Handbook of Research on Architectural Trends in Service-Driven Computing

Raja Ramanathan
Independent Researcher, USA

Kirtana Raja
IBM, USA

A volume in the Advances in Systems Analysis, Software Engineering, and High Performance Computing (ASASEHPC) Book Series

Information Science REFERENCE
An Imprint of IGI Global

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA, USA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Handbook of research on architectural trends in service-driven computing / Raja Ramanathan and Kirtana Raja, editors.
pages cm
Includes bibliographical references and index.
ISBN 978-1-4666-6178-3 (hardcover) -- ISBN 978-1-4666-6179-0 (ebook) -- ISBN 978-1-4666-6181-3 (print & perpetual access) 1. Service-oriented architecture (Computer science) 2. Computer network architectures. 3. Enterprise application integration (Computer systems) I. Ramanathan, Raja, 1958- editor. II. Raja, Kirtana, 1987- editor.
TK5105.5828.H356 2014
004.6'54--dc23
2014013829

This book is published in the IGI Global book series Advances in Systems Analysis, Software Engineering, and High Performance Computing (ASASEHPC) (ISSN: 2327-3453; eISSN: 2327-3461)

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 2
Dynamically Reconfigurable Architectures:
An Evaluation of Approaches for Preventing Architectural Violations

Marek Rychly
Brno University of Technology, Czech Republic

ABSTRACT

Dynamic aspects of behavior of software systems in dynamically reconfigurable runtime architectures can result in significant architectural violations during runtime. In such cases, a system's architecture evolves during the runtime according to the actual state of the system's environment, and consequently, runtime reconfigurations may eventually lead to incorrect architecture configurations that were not considered during the system's design phases. These architectural violations are known as architectural erosion or architectural drift, and they contribute to an increasing brittleness of the system, or a lack of its coherence and clarity of its form. This chapter describes and compares possible measures to prevent architectural violations in dynamic service and component models. The aim of this chapter is to evaluate the applicability of those measures in combination with advanced features of reconfigurable runtime architectures such as ad hoc reconfiguration, service or component mobility, composition hierarchy preservation, and architectural aspects.

DOI: 10.4018/978-1-4666-6178-3.ch002

INTRODUCTION

Current information systems tend to be designed as component-based systems and often utilize Service Oriented Architecture (SOA) and Web service technology. The service orientation enables decomposition of a complex software system into a collection of cooperating and autonomous components known as services. These services cooperate with each other to provide a particular functionality of the implemented software system with defined quality.

Loose binding between the services, which represent individual components of a system, enables *runtime reconfigurations* of the system architectures. In other words, it enables creating, destroying, and updating the services, and establishing and destroying their interconnections dynamically at runtime, on demand, and according to various aspects to move the services into different contexts and to different providers (i.e., service mobility). Eventually, a series of reconfigurations contributing to the evolution of the architecture, of a supposedly well-designed system may lead to incorrect architecture configurations that were not considered during the system's design phase. These incorrect configurations are commonly known as *architectural violations*.

This chapter describes and compares possible measures to prevent the architectural violations, as they are used in the current state-of-the-art approaches. The goal is to evaluate applicability of those measures in combination with the advanced features of dynamic architecture such as ad hoc reconfiguration, service or component mobility, composition hierarchy preservation, and architectural aspects.[1] Specific objectives include an introduction to the problems of dynamically reconfigurable runtime architectures, an analysis of the state-of-the-art approaches in this field with focus on the advanced features of dynamic architectures, and the methods to prevent architectural violations.

The chapter is organized as follows. The next section deals with software architecture in general and introduces component-based development and service-oriented architecture with concepts of dynamically reconfigurable runtime architectures. We also describe several important state-of-the-art works dealing with component-based development and component models supporting features of dynamic and mobile architectures. In the following section, we discuss existing problems relating to the support of dynamic and mobile architectures that cause architectural violations in component-based or service-oriented systems.

Then, we outline possible strategic improvements and introduce approaches to prevent the architectural violations in general, and also describe their applications in the current state-of-the-art related works. The next part of the chapter deals with the evaluation of the previously described approaches for preventing architectural violations. More specifically, we analyze compatibility of the approaches with the advanced features of dynamically reconfigurable runtime architectures. Finally, we discuss future research directions such as possibilities of utilization of the advanced features of dynamically reconfigurable runtime architectures including previously described methods of preventing architectural violations in implementations of service-oriented architectures.

BACKGROUND

According to IEEE (2000), software architecture is defined as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution. Another definition by Bass et al. (2003) adds that the architecture describes only externally visible properties of components, i.e., it is an abstraction of a system that suppresses details of components, except for services published by interfaces, relationships to environment of the components, and their externally observable behavior.

Oquendo (2004) distinguished three types of software architectures according to their evolution which depends on changes to their environment: static architecture, dynamic architecture, and mobile architecture. The last one is also known as a fully dynamic architecture.

Architecture of a software system is the *static architecture* if there are no changes to the system's structure during runtime. After initialization of the system, there are no new connections between the system's components and existing connections are not destroyed.

In the *dynamic architecture*, there exist rules of evolution for a software system in time (also called "dynamics"). The system's components and connections are created and destroyed during runtime according to the rules from design-time.

Finally, the *mobile architecture* is a dynamic architecture of a system where the system's components can change their context in the system's logical structure during its execution (also called "component mobility") according to rules from design-time and functional requirements.

Component-Based Development and Service-Oriented Architecture

Component-based development (CBD) is a software development methodology, which is strongly oriented to composition and reusability in a software system's architecture (Szyperski, 2002). In CBD, a component-based system is composed of components, which are self-contained entities accessible through well-defined interfaces. A connection of compatible interfaces of cooperating components is realized via their bindings (also known as connectors). Actual organization of interconnected components is called configuration.

