Available online at www.sciencedirect.com

ScienceDirect

The Journal of Systems and Software 79 (2006) 1370–1378

The Journal of Systems and Software

www.elsevier.com/locate/jss

An architectural pattern for non-functional dependability requirements

Lihua Xu *, Hadar Ziv, Thomas A. Alspaugh, Debra J. Richardson

University of California, Irvine, Donald Bren School of Information and Computer Sciences, Department of Informatics, Irvine, CA 92697-3425, USA

Received 15 October 2005; received in revised form 20 December 2005; accepted 1 February 2006
Available online 14 June 2006

Abstract

We address the research question of transforming dependability requirements into corresponding software architecture constructs, by proposing first that dependability needs can be classified into three types of requirements and second, an architectural pattern that allows requirements engineers and architects to map the three types of dependability requirements into three corresponding types of architectural components. The proposed pattern is general enough to work with existing requirements techniques and existing software architectural styles, including enterprise and product-line architectures.
© 2006 Elsevier Inc. All rights reserved.

Keywords: Software architecture; Architectural patterns; Aspects; Non-functional requirements

1. Introduction

System dependability has become an umbrella term for the collection of high-assurance and trustworthiness qualities desired of a software system. In their detailed taxonomy of dependability and security (Avizienis et al., 2004), the authors define and distinguish dependability as subsuming availability, reliability, safety, integrity, and maintainability, while security is comprised of confidentiality, availability, and integrity. Other definitions of dependability exist that include additional software "ilities" such as reparability, survivability and fault tolerance (Sommerville, 2004). The stakeholders of a specific system under development (SUD) have specific goals, priorities, and emphases regarding its desired dependability and security attributes. Such high-level goals must be further refined to lower, more precise levels of detail for that SUD. The exact collection of desirable qualities and the mechanisms by which they are specified and decomposed vary considerably among different systems. A system expected to be ultra-reliable, for example, will undoubtedly have more detailed and more precise dependability objectives.

There is tremendous interest and a growing body of work in the emerging discipline of architecture-based software development, in which the high-level structure and behavior of a large software system is specified and designed in terms of components and their connections. There have been significant advances in the field, including architectural design and evaluation methods, reusable artifacts such as architectural patterns and frameworks, and different architecture styles for different problem domains (Taylor et al., 1996). Of particular interest to us, progress was made in the transition from business goals and user requirements to software architectures (Chung et al., 1999; Lamsweerde, 2003) and there is growing consensus that complex stakeholder goals should map to multiple views in architectural designs (Clements et al., 2002). Still, the challenge of describing software architectures driven by original requirements, especially non-functional requirements, remains largely unsolved; we are, in fact, at a relatively early stage of addressing it. In this paper we build upon existing works in architecture-based software development by proposing a reusable architecture pattern, independent of domain-specific architectural styles, that explicitly considers and addresses non-functional requirements.

* Corresponding author. Tel.: +1 949 824 4043.
E-mail addresses: lihuax@ics.uci.edu (L. Xu), ziv@ics.uci.edu (H. Ziv), alspaugh@ics.uci.edu (T.A. Alspaugh), djr@ics.uci.edu (D.J. Richardson).

0164-1212/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2006.02.061

In order to build complex systems predictably and reliably using architectures, an architecture must be modeled and analyzed not only to satisfy functional requirements, but also in the context of its non-functional requirements. We support this approach by proposing an architectural pattern that focuses on modeling identified software requirements in architectures. In other words, we assume that requirements have been identified and refined, and that the next challenge in the development process is realizing and transforming the identified requirements into software architectures.

The motivation of our work is two-fold:

(1) To support early and explicit specification of non-functional requirements during requirements gathering, followed by design of corresponding software architectures, thus offering improved separation of functional and non-functional concerns at the architectural level; and
(2) To support architectural analysis with respect to NFRs early in the software lifecycle, hence establishing confidence that the architecture style and designed architecture are properly chosen, before it is implemented.

Unfortunately, naive modeling or mapping of NFRs into architectural components and connectors using existing architectural styles is not trivial and in many cases not possible. This is due to the fact that rules and constraints defining the architectural style would typically disallow or prevent such components and connectors from being added to the software architecture. Many NFRs, for example, are cross-cutting concerns that should affect potentially many components and connectors in the architecture being designed. As presented in Section 3.1 below, a component connected by connectors to multiple components in multiple layers of the architecture may be best-suited to address this NFR; however, such component is illegal in at least some architectural styles, for example, it violates the rules and constraints defining the C2 architectural style (Taylor et al., 1996).

We propose an architectural pattern that, independent of any architectural style or choice of architecture description language, includes three types of architectural components and one type of architectural connector. Bridging from requirements to architectures is explained in two steps: First, in Section 2, software requirements are classified into three types, namely functional requirements (FRs) and two distinct types of non-functional requirements (NFRs). Then, Section 3 presents the architectural pattern, detailing the three types of architectural components, namely (1) those corresponding to FRs; (2) those corresponding to *operationalizable non-functional requirements* (ONFRs); and (3) those corresponding to *checkable non-functional requirements* (CNFRs), as well as adding a new architectural connector type, captured in an XML Binder, to the existing connectors in the underlying architecture. Section 4 provides an example of applying the proposed architectural pattern to an application designed using the C2 architectural style. The fifth and final section summarizes the contributions of this paper and discusses related and future work.

