping Between Concerns (OOBC)	It counts the number of other concerns with which the assessed concerns share at least an operation.
Component Cohesion	Lack of Concern-based Cohesion (LCC)	It counts the number of concerns addressed by the assessed component.

concern in the system. Therefore, this approach allows the definition and application of metrics that are based on the concern abstraction. In this context, we defined an initial suite of concern-driven architecture metrics [22] and applied them in the HW architecture [22], to evaluate their usefulness on analysing architecture modularity. We also use the same set of metrics in other studies [38].

Our metrics suite mainly relies on evaluating the modularization of architectural concerns. Therefore, it includes metrics for quantifying separation of concerns and their interactions. For instance, it quantifies the diffusion of a concern realization within architecture specification elements, such as components and interfaces. Our concern-oriented metrics focus on the evaluation of software architecture representations, such as UML-based or ADL specifications, and are computed based on the documentation of aspectual templates. Table III presents a summary of the architecture metrics suite with a brief definition for each of the metrics and their association with distinct modularity attributes they measure. To calculate the metrics values, we rely on the aspectual templates for identifying the architectural elements related to a concern.

We undertook a case study to illustrate how aspectual templates can be used in a measurement framework to help architects quickly summarize and evaluate the merits of architectural alternatives. In this case study we compared the concern modularization in two alternatives of the Health Watcher architecture (herein referred to as first and second alternatives). The first alternative is the architecture obtained from the Java version of Health Watcher. The second alternative is the architecture obtained from the AspectJ version of Health Watcher. The architecture description of both alternatives is based on UML diagrams. The second alternative, in particular, uses a UML

DOI: 10.1002/spe

extension to describe component-and-connector views of aspect-oriented software. It is important to highlight that we applied aspectual templates to the architecture description of both alternatives. Having the aspectual templates, we were able to compute the concern-driven architectural metrics (Table III). Note that the purpose of this section is not discussing the differences between the alternatives in details, but showing that aspectual templates make it possible for architects and architecture reviewers to reason about architectures in ways that have previously been difficult to perform.

Table IV presents the results obtained with the application of the concern diffusion and dependence between concerns metrics (Table III) on both alternatives. The metrics results are shown per concern (first column). The results for the first and second architectural alternatives are shown side-by-side for each metric and concern. The detailed discussion of the results and architecture alternatives is out of the scope of this paper. The results for the concern diffusion metrics (CDAC, CDAI and CDAO) show that the persistence and exception handling concerns are spread over more architecture elements in the first solution. The outcomes also show that the second alternative eliminates the dependence between concerns at the component level. Note, for instance, that, in the first alternative, the business concern is interlaced with two concerns at the component level (CIBC metric). The dependence related to interface-level interlacing is also lower in the second solution. For instance, the persistence concern is interlaced with 4 other concerns at the interface level in the first alternative, against only one concern in the second one (IIBC metric). Finally, the results regarding the metric for operation-level overlapping show that the second alternative for the Health Watcher architecture was not able to improve this kind of dependence.

We developed a tool, called Concern-Oriented Measurement Tool (COMET), which automates the application of the concern-driven metrics aforementioned [39]. This tool partially supports the notion of aspectual template. COMET allows the architect to import the architecture specification of a system from an XML file or define the architecture directly in the tool. Having the architecture imported or defined, the architect can specify the architectural elements related to each architecturally-relevant concern. COMET includes a module that allows the architect to specify and manage the list of concerns in the architecture. It also allows the architect to assign each architectural element to the concern being realized by it. In addition, the architect can view all the architecture elements related to a concern in a single place, as is done by the aspectual template mechanism. For now, COMET only supports the notion of mapping rules, more specifically the **Add** mapping rule: each architectural element assigned to a concern represents an element added to the architecture because of the realization of the concern. We are working on extending COMET to support all the features of an aspectual template, such as composition rules, reasoning documentation and the complete set of mapping rules.

5. DISCUSSION

This section discusses the benefits and drawbacks (Section 5.1) of modularly capturing crosscutting decisions using our approach described in Sections 3 and 4. This discussion is based on our extensive experience in both building aspect-oriented software architectures for different application domains [28–34], and defining and assessing aspect-oriented abstractions to the architectural stage [28, 35, 36].