Service-oriented architecture (SOA) (Erl, 2005) represents a model in which functionality is decomposed into small distinct components, known as services, which can be distributed over a network and can be combined together and reused to create business applications. Services are defined as autonomous platform-independent entities enabling access to their capabilities via their provided interfaces.

CBD and SOA are based on the similar principles. Component-based and service-oriented systems are composed of components and services, respectively, which are interconnected into configurations and which can be further decomposed. However, while service design in SOA is business-oriented based on business processes which are realized by the services, components in CBD are implementation-oriented and usually need not respect any business rules or aims.

Service-oriented systems are defined by services, their interfaces, implementations, orchestrations, and resulting choreography. Component-based systems are defined only by their initial configuration, component hierarchy (composition where composite component encapsulates another composite or atomic indecomposable component), and component behavior.

Reconfiguration in Dynamic and Mobile Architectures

For both CBD and SOA, a static architecture has only one way to connect components or services and their connectors or binding into a resulting system, i.e., there is only one configuration. Dynamic and mobile architectures enable software systems to change their architecture during runtime; in other words, a *reconfiguration,* or a runtime modification of the configuration.

Especially in the case of SOA, loose binding between the services, which represent individual components of a service-driven system, allows runtime reconfigurations of the system's architecture. This is the ability to create, destroy, and update the services, and to establish and destroy their interconnections dynamically at runtime, on demand, and according to various aspects; and to move the services into different contexts and to different providers (i.e., service mobility).

As it was mentioned in the introduction, a series of reconfigurations, which represent an incremental evolution of a system's architecture, may eventually lead to incorrect architecture configurations that were not considered during the design phase of a supposedly well-designed system.

The problem of evolving architectures was introduced by Perry & Wolf (1992) and is known as the problem of architectural drift and architectural erosion. A*rchitectural drift* is defined as insensitivity about a system's architecture that, with increasing evolution, leads to its inadaptability and a lack of coherence and clarity of form. A*rchitectural erosion* is defined as violations of a system's architecture that lead to significant problems in the system and contribute to its increasing brittleness. It may be caused by the unrestrained evolution of the architecture as well as violations of an architecture, which has become obscured due to architectural drift. In this chapter, architectural drift and architectural erosion are collectively referred to by the single term, *architectural violations*.

State of the Art in Dynamic and Mobile Architectures

Component-based systems can be modeled as component models or described in architecture description languages. *Component models* are specific meta-models of software architectures supporting the component-based development. According to Lau and Wang (2005), the component models should define syntax, semantics, and composition of components. They are systems of rules for components, connectors, and configurations, and are the rules for changes according to the dynamic architecture (rules for reconfigurations). *Architecture description languages* (ADLs) (Vestal, 1993), are languages for describing software system architectures. They focus on high-level structures of overall applications rather than implementation details of specific source modules. The ADLs can be parts of component models, where they are used for description of a software system's architecture in terms of the component models. Alternatively, ADLs can be realized without the component models, based directly on general principles of component-based development.

In the next part of this chapter, we will refer to several component models and architecture description languages with support for advanced features of dynamic or mobile architectures. In the case of component models, we will deal mainly with: Darwin by Magee et al. (1995), SOFA by Plasil et al. (1998), SOFA 2.0 by Bures et al. (2006), Koala by van Ommering et al. (2000), ArchJava by Aldrich et al. (2002), and Fractal by Bruneton et al. (2004). For architecture description languages without component models, we will refer to ArchWare ADL by Balasubramaniam et al. (2005).

The component models and ADLs above can be used to describe service-oriented architectures, with or without some limitations. Basically, the models and languages often allow describing the service-oriented architectures as component-based systems (e.g., by means of "utility interface" pattern in SOFA 2.0). However, in the next part of the chapter, we will refer also directly to service-oriented architecture, as the architecture of services respecting *SOA principles* described by Erl (2005). These principles such as loose coupling, statelessness, or reusability, allow easy runtime modifications of a composed service-oriented system by changing its particular components/services as described by Karastoyanova et al. (2005).

Desired Features of Modern Software Architectures

Modern software architectures, such as service-oriented architectures (SOAs), are increasingly dynamic. Software systems utilizing such architectures require the ability to reconfigure the architectures with flexibility and extensibility both at design-time and runtime to be able to cope with fluctuating execution context and constrained resources (Malek et al., 2010). This can be achieved by the advanced features of dynamic architectures extending architectural reconfiguration and representing abilities of architecture evolution.

In this chapter, the four key features of dynamic software architectures will be addressed and evaluated in connection with presented approaches to preventing architectural violations:

- **Ad Hoc Reconfiguration:** The ability of software systems to perform runtime reconfigurations that cannot be completely predefined at the system's design-time and are decided at runtime according to ad hoc needs. Ad hoc reconfigurations are necessary whenever a software system should adapt to its fluctuating execution context and to unexpected modifications of deployment architecture (e.g., moving the system's components from highly distributed to almost centralized architecture and vice versa, for example, due to scalability).
- **Service or Component Mobility:** The ability of software systems to move their state-less as well as state-full components into different architectural contexts at runtime to achieve desired architectural configurations. By means of mobility of service or components, a software system can adapt its logical and deployment architecture to better utilize available resources. For example, a service's latency can be improved if a system is deployed such that the most frequent and voluminous interactions among the components involved in delivering the service occur either locally or over reliable and capacious network links (Malek et al., 2010).
- **Composition Hierarchy Preservation:** The ability of software systems to preserve specific hierarchical compositions of their components during runtime reconfigurations as they were predefined at design-time. For example, in SOA, orchestrating services are logically "composed of" orchestrated services and the component hierarchy preservation in SOA means the preservation of service orchestrations which forms service hierarchy, which may be critical for a coupling of business processes to services.
- **Architectural Aspects:** The ability to describe and control runtime architectural reconfigurations of software systems at design-time by globally-defined concerns that cut across architectural entities, such as individual components as well as their hierarchy, interfaces, and connectors, without links to individual architectural entities (Garcia et al., 2006).