2. Classification of non-functional requirements

High-level stakeholder goals are refined into specific, concrete sub-goals which trusted, dependable systems must meet. These sub-goals combine and are instrumental in achieving the stakeholder goals. Goals exist at many levels of abstraction, and a high-level goal is often repeatedly refined into lower-level goals (Alspaugh et al., 2005). This yields a requirements goal graph expressing the results of requirements analysis for the SUD (Lamsweerde, 2003). Fig. 1 depicts part of a possible goal-refinement graph for dependability properties.

Leaf nodes in the goal-refinement graph are typically "PROVIDE" goals—i.e., goals corresponding to functional requirements, transformed into requirements specifying services to be provided and behaviors to be exhibited by the system. Non-functional requirements (NFRs), on the other hand, delineate how well the system should do something. While they occasionally correspond to a PROVIDE goal, NFRs are more often associated with "PREVENT" goals, i.e., stakeholder needs and expectations that harmful, malicious, or otherwise undesirable behaviors will either be preventable or at least recoverable. Since dependability typically includes non-functional quality attributes, such as safety, security, and survivability, we focus here on NFRs for those quality attributes. Our research is aimed at improving system dependability and trustworthiness by improving the modeling, analysis and testing of such NFRs. In this paper we discuss only the modeling of dependability NFRs in software architectures.

Several attempts have been made to systematically build software architectural models from both functional and non-functional requirements (Brandozzi and Perry, 2003;

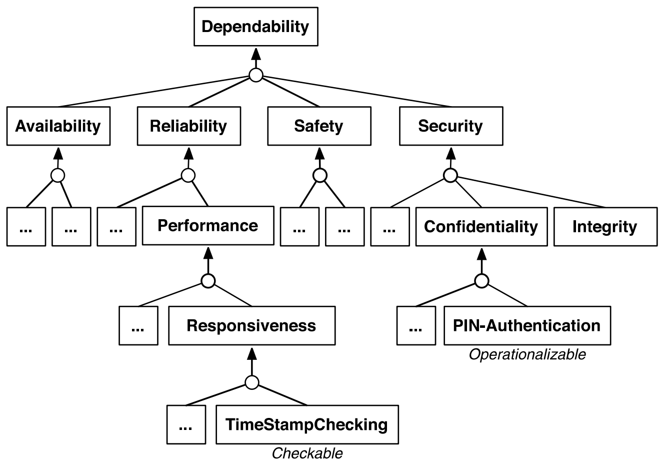

Fig. 1. Partial refinement graph for the dependability NFR.

Cysneiros et al., 2001; Cysneiros and Leite, 2004; Gross and Yu, 2001; Lamsweerde, 2003). Efforts toward providing a means for modeling NFRs at the software architecture level have also been made, such as the work on techniques for representing NFRs as early aspects (WEA, 2002; Xu et al., 2005). Of particular interest is the NFR framework by Mylopoulos et al. (Chung et al., 2000; Mylopoulos et al., 1992; Yu et al., 2004) that provides a comprehensive methodology for NFR identification and refinement, and a starting point for integrating functional and non-functional requirements into software architectures. Their framework views NFRs as high-level soft goals that cannot be fully or absolutely satisfied but at best *satisficed*, namely, satisfied within acceptable limits. Soft goals are then refined into lower-level goals, eventually reaching one or more alternative *satisfying methods*, appropriate solutions that satisfy the NFR within certain limits (Mylopoulos et al., 1992).

In this paper, we extend the work by Mylopoulos and others by further classifying NFRs into two distinct types:

(1) NFRs that are *operationalizable*—similar to Mylopoulos et al., these are NFRs that, upon decomposition and refinement, are realized by functional components in the software architecture. For example, "Security" is first refined into "Confidentiality" and then further refined into "PIN-authentication," and this last can be operationalized.
(2) NFRs that are not operationalized into functional components but are satisfied in other ways during the software development process, such as by choosing a specific engineering solution that meets the required properties. These NFRs can be regarded as *checkable*—in other words, the system can include components that monitor functional behavior to check and verify that the desired quality properties are met. For example, "performance" for a particular system can be refined into "check database workload" and further refined into "monitor and report number of queries per second for database", which can be operationalized into a check that the original performance goal is met.