5.1. Advantages and drawbacks

In the beginning, we identified numerous problems in conventional architecture-centric development approaches. By aspectizing crosscutting architectural decisions, we were able to address those issues and bring additional benefits. First, our documentation approach seems to enhance modular and compositional reasoning of architectural decisions. With aspectual templates architects can reason about the otherwise crosscutting concerns in isolated and combined manners. In fact, the template sections describing the reasoning and the composition rules are more than just simple decisions — they also communicate the compositional rationale, and from where the architectural decisions came from. Suppose, for instance, that an architect wants to work out all the operations that raise or receive persistence-specific exceptions in the Health Watcher architecture. Without

DOI: 10.1002/spe

Table IV. Health watcher: concern diffusion and dependence between concerns measures.

Concerns	CDAC		CDAI		CDAO		CIBC		IIBC		OOBC	
	First altern.	Second altern.	First altern.	Second altern.	First altern.	Second altern.	First altern.	Second altern.	First altern.	Second altern.	First altern.	Second altern.
GUI	1	1	2	2	14	14	0	0	3	0	0	0
Distribution	2	1	5	1	51	16	0	0	3	1	1	1
Business	1	1	8	9	57	57	**2**	**0**	2	0	0	0
Persistence	**5**	**2**	**22**	**9**	**154**	**45**	2	0	**4**	**1**	**1**	**1**
Concurrency	2	1	2	2	4	4	2	0	0	0	0	0
Exception Handling	**5**	**2**	**24**	**8**	**156**	**52**	0	0	3	2	**2**	**2**

DOI: 10.1002/spe

the support of the persistence architectural aspect, global reasoning [3] is required to discover the complete set of interfaces, in the sense that the architect has to examine all the operations in all interfaces of all architectural components. With the support of the persistence architectural aspect, only modular reasoning is necessary, because the architect has only to examine the persistence template (Figure 15). Looking into the composition rules it is straightforward to identify the interfaces whose operations comprises persistence-specific exceptions: the set of interfaces are specified just before each raiseException or receiveException rule.

'Architectural aspectization' also lets you trace decisions back to concerns in requirements (such as, availability, performance, and security). The exception handling aspectual template (Figure 16), for instance, explicitly mentions that the CommunicationException comes from a specific alternative flow entry in the use case description of the Health Watcher system. This is an example on how aspectual templates enhance upstream traceability. With respect to downstream traceability, 'architectural aspectization' also improves the identification of candidates to design and implementation aspects, linking them with their counterparts in the design and implementation artefacts. The composition rules inform the design team that those architectural elements might be potentially modularized as design and implementation aspects. In fact, based on the analysis of the crosscutting impact of the persistence and exception handling concerns, the Health Watcher system implementation was reengineered to modularise the transaction manager control and the exception handling concern with AspectJ [37] aspects [23].

In architectural evolution processes, the aspectual templates let architects by and large know the effects the previous design decisions had in the evolving system. Without such an explicit handling of architectural choices, the evolution process would likely lead to the violation of relevant crosscutting assumptions and influences that were not properly documented just because there was no proper support for their expression. Consider an evolution scenario in which the Health Watcher system has to be changed to support the management of historical information about the people who interact with the system making complaints or asking information about the health services. This category of user is called the citizen in the system requirement specification. In this context, two new interfaces — CitizenRepository and ManageCitizen — have to be included in the Data_Manager and Business_Rules component, respectively (Figure 14). Also, new operations related to the new service have to be included in the InfoServices, ManageInfo, DistributedInfoServices and ManageDistributedInfo interfaces. At this point, the architect must discover what exceptions must be raised or received by the new operations, or more generally, discover the existing *structure* of exception handling. Without the exception handling architectural aspect (Figure 16), this requires global reasoning, because the architect has to check the exceptions raised or received by a number of existing operations in different interfaces of four components: Data_Manager, Business_Rules, Distribution_Manager and GUI_Elements (Figure 14). With the support of the persistence architectural aspect (Figure 16), only modular reasoning is required, because just checking the template the architect can discover the complete structure of exception handling, and, as consequence, find out that: (i) RepositoryException must be handled by all included operations; (ii) TransactionException must be handled by the operations inserted in InfoServices, ManageInfo, DistributedInfoServices and ManageDistributedInfo interfaces; and (iii) CommunicationException must be handled by DistributedInfoServices and ManageDistributedInfo interfaces.