The next section will delve into the details of the various approaches to prevent architectural violations.

APPROACHES TO PREVENTING ARCHITECTURAL VIOLATIONS

The architectural violations of component-based systems with dynamic architectures usually result from unrestrained runtime reconfigurations, as described before. After a series of consequent runtime reconfigurations, an initially well-defined architecture may become unmaintainable and erroneous, and eventually, the reconfigurations may cause architectural drift or architectural erosion. To prevent these architectural violations, different measures can be taken.

Current approaches to dynamic architectures address the problems of architectural violations and their prevention in different ways. Basically, the approaches prevent a system from the architectural violations by means of predefined design-time rules and specific runtime restrictions. Generally, these measures often result in limited reconfiguration possibilities, which may interfere with advanced features of dynamic architectures.

The following sections discuss possible usage of static architecture, static binding, predefined reconfigurations, reconfiguration patterns, restricted reconfiguration controllers, and formalized reconfigurations and invariants, as the measures for prevention of architectural violations.

Static Architecture

The most trivial solution to avoiding architectural violations is to prohibit runtime reconfigurations and to describe the only possible configuration of a component-based or service-oriented system at its design-time. In this case, the system will have a static architecture. Potential variants of the system can be handled at its design-time only, e.g., by means of product line techniques where the component-based or service-oriented system is a member of a product family or a set of product variations (Clements et al., 2001).

Due to the forbidden runtime reconfigurations, architectural violations are not possible in the case of static architecture. This solution is suitable only for software systems deployed into well-understood and strictly defined environments.

Static Binding

In this case, runtime reconfigurations of a system are limited by its inability to reconnect its components or services. The resulting architecture does not need to be static (e.g., dynamic instantiation of components or services is allowed); however, bindings of the components and services are static. With the static bindings, all runtime reconfigurations respect a predefined architectural style, which is defined at the design-time and describes (static) interconnections of components or services into (static) structure of a component-based or service-oriented system, respectively.

Due to the static bindings, the architecture is limited in its runtime evolution and architectural violations are not possible. An example of a component model with static bindings and runtime reconfigurations is SOFA with support for dynamic update of its components, which was introduced by Plasil et al. (1998).

Predefined Reconfigurations

Runtime reconfigurations of a component-based system can be predefined at the system's design-time. In this case, all possible runtime reconfigurations of a system are described in its design specification as a list of permitted configurations of the system's architecture to provide its particular functionalities. Contrary to the previous cases, this architecture is fully dynamic and components or services can be integrated into different contexts. Nevertheless, all future runtime configurations of the architecture have to be considered at design-time, so architectural violations are not possible even in this case.

For example, in service-oriented architecture, a system is composed of individual services that are interconnected at the system's runtime according to their predefined choreography to implement particular business processes. The business processes and the attached service choreographies are described at the system's design-time. Another example is the component model Koala introduced by van Ommering et al. (2000), where runtime reconfigurations are restricted to switching between given components according to the rules predefined at a system's design-time.

Reconfiguration Patterns

Reconfiguration patterns allow controlling the evolution of dynamic architectures by limiting their runtime reconfigurations to be compliant with well-defined patterns. These patterns are usually defined as abstractions for a particular component model where they address specific reconfiguration actions (e.g., a dynamic component instantiation and component removal, referring and dynamic binding of component interfaces, etc.). The permitted reconfiguration actions are defined including prescribed conditions for architecture configurations before and after the actions (i.e., pre- and post-conditions for the process of reconfiguration). Then, in a component-based or service-oriented system, a reconfiguration pattern is applied to a group of components or services, respectively, to define their roles in the system's runtime reconfiguration.

A dynamic architecture described by its initial configuration and with applied reconfiguration patterns can evolve in predefined ways only. However, to prevent architectural violations, it is necessary to provide well-defined reconfiguration patterns fitting the needs of a particular component model.

For example, component model SOFA 2.0 by Bures et al. (2006) defines three reconfiguration patterns for runtime reconfigurations: nested factory (creating a new component and its integration), component removal (vice versa), and utility interface patterns (a component may define utility interfaces that can be freely passed among other components and used later to establish new connections independently of the component's level in the architecture hierarchy).

Another example of a component system with reconfiguration patterns is ArchJava by Aldrich et al. (2002), where possible connections of a new component are restricted by connection patterns defining permitted types of connectible interfaces and connectible components.

Restricted Reconfiguration Controllers

Runtime reconfigurations of a system's architecture are complex processes which themselves can be implemented by specialized composite components or orchestrating services. In such cases, a component-based or service-oriented system contains two types of components or services: the components/services that implement the system's basic functionality and the components/services that control its runtime reconfigurations, i.e., the reconfiguration controllers. Besides the reconfiguration controllers, a component-based or service-oriented system may also contain other types of controllers, e.g., related to a life-cycle of its components or services, respectively. Moreover, each reconfiguration controller, as a component or service of the component-based or service-oriented system, respectively, can be the subject of further runtime reconfigurations realized by other reconfiguration controllers. Eventually, these runtime reconfigurations may result in serious architectural violations.

To avoid these architectural violations, component models can restrict the architecture of the reconfiguration controllers. Typically, the controllers need to be restricted to have strictly static and non-hierarchical architecture, which must be described at a design-time and cannot be a subject of future runtime reconfigurations. Moreover, bindings between interfaces of reconfiguration controllers and interfaces of other components or services have to be limited to events triggering only allowed runtime reconfigurations.