Fig. 2 shows a goal refinement hierarchy containing both types of NFR. Take the example of "Provide good usability" (Usability in Fig. 2) as a high level goal of a SUD. One of its sub-goals might be "Provide a good user interface" (UserInterfaceUsability). Any of several alternative mechanisms could be chosen to satisfy this goal, as in the example in Fig. 2. Here we show that the requirements analyst could choose to further refine the goal into "Allow user to customize the preference interface to some extent" (Customizable) to satisfy the goal of usability, here further refined into the operationalizable NFRs "Make the help file available" (HelpAvailable) and "Allow user to choose a different language" (LanguageSelection). On the other hand, the analyst could also OR-refine UserInterfaceUsability into a subgoal "Collect information every time user makes a mistake in using the interface" (MistakeCollection) that contributes to satisficing Usability; if the MistakeCollection goal is satisfied, then we accept that the Usability goal has been satisficed.

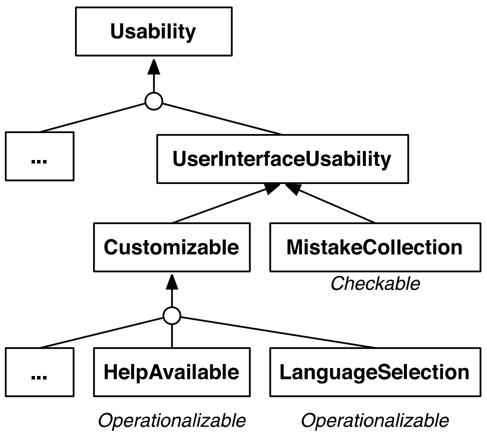

Fig. 2. Partial refinement graph for the usability NFR.

3. The architectural pattern

In the book Pattern Oriented Software Architecture (Buschmann et al., 1996), the authors define an architectural pattern as

"expressing a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them."

Our interpretation considers that the purpose of the software architecture is to ensure that the system meets the stakeholders' goals with respect to various software qualities.

As mentioned before, our approach addresses the problem of analyzing the designed software architecture against identified non-functional requirements. To avoid confusion, we use *architecture style* to refer to the types of constructs that are used in the designed architecture, while the *architectural pattern* proposed here concentrates more on modeling non-functional requirements in architecture level and linking them with the designed architecture.

Our approach rests on the well-known software engineering principle of *separation of concerns*, and applies it to software architectural design and analysis. We argue that in modeling and analysis, FRs and NFRs should be considered separately, while still modeling and maintaining their many-to-many interrelationships (i.e., one FR is related to and affected by many NFRs, one NFR relates to many FRs, etc.). This follows the same argument used for Aspect-Oriented Development (AOD), namely that without separation of concerns, we face the risk that FRs and NFRs will be modeled and analyzed in a scattered and tangled fashion, and the resulting software architecture will be scattered and tangled as well.

3.1. Architectural components

We propose an architectural pattern consisting of components corresponding to the three types of requirements identified in Section 2 above—functional requirements, operationalizable NFRs, and checkable NFRs—as follows:

(1) *Core functional components*: The first type of architectural component realizes and ultimately implements the system's functional requirements.
(2) *Aspectual components*: The second component type corresponds to ONFRs, which are likely to be represented as aspects at the software design and implementation levels. Aspectual components represent the semantics of those ONFRs; these components correspond to advice tasks in the aspect-oriented world, and are designed to be woven with one or more functional components (i.e., core functional components of type (1) above).
(3) *Monitoring components*: The third component type corresponds to CNFRs specifying how to monitor and check for specific quality attributes, and are distributed over both functional and aspectual components. Monitoring components may be woven with other types of components, or may remain external to the design and implementation of the SUD, for example, residing in an external GUI or a monitoring harness.

3.2. Architectural connectors

The understanding and precise definition of connectors in software architecture have solidified and converged over time. An early paper by (Perry and Wolf, 1992) describes connectors informally as the glue that holds together various pieces of the architecture. A subsequent paper by (Shaw and Clements, 1997) defines a connector as an abstract mechanism that may mediate communication, coordination or cooperation among software components. Later, authors in (Egyed et al., 2000) further defined a connector as an architectural building block used to model the interactions among components as well as the rules governing those interactions. They also presented a taxonomy classifying components into several categories, such as Interfaces, Types, Semantics, and Constraints.

The architectural pattern proposed here views the functional architecture components as one layer, and the aspectual components and monitoring components as another layer. In addition, architectural connectors may be used to capture the semantics of rules for weaving aspectual components and monitoring components into the functional architecture layer. We suggest that connectors be modeled as XML-based binding specifications—similar to those proposed in the Aspect Markup Language (AML) by Lopes and Ngo (2004)—that include instructions on how the component weaving works. Fig. 3 illustrates this architectural pattern. The first layer in the model contains the functional architectural components in some architectural style or language. The second layer captures the semantics of both ONFRs and CNFRs, which represent the constraints designed to be distributed over one or more components in the functional architecture layer. The connectors of the architecture pattern are actually the semantics of weaving rules for each aspectual component or monitoring component, represented as XML Binders.

