Dynamically Reconfigurable Runtime Architectures: Challenges and Service-driven Approaches

Marek Rychly

Brno University of Technology, Czech Republic

ABSTRACT

Dynamic aspects of behavior of software systems with dynamically re-configurable runtime architectures can result into significant architectural violations during their runtime. In such cases, a system's architecture evolves during the runtime according to the actual state of the system's environment and consequent runtime re-configurations 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 a system or a lack of its coherence and clarity of its form. This chapter will describe and compare possible measures to prevent the architectural violations, as they are used in dynamic service and component models. The aim of this chapter is to evaluate applicability of those measures in combination with advanced features of re-configurable runtime architectures such as ad-hoc re-configuration, service or component mobility, composition hierarchy preservation, and architectural aspects.

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 allows decomposing 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, allows runtime re-configurations of the systems' architectures, i.e., 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 (i.e., service mobility); etc. Eventually, a series of re-configurations, i.e., 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 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 re-configuration, service or component mobility, composition hierarchy preservation, and architectural aspects.

Specific objectives include an introduction to the problems of dynamically re-configurable 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 of architectural violations prevention, an evaluation of compatibility of the advanced features of dynamic architectures with the methods of architectural violations prevention, and a discussion of utilization issues of the previously evaluated methods of architectural violations prevention in combination with the advanced features of dynamic architectures.

The chapter is organised as follows. After the introduction, the next section deals with software architecture in general and introduces component-based development and service-oriented architecture with concepts of dynamically re-configurable runtime architectures. We also describe several important state-of-the-art works dealing with component-based development and components models supporting features of dynamic and mobile architectures. In the next section, we discuss existing problems related to the support of dynamic and mobile architectures and causing 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 to the architectural violations prevention. More specifically, we analyse compatibility of the approaches with the advanced features of dynamically re-configurable runtime architectures. Finally, we discuss future research directions such as possibilities of utilisation of the advanced features of dynamically re-configurable runtime architectures including previously described methods of the architectural violations prevention 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 in dependence on changes of 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 of the system’s structure during the system’s 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 of a software system in time (also called a “dynamics”). The system’s components and connections are created and destroyed during the system’s runtime according to the rules from the system’s 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 the system’s design-time and according to functional requirements.

Component-based development and Service-oriented Architecture

Component-based development (CBD), see Szyperski (2002), is a software development methodology, which is strongly oriented to composition and re-usability in a software system’s architecture. In the 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), see 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 or service-oriented systems are composed of components or services, respectively, which are interconnected into configurations and which can be further decomposable. However, while service design in SOA is business oriented, i.e., 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 encapsulate another composite or atomic indecomposable component), and components’ behavior.

Reconfiguration in Dynamic and Mobile Architectures

For both CBD and SOA, a static architecture has only one way how 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 architectures during their runtimes. It means runtime modifications of the configuration, in other words a reconfiguration.

Especially in the case of SOA, loose binding between the services, which represent individual components of a service-oriented system, allows runtime re-configurations of the system’s architecture. This is ability 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 (i.e., service mobility); etc.

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

The problem of evolving architectures was introduced by Perry and Wolf (1992) and is known as the problem of architectural drift and architectural erosion. The architectural drift is defined as insensitivity about a system's architecture that, with increasing evolution, leads to its in-adaptability and a lack of
coherence and clarity of form. The *architectural erosion* is defined as violations of a system's architecture that lead to the 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 by violation of the architecture that become obscured due to the architectural drift. In this chapter, the architectural drift and architectural erosion are referred by a single term as the *architectural violations*.

State of the Art in Dynamic and Mobile Architectures

Component-based systems can be modeled as components 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, configurations, rules for changes according to the dynamic architecture (rules for reconfigurations), etc. *Architecture description languages* (ADLs), see Vestal (1993), are languages for describing software systems’ 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 the component-based development.

In the next part of this chapter, we will refer to several component models and architecture description languages with support of advanced features of dynamic or mobile architectures. In the case of component models, we will deal namely with component models Darwin described by Magee et al. (1995), SOFA by Plasil et al. (1998) and SOFA 2.0 by Bures et al. (2006), Koala by van Ommering et al. (2000), ArchJava by Aldrich et al. (2002), and with component model 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 is 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, allows easy runtime modifications of a composed service-oriented system by changing its particular components/services as described by Karastoyanova et al. (2005).