For example, Bures et al. (2006) described SOFA 2.0 reconfiguration controllers which must be realized as so-called micro-components, i.e., primitive components without controller parts. Similarly, Fractal components by Bruneton et al. (2004) contain simple content-controller interfaces to introspect and reconfigure their subcomponents and internal bindings.

Formalized Reconfigurations and Invariants

Another possibility to avoid architectural violations is to define a formal system for description of permitted runtime reconfigurations. Runtime reconfigurations of a component-based or service-oriented system can be described at its design-time as the system's behavior or as a set of invariants of its architecture configurations.

Moreover, the formal description can be used for model checking of a system's behavioral properties during reconfigurations of its architecture and for formal verification of the invariants in resulting configurations at the system's runtime. The model checking and formal verification ensure that an evolving architecture meets its design-time requirements, i.e., they prevent architectural violations.

Several component models use formal architecture description languages with behavioral description of modeled component-based systems. These are namely: component model Darwin with Tracta approach to formally describe behavior of its components by Giannakopoulou et al. (1999), the previously mentioned SOFA with behavior protocols by Plasil and Visnovsky (2002), and Fractal with behavior formally described by means of parameterized networks of communicating automata by Barros (2005).

Other formal approaches are based on grammars and automata. For example, in graph grammars (Hirsch et al., 1998), hyper-edges represent components, nodes linked to the hyper-edges represent communication ports of the components, and rules in the grammars generating graphs describe possible connections of hyper-edges, via nodes representing possible communication patterns of multiple components through their ports.

EVALUATION OF THE APPROACHES

The approaches for prevention of architectural violations, which have been described in the previous section, result in limited reconfiguration possibilities. Therefore, the measures proposed by the approaches may interfere with advanced features of dynamic architecture in current component-based or service-oriented systems.

The following sections evaluate compatibility of the measures with the advanced features of dynamic architecture, such as ad-hoc reconfigurations, component mobility, component hierarchy preservation, and architectural aspects.

Ad Hoc Reconfigurations

The ad hoc reconfigurations represent the ability of a system to perform runtime reconfigurations that cannot be predefined at the system's design-time. Typically, during the design-time of a system which enables evolution of its architecture, a system architect does not have the correct or complete knowledge of all possible runtime reconfigurations of the system's architecture and the reconfigurations can be defined only by their assumed properties.

For obvious reasons, the ad-hoc reconfigurations are not supported by the static architecture, static binding, and predefined reconfigurations approaches. These approaches do not allow runtime reconfigurations at all or permit only those runtime reconfigurations which are known at design-time. Also the *reconfiguration patterns* approach does not enable the ad hoc reconfigurations because each future runtime reconfiguration is an application of a specific pattern described at design-time.

Contrary to the previous approaches, ad hoc reconfigurations are supported by the restricted reconfiguration controllers and the formalized reconfigurations and invariants approaches. Both of these approaches do not refer to particular runtime reconfigurations, but define generally applicable restrictions at design-time.

Ad hoc reconfigurations are not common in service-oriented architecture (SOA). In SOA, service orchestration and choreography, which determine configuration of the resulting architecture, are driven by business processes, in the case of business services, or by a composition hierarchy and technical needs in the cases of controller and utility services, respectively, the types of services that were defined by Erl (2005). However, ad hoc reconfigurations of SOA may be required in the cases of systems adapting automatically to changing business processes or to a varying deployment environment. In these cases, the restricted reconfiguration controllers approach can be utilized with specific services implementing the reconfiguration controllers. The formalized reconfigurations and invariants approach requires the ability to describe a service-oriented architecture and its evolution formally, for example, by application of a component model or an architecture description language with support for SOA, e.g., SOFA 2.0 by Bures et al. (2006).

Component Mobility

Component mobility (or service mobility, as services can be considered as components) enables components to be instantiated and connected, or reconnected in the case of existing components, at runtime into different contexts in a system's architecture. Component mobility can be an essential feature of dynamic architectures with reusability of components, i.e., for service-oriented architecture (SOA). Analogously to the ad hoc reconfigurations, component mobility is also not supported by the static architecture and the static binding approaches, which do not allow runtime reconfigurations.

Component mobility is supported by the predefined reconfigurations approach where components can change their contexts at a system's runtime according to the reconfigurations predefined at its design-time (SOA was also mentioned as one of the examples in the section describing predefined reconfigurations). Moreover, component mobility is supported also by the reconfiguration patterns, the restricted reconfiguration controllers, and the formalized reconfigurations and invariants approaches. In the cases of the reconfiguration patterns and the restricted reconfiguration controllers, a particular realization of the component mobility depends on the design of specific patterns or controller restrictions. These are, for example, "nested factory" and "utility interface" patterns in SOFA 2.0 described by Bures et al. (2006). In the case of the formalized reconfigurations and invariants approach, the component mobility depends on a utilized formalism for description of reconfiguration processes or their invariants, for example, the π-calculus formalism described by Oquendo (2004) in the ArchWare project.

Component Hierarchy Preservation

The component hierarchy preservation is the ability of an approach to preserve, during runtime reconfigurations, a specific hierarchical composition of components predefined at system design-time. The component hierarchy preservation can be an important feature of hierarchical component models to prevent an architectural drift which may cause further architectural violations. Nevertheless, it can also be an insuperable obstacle to advanced features of dynamic architectures such as component mobility.