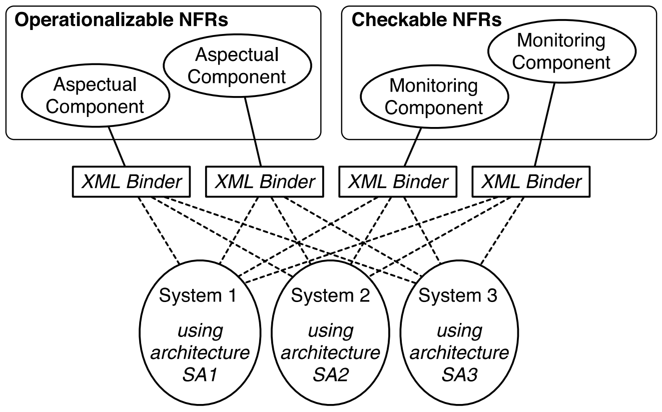

Fig. 3. Architectural pattern modeled from requirements.

Each XML Binder consists of pointcut and interceptor definitions and is designed specifically for a particular aspectual or monitoring component. Each pointcut has a signature pattern that identifies if and where the aspectual or monitoring component will be woven. An interceptor identifies the aspectual or monitoring component and method, and whether it is to be attached before or after the matched locations. If the binder has more than one pointcut, each intercepter can specify the ones it uses.

Fig. 4 shows a template of an architectural connector for our pattern. This connector refers to architectural components only by their names, for example `AspectualMonitoringComponent`, and by names of specific methods in that component, for example `InvokeMethod()`. Thus

```
<xml>
    <Binder id = "xmlBinderID">
        <pointcut id = "pointcutID">
            <or>
                <pointcut type = "component" pattern = "matching conditions" />
            </or>
        </pointcut>

        <interceptor
            Component = "aspectual/MonitoringComponent">
            <advice
                type = "before/After"
                pointcut-refid = "pointcutID"
                interceptor-method = "invokedMethod()" />
        </interceptor>

    </Binder>

</xml>
```

Fig. 4. XML binder example.

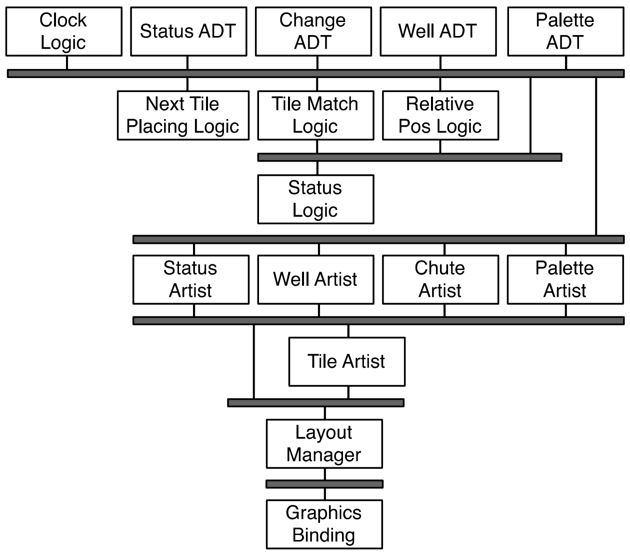

Fig. 5. C2 Architecture for KLAX (Taylor et al., 1996).

the XML Binders are independent of any programming language or other details used in the implementation of the component (see Fig 5).

3.3. Support for enterprise architectures

With the shift in software engineering from development of monolithic, standalone applications to development of componentized enterprise architectures, proper reuse of available components and other architectural elements is crucial to successful enterprise architecture design. We contend that our architectural pattern supports this activity as follows:

- Core components providing functional services can be reused across multiple applications, i.e., they serve as enterprise application components.
- An aspectual component realizing operationalizable NFRs can be woven with one or more core functional components, across multiple applications and even across multiple product-line architectures.
- A component realizing checkable NFRs can monitor one or more core functional components, one or more aspectual components, or any new component resulting from the weaving of such components.
- An XML binder can link any number of components of any type.

Our architectural pattern therefore allows modeling of the various kinds of many-to-many relationships often found in enterprise and product-line architectures.

4. An example with C2 and KLAX

To foster understanding of the architectural pattern, a small example is presented that applies our approach to the C2 architectural style.

A canonical example used for demonstrating the C2 architectural style is a video game called KLAX (Taylor et al., 1996). The game includes three parts: KLAX Chute drops tiles of random colors at random times and locations; KLAX Palette catches tiles coming down the Chute and drops them into the Well; and the Well removes horizontal, vertical, and diagonal sets of three or more consecutive tiles of the same color and collapses down the tiles above them to fill in the newly created empty spaces.

The game calculates the scores accordingly. The C2 architecture designed for KLAX addresses the game's functional requirements and consists of ADT components encapsulating the game's state, logic components, and artist components that maintain the state of a set of abstract graphical objects.

To apply our approach, we first identify requirements of the three types identified above, namely, FRs, ONFRs, and CNFRs, for the KLAX video game. The original C2 architecture for KLAX already captures the functional services provided by the system. By following, for example, the NFR framework of (Mylopoulos et al., 1992), designers are able to identify the ONFRs and CNFRs, based on their chosen satisfying method. We illustrate our approach using one CNFR and one ONFR for the KLAX system (Fig. 6).