An aspect-oriented approach enriches the knowledge embedded in architectural models. We explicitly model the implications of broadly scoped properties, in the same way we model components, interfaces, processes, or a design space of possible architectural solutions. This externalizes architectural knowledge present in a development team or organization, and is the basis for reuse. With the support of the architectural template the knowledge about Security is explicitly expressed and include all the information about it. This modular representation of the security concern also allows the reuse of this concern in other composition. For instance, to reuse the security concern of Figure 4 in the Health Watcher system the architect has to adapt the composition rules to the new scenario and to adjust the reasoning description. Without the template, the security concern would be scattered and tangled over the context-sensitive tourist guide and it cannot be reused in other systems.

DOI: 10.1002/spe

We believe that the effort required to grasp the proposed templates is not a major bottleneck because they use the concern-specific terminology to describe the effects of the architectural decisions. Anybody can read the templates and respective composition rules in Figures 3 and 4, and understand how the team developed them. The architects do not need to change the way that they work while expressing architectural aspects. The aspectual templates can be seen as a complementary architectural view in addition to the views commonly used by the architects. For instance, no additional effort or extra technical ability is required from the architect to specify an architectural aspect for the Performance concern of the Tourist Guide architecture (Figure 2). In addition, because the template provides a localized abstraction, it facilitates the specification of the architectural decisions and also the composition rules and reasoning. Without the template, the specification of the composition between the Performance architectural aspect and navigate, ext_service, and get_info methods would be spread in different places: navigator and information_retrieval components. It is important to highlight that the idea is that the architect should not work directly with low-level mapping rules for most of the activities. He or she should mainly work with high-level composition rules and with tools, such as COMET (Section 4.1), that support the generation and maintenance of mapping rules.

The challenge of reasoning about architectural decisions stems also from understanding how the decisions related to a concern interact with the decisions related to other concerns. One of the limitations of architectural aspects is that an aspectual template deals with a single concern at a time and does not systematically document its concern interactions with other concerns. However, the analysis of the mapping rules generated from all templates allows the detection of architecture elements that are impacted shared by more than one concern architectural aspect. We can say that these elements represent architectural decisions shared by concerns. For instance, shared architecture elements represent a kind of concern interaction. In the Health Watcher architecture, for example, persistence and exception handling concerns interact with each other, because the system handles two persistence-specific exceptional events: repositoryException and transactionException. These two exceptions appear in the mapping of both Persistence and Exception Handling architectural aspects (Figures 17 and 18, respectively). If either concern is removed from the architecture specification, both exceptions have to be removed. In the future we plan to extend the aspectual template to systematic document interaction among concerns.

6. FUTURE DIRECTIONS AND RELATED WORK

Nowadays companies rely on architectural design reviews as critical points. Architects recognize the importance of making explicit assumptions [1] and tradeoffs within the architectural design space. However, the management of broadly scoped architectural concerns is still made in an idiosyncratic fashion, with limited support for their modular and compositional reasoning. Research on software architecture will certainly have to face this problem, and the marriage of architecture design and aspect orientation might potentially play a key role to address this challenge at different levels. Section 6.1 discusses future trends related to this marriage and Section 6.2 presents work already done within the area of software architecture that is related to architectural aspects.

6.1. Future directions

The crosscutting nature of architectural decisions can manifest in several ways. As a result, architectural aspects require proper mechanisms and notations to identify, represent and compose them. There are some modelling approaches to exclusively support the explicit description of architectural aspects at the logical view [28, 35, 36, 40]. To the best of our knowledge, there is no work in the literature that provides modularity mechanisms to capture crosscutting decisions in multiple architectural views, as we have presented in this paper. However, our proposal does not tackle all the possible architectural views being used in a single architecture. Because the architecture of a system is represented by general and domain-relevant views, each providing a distinct perspective of the system, the crosscutting concerns must be also modularly represented in the multiview scenario.