A component hierarchy is always preserved by the static architecture and the static binding approaches because missing runtime reconfiguration features do not allow for any possible changes in the component hierarchy. In the case of the predefined reconfigurations, the component hierarchy preservation is determined by a system architect who defines the reconfigurations. Analogously, in the cases of the reconfiguration patterns, the restricted reconfiguration controllers, and the formalized reconfigurations and invariants approaches, the component hierarchy preservation can be effectively implemented if needed; however, it may not be necessary, e.g., as in the case of graph grammars by Hirsch et al. (1998). For example, the previously mentioned SOFA 2.0 "nested factory" pattern described by Bures et al. (2006) allows inserting a new component into a predefined context only, thus, with respect to a component hierarchy. On the contrary, component model Fractal by Bruneton et al. (2004) partially breaks a tree-like component hierarchy by introducing shared components as sub-components nested in several components at the same time.

From a structural point of view, service-oriented architecture (SOA) is a *flat model* where "composite" orchestrating services do not enclose their "internal" orchestrated services participating in the orchestrations. The flat model provides better reusability of services, because the context of each service is defined only by its provided and required interfaces, which are the same for all use cases of the service, not by its position in the hierarchy, which may vary in the use cases. Without the flat model, the SOA principles described by Erl (2005) would be violated. However, from a logical point of view, orchestrating services are "composed of" orchestrated services and the component hierarchy preservation means the preservation of service orchestrations which forms the service hierarchy.

Therefore, violations of the component hierarchy preservation are caused by changes in the service orchestrations, e.g., because of adaptation to changing business processes which drive service choreography or because of changes in realization of the services due to an unstable deployment environment with service providers of varying quality.

Architectural Aspects

The architectural aspects have been introduced by Garcia et al. (2006) as a representation of crosscutting concerns at the architectural level, i.e., the concerns that cut across architectural entities such as individual components as well as their hierarchy, interfaces, and connectors. The architectural aspects enable a designer of a system to describe properties of its architecture without links to individual architectural entities. These aspects can be defined globally, at a system's design-time, and for all its architectural entities that meet predefined conditions in its current configuration and future runtime reconfigurations.

A measure preventing architectural violations should be, ideally, describable as an architectural aspect. Then, it is able to persist through unexpected changes in a system's architecture caused by future design-time decisions as well as runtime reconfigurations that cannot be foreseen at the system's design-time.

In the static architecture approach, corresponding measures are defined for the whole architecture and no architectural aspects are needed. The static binding approach has strictly localized measures related to updateable components only, which do not define crosscutting concerns typical for the aspects orientation. Similarly, the predefined reconfigurations in their approach cannot be recognized as the crosscutting concerns. In the case of reconfiguration patterns approach, the patterns are just abstractions applied in component-models and component-based or service-oriented systems locally on specific sets of their components or services, respectively; therefore they cannot be considered as architectural aspects, although they are defined as crosscutting concerns at a general architectural level.

In the restricted reconfiguration controllers and formalized reconfigurations and invariants approaches, specific restrictions of controllers or specific formal descriptions of reconfiguration processes and their invariants can be defined universally for all affected entities of a system's architecture. At the system's architectural level, these restrictions or descriptions represent crosscutting concerns, or the architectural aspects.

Also the principles of service-oriented architecture (SOA), which were defined by Erl (2005), as well as potential global limitations set by Quality of Service (QoS) requirements, can be informally considered as architectural aspects. In the cases of implementations of restricted reconfiguration controllers and formalized reconfigurations and invariants approaches, the SOA principles and QoS requirements have to be respected by these implementations.

Summary

The results of the evaluation are summarized in Table 1. The most compatible approaches for preventing the architecture violations are the *restricted reconfiguration controllers* approach and the *formalized reconfigurations and invariants*

Table 1. Compatibility of the approaches to prevent architectural violations

	Ad Hoc Reconfigurations	Component Mobility	Hierarchy Preservation	Architectural Aspects
Static Architecture	No	No	Yes	N/A
Static Bindings	No	No	Yes	No
Predefined Reconfigurations	No	Yes	Yes/No	No
Reconfiguration Patterns	No	Yes	Yes/No	No
Restricted Reconfiguration Controllers	Yes	Yes	Yes/No	Yes
Formalized Reconfiguration, Invariants	Yes	Yes	Yes/No	Yes

approach. These approaches are compatible with all considered features of dynamic architecture. Nevertheless, both mentioned approaches propose measures that are restrictive or require an advanced knowledge of utilized formalisms (in the case of the formalized reconfigurations). More suitable *predefined reconfigurations* and *reconfiguration patterns* approaches can be recommended for dynamic architectures without ad hoc reconfigurations and usage of architectural aspects.

All advanced features of dynamic architectures discussed above, which are ad hoc reconfigurations, component mobility, component hierarchy preservation, and architectural aspects, can be utilized in service-oriented architecture (SOA). However, according to the presented evaluation, some of the approaches to preventing the architectural violations which are possible in SOA may preclude the usage of some of the discussed features. For example, a typical service-oriented system with services discovered and bounded at runtime from service repositories (e.g., from UDDI registries) implements the predefined reconfigurations approach and it can utilize service mobility (i.e., the component mobility feature) but cannot make ad hoc reconfigurations.

FUTURE RESEARCH DIRECTIONS

New approaches to preventing the architectural violations in dynamically reconfigurable architectures should be compatible with all mentioned advanced features of dynamic architectures, easily integrated into existing modeling tools utilizing existing skills of architects, and supported by well-established implementation technologies and frameworks.

Current research approaches and future research directions mostly address problems of component and service mobility and adaptability of service-oriented architectures with services implementing volatile business processes. Future research is expected to focus on component models and architecture description languages for supporting the latest architectural concepts, such as service-oriented architectures with advanced features and Cloud computing. This research work should be complemented by work in supporting implementation technologies for the architectural concepts, such as implementation frameworks or middleware for component-based systems with mobile and context-aware components.