4.1. Monitoring components and binders for CNFRs

In Fig. 1 we showed a partial graph of goal refinement for dependability, in which the Performance goal is refined into several sub-goals, including Responsiveness, which in

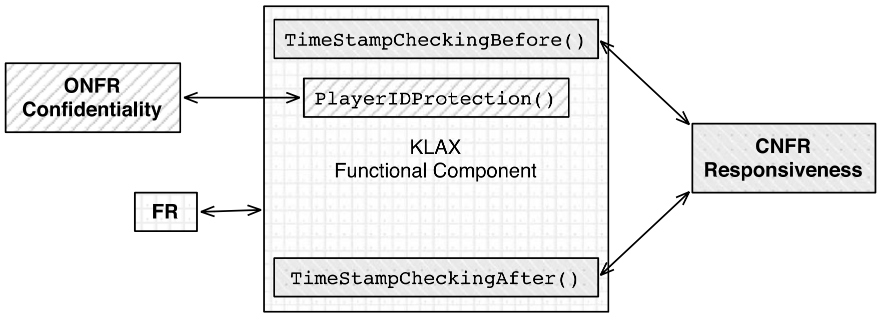

Fig. 6. CNFR and ONFR for the KLAX system and corresponding components.

turn is refined into requirements, including the checking of timestamp data.

We consider the goal Responsiveness to be one whose satisfying method is chosen by designers to be the monitoring component **ResponsivenessInterceptor** that checks whether Responsiveness is satisficed. This component includes **TimeStampCheckingBefore()** and **TimeStampCheckingAfter()**, to provide mechanisms for monitoring the timeslot required by performing each user request. A sketch of code for representing this monitoring component for Responsiveness is shown in Fig. 7.

Fig. 8 shows how the monitoring component for checkable NFR Responsiveness can be woven with the core functional component and the join points that define the matching conditions. The first section defines the XML binder for Responsiveness consisting of two pointcuts and an interceptor. The pointcuts insert monitors into the starting point of each event—the components that prepare and send the request—and the end point of the response—the components that receive the event and update the database according to the request. The interceptor definition uses a nested advice definition to specify that the methods **TimeStampCheckingBefore()** and **TimeStampCheckingAfter()** of the **ResponsivenessInterceptor** monitoring component (shown in Fig. 7) are advices for the pointcuts. The method **TimeStampCheckingBefore()** is invoked just before any *starting* point is executed, and the method **TimeStampCheckingAfter()** is invoked just after any *update* point is executed (see Fig. 9).

```
Component monitor ResponsivenessInterceptor {
      TimeStampCheckingBefore() {
      // the code for gathering starting-time data goes here
            // add one timestamp to the start point of
            // each function where the request has been made
      }
      TimeStampCheckingAfter() {
      // the code for gathering stopping-time data goes here
            // add one timestamp to the end point of
            // each function where the request has been received
            // and updates have been made accordingly
      }
      TimeStampChecking() {
      // the code for verifying the timeslot used for
      // performing the request goes here
            // record the time difference between timestamps
            // resulting from the previous two methods,
            // and verify against the requirement
      }
}
```

Fig. 7. Monitoring component for responsiveness NFR.

```
<xml>
      <Binder id = "responsiveness">
            <and>
                  <pointcut id = "Starting">
                        <or>
                              <pointcut
                                    type = "component"
                                    pattern = "Preparing(event) && Sending(event)" />
                        </or>
                  </pointcut>

                  <pointcut id = "Update">
                        <or>
                              <pointcut
                                    type = "component"
                                    pattern = "Received(event) && Update()" />
                        </or>
                  </pointcut>
            </and>

            <interceptor
                  Component = "ResponsivenessInterceptor">
                  <advice
                        type = "before"
                        pointcut-refid = "Starting"
                        interceptor-method = TimeStampCheckingBefore()" />
                  <advice
                        type = "after"
                        pointcut-refid = "Update"
                        interceptor-method = TimeStampCheckingAfter()" />
            </interceptor>
      </Binder>
</xml>
```

Fig. 8. XML binder for responsiveness NFR.

```
Component aspect ConfidentialityInterceptor {
      PlayerIDProtection() {
            // description for checking PlayerID goes here
      }
      ...
}
```

Fig. 9. Aspectual component for confidentiality NFR.

In this example, we show how one NFR can be represented as a monitoring component, and illustrate the binding rules to weave this particular component into the functional architecture layer. For systems with scattered and tangled crosscutting NFRs or product line architectures, architects can provide their own *pointcut* and *interceptor* definitions accordingly.

4.2. Aspectual components and binders for ONFRs

For aspectual components, we consider the requirement of Confidentiality, namely that only one authenticated player play at a time to ensure that the score earned is attributed correctly. The derived aspectual component **ConfidentialityInterceptor** would include the advising task **PlayerIDAuthentication()**, to prevent any updates by unauthenticated players.

Therefore, the derived aspectual components would include the advising task **PlayerIDProtection()**. A sketch for representing the aspectual component is given below.