**APPROACHES TO PREVENT ARCHITECTURAL VIOLATIONS**

The architectural violations of component-based systems with dynamic architecture usually result from unrestrained run-time reconfigurations, as it has been described before. After a series of consequent run-time 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 into limited re-configuration 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 the architectural violations.

Static Architecture

The most trivial solution to avoid architecture violations is to prohibit run-time 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.

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

Static Binding

In this case, run-time 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 run-time reconfigurations respect a predefined architectural style, which is defined at a 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 run-time evolution and the architectural violations are not possible.

An example of a component model with the static bindings and run-time reconfigurations is SOFA with support of dynamic update of its components, which has been introduced by Plasil et al. (1998).

Predefined Reconfigurations

Run-time reconfigurations of a component-based system can be predefined at the system's design-time. In this case, all possible run-time 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 run-time configurations of the architecture have to be considered at a design-time and the architectural violations are not possible even in this case.

For example, in the service-oriented architecture, a system is composed of individual services that are interconnected at the system's run-time 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 can be component model Koala which has been introduced by van Ommering et al. (2000) and where run-time 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 limitation of their run-time 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 components' 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 run-time reconfiguration.

A dynamic architecture described by its initial configuration and with applied reconfiguration patterns can evolve in predefined ways only. However, to prevent architecture 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 run-time 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 architecture hierarchy).

Another example of component system with reconfiguration patterns can be 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

Run-time 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 run-time reconfigurations, i.e., the reconfiguration controllers. Moreover, each reconfiguration controller, as a component or service of the component-based or service-oriented system, respectively, can be the subject of further run-time reconfigurations realized by other reconfiguration controllers. Eventually, these run-time reconfigurations may result into serious architectural violations.

To avoid these architectural violations, component models can restrict architecture of the reconfiguration controllers. Typically, the controllers have strictly static and non-hierarchical architecture, which must be described at a design-time and cannot be a subject of future run-time reconfigurations. Moreover, bindings between interfaces of reconfiguration controllers and interfaces of other components or services are often limited to events triggering possible run-time 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) can contain simple content-controller interfaces to introspect and reconfigure their subcomponents and internal bindings.
**Formalized Reconfigurations and Invariants**

Another possibility to avoid architecture violations is to define a formal system for description of permitted run-time reconfigurations. Run-time 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 run-time. 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 by Giannakopoulou et al. (1999) to formally describe behavior of its components, 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).

**EVALUATION OF THE APPROACHES FOR PREVENTION OF ARCHITECTURAL VIOLATIONS**

The approaches for prevention of architectural violations, which has been described in the previous section, result into 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 run-time 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 correct or complete knowledge of all possible run-time 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* and *static binding* approaches and by the *predefined reconfigurations* approach. These approaches do not allow run-time reconfigurations at all or permit only those run-time reconfigurations which are known at a design-time. Also the *reconfiguration patterns* approach does not enable the ad-hoc reconfigurations because each future run-time reconfiguration is application of a specific pattern described at a design-time.

Contrary to the previous approaches, the 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 run-time reconfigurations but, at a design-time, define generally applicable restrictions.
The 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 services and utility services, respectively, as these types of services 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 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 of SOA, e.g., SOFA 2.0 by Bures et al. (2006).

Component Mobility

The 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 a run-time into different contexts in a system's architecture. The 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, also the component mobility is not supported by the static architecture and the static binding approaches, which do not allow run-time reconfigurations.

The component mobility is supported by the predefined reconfigurations approach where components can change their contexts at a system's run-time 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, the component mobility is supported also by the reconfiguration patterns, the restricted reconfiguration controllers, and by 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 a design of specific patterns or controllers' 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 run-time reconfigurations, a specific hierarchical composition of components predefined at a system's 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 be also an insuperable obstacle to advanced features of dynamic architectures such as the previously mentioned component mobility.

A component hierarchy is always preserved by the static architecture and the static binding approaches because missing run-time reconfiguration feature does not allow 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 necessary). 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 services 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 run-time 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 run-time 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 got 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 the 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, 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 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.