Very promising research has been ongoing in component and service mobility with research approaches in the above mentioned research areas. For example, multi-agent system approaches JADE described by Bellifemine et al. (2003), Mobile-C by Chen et al. (2006), and AgentScape by Wijngaards et al. (2002), all provide middleware for mobile agents in distributed systems, and the service-oriented approach MobiGo by Song and Ramachandran (2007) provides middleware for seamless mobility of service based on user needs. To prevent the architectural violations and support desired advanced architectural features, these approaches have to implement appropriate techniques for preventing architectural violations as described in this chapter (e.g., agent-based mobility can utilize a formal description of agent-oriented system and implement the formalized reconfigurations and invariants approach).

Future research should also address approaches that are based on planning techniques, e.g., (Barnes et al., 2013). These approaches describe an evolving architecture by its initial and final (planned) configurations, while the best series of reconfiguration leading from the initial configuration to the final configuration, i.e., the best evolution, is to be found with respect to given criteria (by a metric and its preferred value). These approaches are very similar to the previously mentioned agent-based approaches as they do not describe or particularly limit reconfigurations, but operate with evolution goals. Contrary to the agent-based approaches, in the approaches based on planning techniques, the resulting architecture/configuration is known (which is not for the agent-based approaches where the resulting architecture is derived, for example, by beliefs, desires, and intentions of individual agents).

Other promising approaches are related to evolution of business processes and adaptation of underlying workflows and service-oriented architectures implementing process choreographies. Currently, most of the research work in this area addresses just the evolution of business processes and workflows and does not consider the impact of such changes on the underlying implementations.

CONCLUSION

In this chapter, we addressed the problem of architectural violations in the dynamic architectures with advances features such as ad hoc reconfiguration, service or component mobility, composition hierarchy preservation, and architectural aspects. The evaluation of several approaches to preventing the architectural violations was performed to check compatibility of the approaches with the features of dynamic architectures. We focused on service-oriented architectures which are considered to be a special case of component-based system architecture.

All advanced features of dynamic architectures discussed in this chapter can be utilized in a service-oriented architecture. However, the evaluation indicated that some of the approaches for preventing architectural violations which are possible to implement in SOA may preclude the usage of some of the discussed features. The most compatible approaches for preventing the architecture violations are the restricted reconfiguration controllers approach and the formalized reconfigurations and invariants approach supporting all the advanced architectural features. The predefined reconfigurations and reconfiguration patterns approaches can be recommended for dynamic architectures without ad hoc reconfigurations and without usage of architectural aspects.

We intentionally did not discuss performance and scalability (and many other) issues of the presented approaches as these usually depend on particular implementations; however, they also need to be also considered for implementation (e.g., verification of invariants and model checking in the formalized reconfigurations and invariants approach can be expensive in practice).

The results of the evaluation presented in this chapter can be used by architects or developers of component-based and service-oriented software systems with presented features of dynamic architectures, as guidance for further analysis of utilized techniques to control architectural evolution.

REFERENCES

Aldrich, J., Chambers, C., & Notkin, D. (2002). ArchJava: Connecting software architecture to implementation. In *Proceedings of the 24rd International Conference on Software Engineering*, (pp. 187–197). Academic Press.

Balasubramaniam, D., Morrison, R., Oquendo, F., Robertson, I., & Warboys, B. (2005). *Second release of ArchWare ADL* (Technical Report D1.7b [and D1.1b]). ArchWare Project IST-2001-32360.

Barnes, J. M., Pandey, A., & Garlan, D. (2013). Automated Planning for Software Architecture Evolution. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. IEEE/ACM.

Barros, T. (2005). *Formal specification and verification of distributed component systems*. (PhD thesis). Universite de Nice, INRIA Sophia Antipolis.

Bass, L., Clements, P., & Kazman, R. (2003). *Software Architecture in Practice* (2nd ed.). Addison Wesley Professional.

Bellifemine, F., Caire, G., Poggi, A., & Rimassa, G. (2003). JADE: A white paper. *EXP in Search of Innovation, 3*(3), 6–19.

Bruneton, E., Coupaye, T., & Stefani, J.-B. (2004). *The fractal component model, draft of specification version 2.0-3*. The ObjectWeb Consortium.

Bures, T., Hnetynka, P., & Plasil, F. (2006). SOFA 2.0: Balancing advanced features in a hierarchical component model. In *Proceedings of SERA*. Seattle, WA: IEEE Computer Society. doi:10.1109/SERA.2006.62

Chen, B., Cheng, H. H., & Palen, J. (2006). Mobile-C: A mobile agent platform for mobile C-C++ agents. *Software, Practice & Experience*, *36*(15), 1711–1733. doi:10.1002/spe.742

Clements, P., & Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison Wesley.

Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ: Prentice Hall PTR.

Garcia, A., Chavez, C., Batista, T., Santanna, C., Kulesza, U., Rashid, A., & Lucena, C. (2006). On the modular representation of architectural aspects. In *Proceedings of the Third European Conference on Software Architecture*. Berlin: Springer-Verlag. doi:10.1007/11966104_7

Giannakopoulou, D., Kramer, J., & Cheung, S. C. (1999). Behaviour analysis of distributed systems using the Tracta approach. *Automated Software Engineering*, *6*(1), 7–35. doi:10.1023/A:1008645800955

Hirsch, D., Inverardi, P., & Montanari, U. (1998). Graph grammars and constraint solving for software architecture styles. In *Proceedings of the Third International Workshop on Software*. New York, NY: ACM. doi:10.1145/288408.288426

IEEE. (2000). Recommended practice for architectural description of software intensive systems (Technical Report IEEE P1471–2000). The Architecture Working Group of the Software Engineering Committee, Standards Department, IEEE.

Karastoyanova, D., Houspanossian, A., Cilia, M., Leymann, F., & Buchmann, A. (2005). Extending BPEL for runtime adaptability. In *Proceedings of Ninth IEEE International EDOC Enterprise Computing Conference*, (pp. 15–26). IEEE. doi:10.1109/EDOC.2005.14

Lau, K.-K., & Wang, Z. (2005). A taxonomy of software component models. In *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, (pp. 88–95). IEEE Computer Society.