DOI: 10.1002/spe

Because the architecture of a system are represented by several views, each providing a distinct perspective of the system, the crosscutting concerns must be also modularly represented in the multiview scenario. There is no research work that copes with an aspect-oriented architectural view and the provision of multiview 'weavers', which automate their composition [36]. Such a view would simplify the architecting process and give a better picture of the system's overall structure. In addition, there is a need for the development of methodologies and tools to bridge the gap between the decisions specification and architecture descriptions based on ADLs. How to represent the architectural decisions at the ADL level is still a major challenge to software engineers. Although recently various proposals [28, 40] that integrate aspect-orientation and ADLs have emerged, they do not cope with abstractions and mechanisms to represent crosscutting architectural choices. Some works extend the component-connector specifications with new elements to represent architectural aspects and composition rules as first-class elements.

It is almost always cost-effective to assess the crosscutting design choices as early as possible in the life cycle. Thus, to foster the benefits of more modular software architectures, we also need architecture design analysis methods and tools to evaluate if the architecture reflects a proper modularization and composition of architectural aspects. Traditional methods for architecture assessment, such as ATAM [19], can be extended to deal with those issues. Tekinerdogan [41] provides a first step in that direction by defining an ATAM extension to support the identification of candidates for architectural aspects.

6.2. Related work

The architectural perspectives approach [8] is closely related to our work in the sense that it also considers crosscutting concerns at software architecture specification. Architectural perspectives provide a framework for structuring about how to design systems to achieve a particular quality attribute. An architectural perspective attempts at providing advice relating to the cross view concerns of a particular quality attribute, such as security. It includes activities, checklists, tactics and guidelines to guide the process of ensuring that a system exhibits a particular set of closely related quality properties that require consideration across a number of the system's architectural views. However, the use of a perspective does not explicitly record the architectural decisions. Therefore, the aspectual templates that we propose in this work can be complementarily used to record the architectural decisions (and their rationale) made as a result of applying a perspective. Moreover, architectural perspectives are only about concerns related to quality attributes, whereas architectural aspects can include other kinds of concerns, such persistence.

Architectural tactics [20, 21] are also related to our work. An architectural tactic is a characterization of architectural decisions that are used to achieve a desired quality attribute response. For instance, *Break the dependency chain* is a key modifiability tactic that prescribes inserting an intermediary between the publisher and consumer of data and service to prevent propagation of change. The decisions associated to an architectural tactic can impact different parts (and views) of a system architecture specification. Nevertheless, likewise architectural perspectives, the architectural tactics approach does not provide a support for recording the derived architectural decisions and for mapping the decisions to a language that express the compositional relationship among the architectural elements. In fact, architectural perspectives (mentioned before) embrace and extend tactics by providing advice relating to what the architect should know, do and be aware of, and the specific solution advice provided by an architectural tactic [8]. An architectural perspective can include a set of architectural tactics.

More recently, Bass *et al.* [9] claimed that the design decisions derived from an architectural tactic can be viewed as an architectural aspect. In other words, each use of a tactic can be considered as an architectural aspect, where the join points are the places in the architecture where it was applied. They defined architectural join points as well-defined points in the specification of the software architecture. Architectural point cuts are means of referring to collections of architectural join points. An architectural advice is a specification of transformations to perform at architectural join points. Architectural aspects are architectural views consisting of architectural point cuts and architectural advices. This definition is based on the AspectJ programming language [37] terms.

DOI: 10.1002/spe

Nonetheless, they do not define a systematic way for describing an architectural aspect. Besides, this approach is also restricted to concerns related to quality attributes. The architectural templates and the associated mapping rules that allow the translation of the decisions to different architectural views go beyond the use of architectural tactics to guide the identification of candidate architectural aspects.