Fig. 10 shows the aspectual component that operationalizes the Confidentiality NFR woven with the core functional component and the join points that define the matching conditions. This NFR is operationalized by the **ConfidentialityIntercepter** component. The first section in Fig. 10 defines the Confidentiality binder. Its pointcut picks out every component that receives an event and is preparing to perform some update action. The interceptor definition uses a nested advice definition to specify that the **PlayerIDProtection()** method of the **ConfidentialityInterceptor** aspectual component is the advice for the *update* pointcut. The **PlayerIDProtection()** method will be invoked just before any update point is executed.

```
<xml>
    <Binder id = "confidentiality">
        <pointcut id = "Update">
            <or>
                <pointcut
                    type = "component"
                    pattern = "Received(event) && Preparing(action)" />
            </or>
        </pointcut>

        <interceptor
            Component = "ConfidentialityInterceptor">
            <advice
                type = "before"
                pointcut-refid = "update"
                interceptor-method = "PlayerIDProtection()" />
        </interceptor>
    </Binder>
</xml>
```

Fig. 10. XML binder for confidentiality NFR.

In this example, we showed how one NFR can be operationalized and represented as an aspectual component, and the binding rules for this particular aspectual component with the functional architecture layer. For systems that have scattered and tangled crosscutting NFRs, or product line architectures, architects can define their own pointcut and interceptor definitions accordingly.

Further details on representing aspectual components and their binding rules, including the specific example of the Confidentiality goal, can be found in an earlier paper (Xu et al., 2005).

4.3. Applying the pattern to KLAX

The outcome of applying our architectural pattern to KLAX is illustrated graphically in Fig. 11. The resulting architectural model demonstrates our ability to support:

- Tracing software requirements to the architectural level, including dependability and other NFRs for which this is often difficult if not impossible.

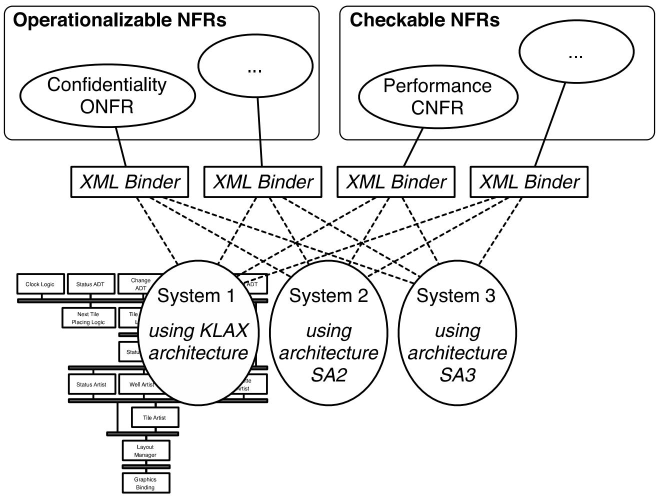

Fig. 11. Applying the pattern to KLAX.

- Untangling the scattered components corresponding to ONFRs and CNFRs, and identifying them as first-class architectural components.
- Early analysis and verification that a designed architecture can be expected to support dependability and other NFRs.
- Reuse of aspectual and monitoring components corresponding to NFRs.

5. Related and future work

The pattern proposed in this paper allows the modeling of dependability NFRs as first class requirements elements during software development, followed by explicit mapping of such NFRs into software architectures, all while embracing traditional architectural design principles for meeting the stated FRs.

This work differs from related work in this area in the following ways:

- Unlike previous work (Barbacci, 2002; Chung et al., 1999; Mylopoulos et al., 1992) that considered NFRs to be an integral part of the system and used them to drive the development process, we consider both FRs and NFRs to be parts of the requirements elements that will be mapped into architectural design elements to be implemented later.
- Our work is more general than previous approaches (Cysneiros et al., 2001; Cysneiros and Leite, 2004) in that the design model does not require particular specific techniques to be used by the designer; instead, it can be used together with any traditional design technique, including architectural styles, design patterns, UML, and so forth.
- Compared to previous work (Brandozzi and Perry, 2003), our work goes a step further in providing clear means and guidance to identify the related core components for each NFR, and to integrate the several types of components.

Yu et al. published an approach for identifying aspects by analysis of a system's V-graph (Yu et al., 2004). A V-graph is a goal graph containing hard goals and soft goals, and also tasks whose completion contribute to the satisfaction or satisficing of hard goals and soft goals in the graph. In this approach, a task that contributes directly to a soft goal is identified as an aspect, crosscutting all the hard goals to which it contributes. In this view, an aspect is a consequence of goal analysis rather than an architectural, design, or implementation decision. However, it is not entirely clear that the allocation of tasks to goals done in this approach does not involve choices that in other approaches would be considered part of these later development phases.