<table>
<thead>
<tr>
<th></th>
<th>ad-hoc reconfigurations</th>
<th>component mobility</th>
<th>hierarchy preservation</th>
<th>architectural aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>static architecture</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>static bindings</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>predefined reconfigurations</td>
<td>No</td>
<td>Yes</td>
<td>Yes/No</td>
<td>No</td>
</tr>
<tr>
<td>reconfiguration patterns</td>
<td>No</td>
<td>Yes</td>
<td>Yes/No</td>
<td>No</td>
</tr>
<tr>
<td>restricted reconfiguration controllers</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes/No</td>
<td>Yes</td>
</tr>
<tr>
<td>formalized reconfiguration, invariants</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes/No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1. Compatibility of the approaches to the architectural violations prevention (in rows) with features of dynamic architectures (in columns).

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 approach. These approaches are compatible with all considered features of dynamic architecture. Nevertheless, both mentioned approaches propose measures that are restrictive (in the case of the restricted reconfiguration controllers as well as the invariants of run-time reconfigurations) 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 the architectural violations prevention which are possible in SOA may prevent 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 the architectural violations prevention should be compatible with all mentioned advanced features of dynamic architectures, easily integrated into existing modeling tools utilizing existing skills of architecture designers, and supported by well-established implementation technologies and frameworks.

There exist two possible directions of the future research which should be done in parallel and which results affect each other. The first direction is aimed at component models and architecture description languages supporting the latest architectural concepts, such as already mentioned service-oriented architectures with advance features or cloud computing. The second direction is aimed at supporting implementation technologies for the architectural concepts, such as implementation frameworks or middleware for component-based systems with mobile and context-aware components.

A very promising research is going in component and service mobility with research approaches in both above mentioned research directions. For example, multi-agent system approaches JADE described by Bellifemine et al. (2003), Mobile-C by Chen et al. (2006), or AgentScape by Wijngaards et al. (2002), provide middleware for mobile agents in distributed systems, or 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 one of the architectural violations prevention approaches 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).

Other promising approaches are related to evolution of business processes and adaptation of underlying workflows and service-oriented architectures implementing process choreographies. Currently, the most of research approaches in this direction address just the evolution of business processes and workflows and do not consider the impact of such changes on the underlying implementations. For example, Ellis et al. (1995) addressed an update of business processes to their new versions during their runtime, which may vary significantly depending on how the processes are implemented (e.g., as an automated service-oriented architecture or as a workflow of human tasks).

**CONCLUSION**

In this chapter, we addressed the problem of architectural violations in the dynamic architectures with advances features such as ad-hoc re-configuration, service or component mobility, composition hierarchy preservation, and architectural aspects. The evaluation of several approaches to the architectural
violations prevention has been performed to check compatibility of the approaches with the features of dynamic architectures. We focused on service-oriented architectures which were considered to be a special case of architecture of component-based systems (i.e., services are components) and discussed during the evaluation.

All advanced features of dynamic architectures discussed in this chapter can be utilized in service-oriented architecture. However, the evaluation indicated that some of the approaches for architectural violations prevention which are possible to implement in SOA may prevent 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.

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.

Acknowledgement

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.

REFERENCES


**ADDITIONAL READING SECTION**


KEY TERMS & DEFINITIONS

Configuration of Architecture: a particular way how a system’s components or services and their connectors or bindings are composed and built into the resulting system.

Re-configuration of Architecture: a modification of the configuration of a system’s architecture.

Runtime Re-configuration of Architecture: an ability of a software system to perform re-configuration of its architecture at the system’s run-time, e.g., to create, destroy, and update the services and to establish and destroy their interconnections dynamically at the run-time, on demand, and according to various aspects; to move the services into different contexts and to different providers; etc.

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 run-time; a specific type of run-time reconfiguration.

Static (Software) Architecture: a software architecture without ability to be modified during a system’s runtime; after initialization of the system, there are no new connections between the system’s components and existing connections can not be destroyed.

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

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

Architectural Drift: insensitivity about 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 lead to the significant problems in the system and contribute to its increasing brittleness.

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

Besides the reconfiguration controllers, a component-based or service-oriented system may contain also other types of controllers, e.g., related to a life-cycle of its components or services, respectively.