Pinto and Fuentes [44] proposed an XML-based aspect-oriented ADL called AO-ADL. The structural organization of AO-ADL is based on the fact that the main difference of architectural crosscutting and noncrosscutting concerns is in the role they play in a particular composition binding and not in the internal behaviour itself. Therefore, AO-ADL does not include a new element to model aspects. Components in AO-ADL model either crosscutting or non-crosscutting behaviour. This is called a symmetric approach. Thus, a component is considered an aspect when it participates in an aspectual interaction. In this context, another contribution of AO-ADL is the extension of the semantic of conventional connectors to represent the crosscutting effect of 'aspectual' components. This means that AO-ADL connectors provide support to describe not only typical communication as in traditional ADLs, but also crosscutting influence among components. AO-ADL, similarly to other aspect-oriented ADLs, only provides mechanisms to represent the component-and-connector view of an architecture. It is not able to describe the influence of architectural concerns on different views.

7. CONCLUSIONS

Architectural decisions are in the heart of the software development process because they provide the bridge between the problem space and the solution space. The promotion of modular and compositional reasoning about architectural decisions is essential to help software developers to understand if they have an architecture compatible with their requirements. It is also a critical success factor for further system design and implementation. However, the broadly scoped nature of early design choices imposes a number of problems to software engineers. In fact, architectural crosscutting concerns are even more challenging than implementation crosscutting concerns. While the latter typically impacts a single artefact (source code) often based on a single programming language, crosscutting concerns at the architectural level impacts a multitude of views with heterogeneous representations. Using only conventional approaches architects often get in trouble because important influences are scattered and tangled in the architectural views.

We proposed an aspect-oriented approach for documenting crosscutting high-level design decisions and supporting modular and compositional reasoning about them. We defined an abstraction called architectural aspect. An architectural aspect is represented by a template that captures the architectural decisions related to a broadly scoped concern, which otherwise would be scattered and tangled over the architecture description and its multiple views. We evaluated the usefulness of architectural aspects in the light of two case studies from different application domains. On the basis of these cases studies, we conclude that the proposed technique is promising to improve the modular and compositional reasoning of architectural crosscutting choices, and, as a consequence, enhance architecture evolvability, modularity assessment and promote knowledge reuse. We have developed a tool called COMET (Section 4.1), which provides automated support for the essential elements of our aspectual templates. We also plan to undertake other case studies to investigate whether and how the aspectual templates need to be extended for explicitly documenting issues related to interactions between crosscutting architectural decisions. However, this open research question is not limited to our work. In fact, this is the next challenge to be addressed by most of the existing aspect-oriented languages, whether targeted either at the architectural or implementation stage.

On the basis of our experience, AOSD techniques can certainly help organizations to improve their state of practice of software architecture. They support software architects with enhanced modular and compositional reasoning, which are imperative throughout all the software development phases. They also complement existing architecture-centric development approaches, both upstream and downstream. Upstream, aspect-oriented abstractions provide a natural way to modularize and

DOI: 10.1002/spe

compose decisions that are directly influenced by broadly scoped concerns coming from the requirements. At the same time, downstream, explicit representation of architectural aspects facilitates the satisfaction of top-level crosscutting decisions at the detailed design and implementation stages.

ACKNOWLEDGEMENTS

This work was partially supported by European Commission Grant IST-2-004349: European Network of Excellence on AOSD (AOSD-Europe). Claudio is also supported by CNPq: National Institute of Science and Technology for Software Engineering (grant 573964/2008-4) and Universal Project (grant 480374/2009-0); and CAPES: PROCAD-NF (grant 720/2010) . Alessandro is supported by FAPERJ: distinguished scientist (grant E-26/102.211/2009) and DANSis project (grant E-26/111.152/2011); CNPq: productivity scholarship (grant 305526/2009-0) and Universal Project (grants 483882/2009-7, 483699/2009-8, 485348/2011-0); CAPES: international collaboration scheme (grant 5688-09), PROCAD-NF (grant 720/2010); and PUC-Rio (productivity grant). Thais is supported by CNPq: productivity scholarship (grant 307269/2010-8) and PDI (grant 560266/2010-3).