In the aspect-oriented community, our work falls under "Early Aspects" WEA, 2002. In their work on AML (Lopes and Ngo, 2004), the authors strengthened the separation of concerns in Aspect-Oriented Programming (AOP) by delineating an aspect into an advice (a Java class) and a set of rules governing the weaving of that advice into other "core" Java classes (an XML Binder). Similarly, our architectural pattern delineates an architecture-level "concern" into an architectural component (the advice) and an architectural connector (an XML Binder) governing the weaving of that advice into other architectural components. It also adds a new type of component, the monitoring component. Thus, our architectural pattern is directly relevant and applicable to Aspect-Oriented Design (AOD), specifically to addressing the issue of modeling and mapping NFRs to software architectures. We view and use requirements modeling as a form of concern modeling; as such our work is clearly related to the original work on multi-dimensional separation of concerns by (Tarr et al., 1999), followed by concern modeling in COSMOS (Sutton and Rouvellou, 2000; Sutton and Rouvellou, 2002). While their work provides a taxonomy of types of concerns in general, we focus on requirement decomposition, followed by mapping and forward-engineering of requirement models into architectural designs and system implementations. Also, our approach supports analysis of linked software artifacts for traceability and change management, which in turn could benefit the general model of concerns put forth by Sutton and Rouvellou.

Another method for classifying and categorizing aspects—at the architectural component level—is Aspect-Oriented Component Requirements Engineering (AOCRE, Grundy, 1999). Continuing with aspect-oriented architectural approaches, Baniassad and Clarke propose Theme/Doc and Theme/UML in support of aspect-oriented analysis and design, respectively (Baniassad and Clarke, 2004). Their work employs multi-dimensional separation of concerns by separately modeling different concern types, then re-composing them at a later stage using a UML extension called the composition relationship. Compared to the Theme approach, our approach does not assume or rely on any specific architectural modeling language or approach (such as UML), but consequently does not benefit from any language-specific constructs (such as UML extensions). Our approach does not ignore or conflict with existing architectural styles or other approaches to modeling and mapping of NFRs (Gross and Yu, 2001; Lamsweerde, 2003); rather it extends and complements them, making it relatively painless for engineers and architects to adopt and apply.

The pattern offers a multi-dimensional view of software architecture design: the first dimension includes a typical software architecture, following an existing architectural style to structure the system to support required services; the second dimension reflects cross-cutting non-functional dependability requirements, and imposes constraints for making the architecture correspond to and support those NFRs. Finally, an implied third dimension represents the resulting enterprise software architecture design, which satisfies both functional requirements and non-functional requirements. Here, we plan to investigate the possibility of "weaving" the architectural views corresponding to FRs and NFRs into the resulting enterprise architecture.

We anticipate several potential benefits to modeling NFRs in architectural designs. The primary benefit of interest to us is the ability to analyze and verify the software architecture before it implemented. Currently, we are experimenting with modeling and analyzing several NFRs for C2 architectures by extending Argus-I (Dias and Vieira, 2000), a comprehensive set of specification-based analysis tools focusing on both the component and architecture. More specifically, the existing Argus-I toolset supports behavior descriptions for C2 architectures by StateChart specifications. We are trying to augment it with NFR descriptions, which will then be subject to various analyses.

Our future work includes case studies in modeling additional NFRs for KLAX and real-world applications. We plan to develop techniques for analysis and evaluation of architectures where the proposed pattern is applied. Potential techniques include architecture-based testing against non-functional requirements; subsequent mapping of such architectures to programs, in particular aspect-oriented programs; and automated tool support for weaving the new architectural components and connectors into such architectures.

Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful and helpful comments. We also acknowledge members of the Rosatea group at University of California, Irvine, in particular Thomas A. Standish, for their feedback on earlier drafts of this paper.

References

Alspaugh, T.A., Richardson, D.J., Standish, T.A., Ziv, H. 2005. Scenario-driven specification-based testing against goals and requirements. In: 11th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'05). pp. 187–202.

Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secur. Comput. 1 (1), 11–33.

Baniassad, E., Clarke, S., 2004. Theme: an approach for aspect-oriented analysis and design. In: 26th International Conference on Software Engineering (ICSE '04). pp. 158–167.

Barbacci, M.R., 2002. Sei architecture analysis techniques and when to use them. Technical Note CMU/SEI-2002-TN-005, SEI.

Brandozzi, M., Perry, D.E., 2003. From goal-oriented requirements to architectural prescriptions: the preskriptor process. In: Second International Software Requirements to Architectures Workshop (STRAW'03). pp. 107–113.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. Pattern-Oriented Software Architecture: A System of Patterns. Wiley.

Chung, L., Gross, D., Yu, E.S.K., 1999. Architectural design to meet stakeholder requirements. In: WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1). pp. 545–564.

Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J., 2000. Non-functional Requirements in Software Engineering. Springer.

Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., et al., 2002. Documenting Software Architectures: Views and Beyond. Pearson Education.

Cysneiros, L.M., Leite, J.C.S.d.P., 2004. Nonfunctional requirements: from elicitation to conceptual models. IEEE Trans. Softw. Eng. 30 (5), 328–350.

Cysneiros, L.M., Leite, J.C.S.d.P., Neto, J.d.M.S., 2001. A framework for integrating non-functional requirements into conceptual models. Requir. Eng. 6 (2), 97–115.