Magee, J., Dulay, N., Eisenbach, S., & Kramer, J. (1995). Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*. London, UK: Springer-Verlag.

Malek, S., Edwards, G., Brun, Y., Tajalli, H., Garcia, J., & Krka, I. et al. (2010). An architecture-driven software mobility framework. *Journal of Systems and Software*, *83*(6), 972–989. doi:10.1016/j.jss.2009.11.003

Oquendo, F. (2004). π-ADL: An architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, *29*(3), 1–14. doi:10.1145/986710.986728

Perry, D. E., & Wolf, A. L. (1992). Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, *17*(4), 40–52. doi:10.1145/141874.141884

Plasil, F., Bilek, D., & Janecek, R. (1998). SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proceedings of 4th International Conference on Configurable Distributed Systems*. Los Alamitos, CA: IEEE Computer Society. doi:10.1109/CDS.1998.675757

Plasil, F., & Visnovsky, S. (2002). Behavior protocols for software components. *IEEE Transactions on Software Engineering*, *28*(11), 1056–1076. doi:10.1109/TSE.2002.1049404

Song, X., & Ramachandran, U. (2007). Mobigo: A middleware for seamless mobility. In *Proceedings of 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, (pp. 249–256). IEEE.

Szyperski, C. (2002). *Component Software: Beyond Object-oriented Programming* (2nd ed.). Addison Wesley Professional.

van Ommering, R., van der Linden, F., Kramer, J., & Magee, J. (2000). The Koala component model for consumer electronics software. *Computer*, *33*(3), 78–85. doi:10.1109/2.825699

Vestal, S. (1993). *A cursory overview and comparison of four architecture description languages (Technical Report)*. Honeywell Technology Center.

Wijngaards, N. J. E., Overeinder, B. J., van Steen, M., & Brazier, F. M. T. (2002). Supporting internet-scale multi-agent systems. *Data & Knowledge Engineering*, *41*(2-3), 229–245. doi:10.1016/S0169-023X(02)00042-3

ADDITIONAL READING

Aguirre, N., & Maibaum, T. (2002). A temporal logic approach to the specification of reconfigurable component-based systems. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, Los Alamitos, CA, pp. 271–274. IEEE Computer Society. doi:10.1109/ASE.2002.1115028

Arbab, F. (1999). Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, *14*(3), 329–366. doi:10.1017/S0960129504004153

Attiogbe, C. (2009). Can component/service-based systems be proved correct? *Lecture Notes in Computer Science*, *5404*, 3–18. doi:10.1007/978-3-540-95891-8_2

Barros, T., Cansado, A., Madelaine, E., & Rivera, M. (2006). Model-checking distributed components: The Vercors platform. In F.S. de Boer & V. Mencl (Eds.), *Preliminary Proceedings of the Third International Workshop on Formal Aspects of Component Software, number 344 in UNU-IIST Report.*

Baude, F., Caromel, D., & Morel, M. (2003). In R. Meersman, Z. Tari, & D. C. Schmidt (Eds.), *From distributed objects to hierarchical grid components* (Vol. 2888, pp. 1226–1242). Lecture Notes in Computer Science Springer-Verlag. doi:10.1007/978-3-540-39964-3_78

Breivold, H. P., Crnkovic, I., & Larsson, M. (2012). A systematic review of software architecture evolution research. *Information and Software Technology*, *54*(1), 16–40. doi:10.1016/j.infsof.2011.06.002

Bruneton, E., Coupaye, T., & Stefani, J.-B. (2002). Recursive and dynamic software composition with sharing. In *Proceedings of the 7th International Workshop on Component-Oriented Programming,* Malaga, Spain.

Bruni, R., Bucchiarone, A., Gnesi, S., Hirsch, D., & Lafuente, A. L. (2008). Graph-Based Design and Analysis of Dynamic Software Architectures. In P. Degano, R. Nicola, & J. Meseguer (Eds.), Lecture Notes in Computer Science: Vol. 5065. *Concurrency, Graphs and Models* (pp. 37–56). doi:10.1007/978-3-540-68679-8_4

Bruni, R., Bucchiarone, A., Gnesi, S., & Melgratti, H. (2008). Modelling Dynamic Software Architectures using Typed Graph Grammars. *Electronic Notes in Theoretical Computer Science*, *213*(1), 39–53. doi:10.1016/j.entcs.2008.04.073

Bruni, R., Corradini, A., Gadducci, F., Lluch Lafuente, A., & Vandin, A. (2012). A conceptual framework for adaptation. *Fundamental Approaches to Software Engineering*. *Lecture Notes in Computer Science*, *7212*, 240–254. doi:10.1007/978-3-642-28872-2_17

Bucchiarone, A., Marconi, A., Pistore, M., & Raik, H. (2012). Dynamic Adaptation of Fragment-Based and Context-Aware Business Processes. In *IEEE 19th International Conference on Web Services (ICWS)*, pp. 33–41. doi:10.1109/ICWS.2012.56

Causevic, A., & Vulgarakis, A. (2009). Towards a unified behavioral model for component-based and service-oriented systems. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference*, pp. 497–503. IEEE Computer Society. doi:10.1109/COMPSAC.2009.182

Chappell, D. (2007). Introducing SCA. *White paper.* Chappell & Associates.

Collet, P., Coupaye, T., Chang, H., Seinturier, L., & Dufrene, G. (2007). *Components and services: A marriage of reason. Technical Report ISRN I3S/RR-2007-17-FR*. Project RAINBOW.