REFERENCES

1. Lago P, van Vliet H. Explicit assumptions enrich architectural models. *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, 2005; 206–214.
2. Louridas P, Loucopoulos P. A generic model for reflective design. *ACM (TOSEM)* 2000; **9**(2):199–237.
3. Kiczales G, Mezini M. Aspect-oriented programming and modular reasoning. *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, 2005; 49–58.
4. Rashid A, Moreira A. Domain models are not aspect free. *Proceedings of MoDELS/UML, Springer Lecture Notes in Computer Science, 4199*, 2006; 155–169.
5. Kruchten P. Architectural blueprints – The "**4 + 1**" view model of software architecture. *IEEE Software* November, 1995; **12**(6):42–50.
6. Shaw M, Garlan D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc.: Upper Saddle River, NJ, USA, 1996.
7. Tyree J, Akerman A. Architecture decisions: Demystifying architecture. *IEEE Software* 2005; **22**(2):19–27.
8. Woods E, Rozanski N. Using architectural perspectives. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA '05) - Volume 00 (November 06 – 10, 2005). IEEE Computer Society*, Washington, DC, 2005; 25–35.
9. Bass L, Klein M, Northrop L. Identifying aspects using architectural reasoning. *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, AOSD'04*, 2004; 50–56.
10. Molesini A, Garcia A, Chavez C, Batista T. On the quantitative analysis of architecture stability in aspectual decompositions. *Proceedings of the 7th Working IEEE/IFIP Conference on Software Architecture (WICSA'08)*, Vancouver, BC, Canada, 2008; 29–38.
11. Greenwood P, Bartolomei T, Figueiredo E, Dosea M, Garcia A, Cacho N, Sant'Anna C, Soares S, Borba P, Kulesza U, Rashid A. On the impact of aspectual decompositions on design stability: An empirical study. *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP.07)*, Berlin, Germany, 2007; 176–200.
12. Sant'Anna C, Figueiredo E, Garcia A, Lucena C. On the modularity of software architectures: A concern-driven measurement framework. *Proceedings of the 1st European Conference on Software Architecture*, Madrid, Spain, September 24-26, 2007; 207–224.
13. Filman R, *et al. Aspect-Oriented Software Development*. Addison-Wesley: Boston, MA, 2004.
14. Kiczales G, *et al.* Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, 1997; 220–242.
15. Rashid A, Moreira A, Araújo J. Modularization and composition of aspectual requirements. *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, 2003; 11–20.
16. Sampaio A, Greenwood P, Garcia A, Rashid A. A comparative study of aspect-oriented requirements engineering approaches. *Proceedings of the International Symposium on Empirical Software Engineering and Measurement, ESEM.07*, Madrid, Spain, 2007; 166–175.
17. Davies N, *et al.* Using and determining location in a context-sensitive tour guide. *IEEE Computer* 2001; **34**(8):35–41.
18. Moreira A, Rashid A, Araujo J. Multi-dimensional separation of concerns in requirements engineering. *International Conference on Requirements Engineering (RE), IEEE Computer Society*, 2005; 285–296.
19. Clements P, Kazman R, Klein M. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional: Boston, MA, 2002.
20. Bachmann F, Bass L, Klein M. *Deriving Architectural Tactics: A step towards methodical architectural design (CMU/SEI-2003-TR-004)*. Software Engineering Institute, Carnegie Mellon University: Pittsburgh, PA, 2003.
21. Bass L, Clements P, Kazman R. *Software Architecture in Practice*, (2nd edn). Addison Wesley: Boston, MA, 2003.
22. Sant'Anna C, Figueiredo E, Garcia A, Lucena C. On the modularity assessment of software architectures: Do my architectural concerns count? *Proceedings of the International Workshop on Aspects in Architecture Descriptions*