Dias, M.S., Vieira, M.E.R., 2000. Software architecture analysis based on statechart semantics. In: IWSSD '00: Proceedings of the 10th International Workshop on Software Specification and Design. pp. 133.

Egyed, A., Mehta, N.R., Medvidovic, N., 2000. Software connectors and refinement in family architectures. In: IW-SAPF-3: Proceedings of the International Workshop on Software Architectures for Product Families. pp. 96–106.

Gross, D., Yu, E.S.K., 2001. From non-functional requirements to design through patterns. Requir. Eng. 6 (1), 18–36.

Grundy, J.C., 1999. Aspect-oriented requirements engineering for component-based software systems. In: RE'99. pp. 84–91.

Lamsweerde, A.V., 2003. From system goals to software architecture. In: SFM. pp. 25–43.

Lopes, C.V., Ngo, T.C., 2004. The aspect markup language and its support of aspect plugins. ISR Technical Report UCI-ISR-04-8, University of California, Irvine.

Mylopoulos, J., Chung, L., Nixon, B., 1992. Representing and using nonfunctional requirements: a process-oriented approach. IEEE Trans. Softw. Eng. 18 (6), 483–497.

Perry, D.E., Wolf, A.L., 1992. Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes 17 (4), 40–52.

Shaw, M., Clements, P.C., 1997. A field guide to boxology: preliminary classification of architectural styles for software systems. In: COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference. pp. 6–13.

Sommerville, I., 2004. Software Engineering, 7th Edition. Addison-Wesley.

Sutton, Jr., S.M., Rouvellou, I., 2000. Concerns in the design of a software cache. In: Workshop on Advanced Separation of Concerns (OOPSLA 2000).

Sutton, Jr., S.M., Rouvellou, I., 2002. Modeling of software concerns in cosmos. In: AOSD '02: Proceedings of the 1st international conference on aspect-oriented software development. pp. 127–133.

Tarr, P., Ossher, H., Harrison, W., Sutton, Jr., S.M., 1999. N degrees of separation: multi-dimensional separation of concerns. In: 21st International Conference on Software Engineering (ICSE '99). pp. 107–119.

Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead Jr., E.J., Robbins, J.E., Nies, K.A., et al., 1996. A component- and message-based architectural style for gui software. IEEE Trans. Softw. Eng. 22 (6), 390–406.

WEA, 2002. Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design. Available from: <http://www.early-aspects.net/>.

Xu, L., Ziv, H., Richardson, D., Liu, Z., 2005. Towards modeling non-functional requirements in software architecture. In: Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design.

Yu, Y., Leite, J.C.S.d.P., Mylopoulos, J., 2004. From goals to aspects: Discovering aspects from requirements goal models. In: 12th IEEE International Requirements Engineering Conference (RE'04). pp. 38–47.

Lihua Xu, Ph.D. student in Informatics at UC Irvine Donald Bren School of Information and Computer Sciences, does research on software architecture-based analysis and testing. Her current work concentrates on modeling identified non-functional requirements in software architectures and analyze the architecture against stakeholders' goals. She had several papers published in various conferences, and was awarded Best Paper Award in the Grace Hopper Celebration of Women in Computing (GHC 2004). Xu received her B.S. in Computer Science from PRChina.

Hadar Ziv is a lecturer and research scientist in Informatics, with research interests in software testing, software process, requirements engineering with use cases and scenarios, and uncertainty modeling with Bayesian Networks. His 2005 publications appeared in WADS, ASE/TEFSE, and AOSD/Early Aspects workshops. He was awarded UC Irvine Excellence in Teaching Award in 2003. After receiving his Ph.D. from UC Irvine School of ICS, Ziv served as a consultant, trainer and mentor in several areas of software development to many organizations, including Beckman Coulter, Logicon, Fidelity National Title, and Capital Group Companies.

Thomas A. Alspaugh, Assistant Professor of Informatics, does research on the use of scenarios in requirements and other areas. His current work examines how the implicitly formal aspects of scenarios can be used for automated support and for application in specification-based testing, computed social worlds, and other contexts. Before returning to the academic world, Alspaugh worked in software development at several firms including IBM and Data General, and at the Naval Research Laboratory in Washington, D.C. on the A-7 or Software Cost Reduction project and others. Alspaugh received his Ph.D. in Computer Science from North Carolina State University.

Debra J. Richardson, Professor of Informatics, pioneered research in specification-based testing. Her current work focuses on enabling specification-based testing technology throughout the software lifecycle, from requirements and architecture analysis through operation and evolution. Richardson was appointed Ted and Janice Smith Dean of the Donald Bren School of Information and Computer Sciences in 2002. She has served the research community as Program Co-Chair of IWSSD, General Chair of ISSTA, and General Chair of ASE. Richardson received her B.A. in Mathematics from the University of California–San Diego and the M.S. and Ph.D. in Computer and Information Science at the University of Massachusetts–Amherst.