Crnkovic, I., Chaudron, M., & Larsson, S. (2006). Component-based development process and component lifecycle. In *Proceedings of the International Conference on Software Engineering Advances*, Tahiti, French Polynesia. IEEE.

Diakov, N. K., & Arbab, F. (2004). Compositional construction of Web Services using Reo. In S. Bevinakoppa & J. Hu (Eds.), *Proceedings of the International Workshop on Web Services: Modeling, Architecture and Infrastructure* (pp. 49–58). INSTICC Press.

Ehrig, H., Ermel, C., Runge, O., Bucchiarone, A., & Pelliccione, P. (2010). Formal Analysis and Verification of Self-Healing Systems. *Fundamental Approaches to Software Engineering*. *Lecture Notes in Computer Science*, *6016*, 139–153. doi:10.1007/978-3-642-12029-9_10

Ellis, C., Keddara, K., & Rozenberg, G. (1995). Dynamic change within workflow systems. In *Proceedings of conference on Organizational computing systems*, New York, NY, pp. 10–21. ACM. doi:10.1145/224019.224021

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. PhD thesis. University of California, Irvine.

Garlan, D., Monroe, R. T., & Wile, D. (2000). ACME: Architectural description of component-based systems. In G. T. Leavens, & M. Sitaraman (Eds.), *Foundations of Component-Based Systems* (pp. 47–68). New York, NY: Cambridge University Press.

Hatebur, D., Heisel, M., & Souquieres, J. (2006). A method for component-based software and system development. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 72–80. IEEE Computer Society. doi:10.1109/EUROMICRO.2006.9

Hnetynka, P., & Plasil, F. (2006). Lecture Notes in Computer Science: Vol. 4063. *Dynamic reconfiguration and access to services in hierarchical component models* (pp. 352–359). Springer. doi:10.1007/11783565_27

Kazzaz, M. M., & Rychly, M. (2013). A web service migration framework. In *Proceedings of the Eighth International Conference on Internet and Web Applications and Services*, pp. 58–62. IARIA.

Kral, J., & Zemlicka, M. (2000). Lecture Notes in Computer Science: Vol. 1963. *Autonomous components* (pp. 375–383). Springer.

Lane, S., Bucchiarone, A., & Richardson, I. (2012). SOAdapt: A process reference model for developing adaptable service-based applications. *Information and Software Technology*, *54*(3), 299–316. doi:10.1016/j.infsof.2011.10.003

Lanese, I., Bucchiarone, A., & Montesi, F. (2010). A framework for rule-based dynamic adaptation. In *Proceedings of the 5th international conference on Trustworthly global computing)*, pp. 284–300. doi:10.1007/978-3-642-15640-3_19

Lau, K.-K., & Wang, Z. (2006). A survey of software component models (2nd Ed.). Pre-print CSPP-38. The University of Manchester, School of Computer Science.

Lins, F., Damasceno, J., Souza, A., Silva, B., Aragao, D., Medeiros, R., & Rosa, N. (2012). Towards automation of SOA-based business processes. *International Journal of Computer Science, Engineering and Applications, 2*(2).

Medvidovic, N., & Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, *26*(1), 70–93. doi:10.1109/32.825767

Mencl, V., & Bures, T. (2005). Microcomponent-based component controllers: A foundation for component aspects. In *Proceedings of 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, Taipei, Taiwan, pp. 729–737. IEEE Computer Society Press. doi:10.1109/APSEC.2005.78

Open SOA Collaboration. (2007a). SCA service component architecture: Assembly model specification. *Technical Report SCA version 1.00.*

Open SOA Collaboration. (2007b). SCA service component architecture: Web Service binding specification. *Technical Report SCA version 1.00.*

Open, S. O. A. Collaboration. (2008). Service Component Architecture specifications. Retrieved from http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications

Rychly, M. (2009a). (Manuscript submitted for publication). A case study on behavioural modelling of service-oriented architectures. [AGH University Press.]. *Software Engineering Techniques*, 79–92.

Rychly, M. (2009b). A component model with support of mobile architectures and formal description. *e-Informatica*. *Software Engineering Journal*, *3*(1), 9–25.

Visnovsky, S. (2002). *Modeling software components using behavior protocols.* PhD thesis. Department of Software Engineering, Charles University, Prague.

KEY TERMS AND DEFINITIONS

Architectural Drift: Insensitivity to a system's architecture that, with increasing evolution, leads to its in-adaptability and a lack of coherence and clarity of form.

Architectural Erosion: Violations of a system's architecture that leads to significant problems in the system and contributes to its increasing brittleness.

Architectural Violations: Violations of a system's architecture, usually an architectural drift or architectural erosion.

Component or Service Mobility: An ability of a software system to move its components or services into different contexts and to different deployment nodes or service providers at the system's runtime; a specific type of runtime reconfiguration.

Configuration of Architecture: A particular way in which a system's components or services and their connectors or bindings are composed and built into the resulting system.

Dynamic (Software) Architecture: Software architecture of a software system with rules of evolution of its structure/architecture during runtime. The system's components and connections can be created and destroyed during runtime according to the rules from design-time.

Mobile (Software) Architecture: A dynamic architecture of software with component or service mobility features.

Reconfiguration of Architecture: A modification of the configuration of a system's architecture.

Runtime Reconfiguration of Architecture: An ability of a software system to perform reconfiguration of its architecture at runtime, e.g., to create, destroy, and update the services, and to establish and destroy their interconnections dynamically at the runtime, on demand, and according to various aspects to move the services into different contexts and to different providers.

Static (Software) Architecture: A software architecture without ability to be modified during runtime. After initialization of the system, there are no new connections between the system's components and existing connections cannot be destroyed.

ENDNOTES

1 This work was supported by the research program MSM 0021630528 "Security-Oriented Research in Information Technology" and the BUT FIT grant FIT-S-11-2.