DOI: 10.1002/spe

(AARCH.07), International Conference on Aspect-Oriented Software Development (AOSD'07), Vancouver, Canada, 2007.
23. Soares S, *et al.* Implementing distribution and persistence aspects with aspectJ. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'02)*, 2002; 174–190.
24. Filho F, Garcia A, Rubira C. Extracting error handling to aspects: A cookbook. *Proceedings of the 23rd International Conference on Software Maintenance (ICSM'07)*, Paris, France, 2007; 134–143.
25. Kulesza U, Sant'Anna C, Garcia A, Coelho R, Staa A, Lucena C. Quantifying the effects of aspect-oriented programming: A maintenance study. *Proceedings of the 22nd International Conference on Software Maintenance (ICSM'06)*, Philadelphia, USA, 2006; 223–233.
26. Silva L, *et al.* On the symbiosis of aspect-oriented requirements and architectural descriptions. *10th Workshop on Early Aspects - Aspect-Oriented Requirements Engineering and Architecture Design, International Conference on Aspect-Oriented Software Development (AOSD'07)*, Vancouver, Canada, 2007; 75–93.
27. Buschmann F, *et al. Pattern-Oriented Software Architecture: A system of Patterns*. John Wiley: New York, NY, 1996.
28. Batista T, Chavez C, Garcia A, Sant'Anna C, Kulesza U, Rashid A, Filho F. Reflections on architectural connection: Seven issues on aspects and ADLs. *Proceedings of the International Workshop on Early aspects at ICSE'06*, Shanghai, China, 2006; 3–10.
29. Cacho N, Sant'Anna C, Garcia A, Batista T, Lucena C. Composing design patterns: A scalability study of aspect-oriented programming. *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, Bonn, Germany, 2006; 109–121.
30. Garcia A, *et al.* Modularizing design patterns with aspects: A quantitative study. *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, Chicago, USA, 2005; 3–14.
31. Kulesza U, Alves V, Garcia A, Lucena C, Borba P. Improving extensibility of object-oriented frameworks with AOP. *Proceedings of the 9th International Conference on Software Reuse (ICSR'06), Springer, LNCS*, Torino, Italy, 2006; 231–245.
32. Rashid A, Chitchyan R. Persistence as an aspect. *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, USA, 2003; 120–129.
33. Filho F, Rubira C, Ferreira R, Garcia A. Aspectizing exception handling: a quantitative study. In *Advanced Topics in Exception Handling Techniques, LNCS 4119, Springer*, 2006; 255–274.
34. Figueiredo E, Silva B, Sant'Anna C, Garcia A, Whittle J, Nunes D. Crosscutting patterns and design stability: An exploratory analysis. *Proceedings of the 17th IEEE International Conference on Program Comprehension*, Vancouver, May 2009; 138–147.
35. Krechetov I, Tekinerdogan B, Garcia A, Chavez C, Kulesza U. Towards an integrated aspect-oriented modeling approach for software architecture design. *8th Workshop on Aspect-Oriented Modelling (AOM.06), International Conference on Aspect-Oriented Software Development (AOSD'06)*, Bonn, Germany, 2006.
36. Chitchyan R, Rashid A, Sawyer P, Garcia A, Pinto M, Tekinerdogan B, Clarke SJ. A survey of analysis and design approaches. *AOSD-Europe Report D11*, 2005.
37. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold W. An overview of aspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01), Springer-Verlag*, 2001; 327–355.
38. Sant'Anna C, *et al.* On the quantitative assessment of modular multi-agent system architectures. *NetObjectDays (MASSA)*, 2006.
39. Sant'Anna C. On the modularity of aspect-oriented design: A concern-driven measurement approach. *PhD Thesis*, Computer Science Department, PUC-Rio, Brazil, April 2008.
40. Cuesta C, *et al.* Architectural aspects of architectural aspects. *2nd European Workshop on Software Architecture (EWSA), LNCS 3527*, 2005; 247–262.
41. Tekinerdogan B. ASAAM: Aspectual software architecture analysis method. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, Norway, 2004; 5–14.
42. Magee J, Dulay N, Eisenbach S, Kramer J. Specifying distributed software architectures. *Proceedings of the 5th European Software Engineering Conference (ESEC'95), LNCS 989*, Sitges, Spain, 1995; 137–153.
43. Magee J, Kramer J. Dynamic structure in software architectures. *SIGSOFT '96 Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering. ACM New York*, NY, USA, 1996; 3–14.
44. Pinto M, Fuentes L. AO-ADL: An ADL for describing aspect-oriented architectures. *Early Aspect Workshop at AOSD'07*, 2007; 94–114.

DOI: 10.1002/spe