Rule Type Based Reasoning On Architecture Violations: A Case Study

Leo Pruijt, Wiebe Wiersema
Information Systems Architecture Research Group
HU University of Applied Sciences
Utrecht, The Netherlands
{leo.pruijt, wiebe.wiersema}@hu.nl

Jan Martijn E. M. van der Werf, Sjaak Brinkkemper
Department of Information and Computing Sciences
University Utrecht
Utrecht, The Netherlands
{ j.m.e.m.vanderwerf, s.brinkkemper}@uu.nl

Abstract—**Software architecture compliance checking (SACC) is an approach to monitor the consistency between the intended and the implemented architecture. In case of static SACC, the focus is mainly on the detection of dependencies that violate architectural relation rules. Interpretation of reported violations may be cumbersome, since the violations need to be connected to architectural resolutions and targeted qualities such as maintainability and portability. This paper describes an SACC case study which shows that inclusion of different types of rules in the SACC process enhances reasoning on architecture violations, especially if a rule type is related to specific architectural pattern. The SACC is performed with HUSACCT, an SACC-tool that provides rich sets of module and rule types in support of patterns such as layers, facade, and gateway. The case system is a governmental system developed in C#, which follows the .NET common application architecture. Even though the system appeared to be well-structured, the SACC revealed that 10 of the 17 architectural rules were violated.**

Keywords—Software architecture, modular architecture, architecture compliance, architecture conformance

I. Introduction

The partitioning of an application into smaller units is an important strategy to reduce complexity. Modularization contributes to the manageability of the development process, and it may contribute to quality attributes like maintainability, reusability and portability [1]. A *modular architecture* describes the modular elements, their form (properties and relationships) and the rationale [2]. A modular element, or module, is an implementation unit of software with a coherent set of responsibilities [3]. Properties and relationships express architectural rules. Properties are used to define constraints on the modular element and its content. Relationships are used to constrain how the different elements may interact or otherwise may be related [2].

Ducasse and Pollet [4] make a distinction between the conceptual, or intended, architecture that exists in human minds or in the software documentation, and concrete, or implemented, architecture that refers to the architecture derived from source code. Architectural erosion is "the phenomenon that occurs when the implemented architecture of a software system diverges from its intended architecture" [6]. Software Architecture Compliance Checking (SACC) is a means to prevent or detect architectural erosion by comparison of the intended architecture and the implemented architecture. In our opinion, SACC may not only be used to enforce compliance to the intended architecture, but also to maintain and improve the intended architecture. Furthermore, SACC may increase the architectural awareness and add to a better understanding of the relationship between architecture design and code.

In [5], we introduced the term *semantically rich modular architecture* (SRMA) that we use for an expressive modular architecture description, composed of semantically different types of modules (e.g., layers, subsystems, components), which are constrained by different types of rules such as basic dependency constraints (e.g., Is not allowed to use), constraints related to layers, naming constraints. In practice and literature, many architectures can be labeled as SRMA, since they contain modules with different semantics. Since our research revealed that SRMA support was limited in the set of studied SACC-tools [5], we focused our work on this aspect. As results, we have presented a meta model [6] and a tool, HUSACCT [7], aimed on the provision of extensive and configurable SRMA support. HUSACCT is applied in practice and also in education at several universities in the Netherlands.

The objective of this case study is to explore, in the context of an SACC of a professional system, whether the provided SRMA support is suitable during the registration of the intended architecture and the interpretation of detected violations. Furthermore, we investigate the existence of architecture erosion in this case. In line with these objectives, we formulated the following research questions

1) Is SRMA support suitable within the context of the case?
2) Is SRMA support useful in the SACC process?
3) Is architecture erosion identifiable in this case?

The contribution of this paper is threefold. First, we present the intended SRMA of a professional system and we report on violations against rules. Second, we describe the interpretation of the violations and we demonstrate how rule types may aid architecture reasoning. Third, we answer the research questions defined above.

This paper is outlined as follows. Section II introduces HUSACCT and describes the procedure followed during the case study. Section III introduces the case system and describes its intended architecture. Section IV describes the compliance check and presents, illustrates and interprets its results. Section V discusses the key findings of the case study, the limitations, and the related work. Section VI concludes this paper and addresses future work.

II. SACC Approach

A. Introduction to HUSACCT

HUSACCT (HU Software Architecture Compliance Checking Tool) is a tool that provides support to analyze implemented architectures, define intended architectures, and execute conformance checks [7]. Browsers, diagrams and reports are available to study the decomposition style, uses style, generalization style and layered style [3] of intended architectures and implemented architectures. HUSACCT is free-to-use and open source. It has been developed in Java and analyzes Java and C# source code. The executable and source code are downloadable at http://husacct.github.io/HUSACCT/. User documentation and a video are accessible at the same site.

HUSACCT distinguishes itself from other tools by the provision of extensive and configurable support of semantically rich modular architectures (SRMAs). To enable the provision of SRMA support, we have developed and published the SRMACC metamodel [6], whereof the central part is included in Fig. 1. It includes concepts and their associations relevant to understand our approach. As shown in the figure, an SRMA contains *Modules* of different *ModuleTypes*, where *AppliedRules*, each of a certain *RuleType*, may constrain the *Modules*. For a detailed discussion of the complete metamodel, we refer to [6].

Basic SRMA support includes the provision of rich sets of module and rule types and the functionality to check rules of these types. In a previous publication [5], we identified common module and rule types and discussed their grounding in literature. During the development of HUSACCT, we aimed at support of these common types. Currently HUSACCT provides support for five common ModuleTypes and eleven common RuleTypes.

Extensive semantic support of the module types and rule types is provided in several ways during the definition of the

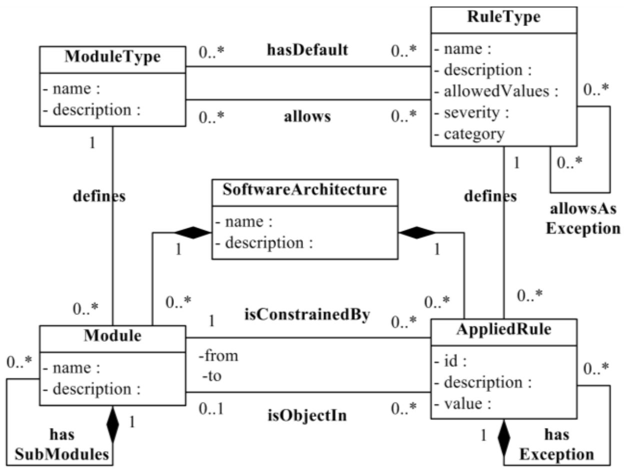


Fig. 1. Part of SRMACC metamodel

intended architecture. For example, in the following situations: a) when a rule is created, only rule types are selectable that are allowed for the type of the constrained module; b) when a module is created, zero, one, or more applied rules will be created, based on the default rule types associated to the module type of the module; c) when an exception rule is created, only rule types are selectable that are allowed as exception the type of the main rule.

Finally, configurable support means that all rules are accessible and that the following configuration options may be applied: 1) generated default rules may be disabled (just as user defined rules); 2) exceptions to generated default rules may be specified (just as exceptions to user defined rules); 3) tool-users may configure the default rule types per module type.

B. Procedure and Data Collection

First, prior to the actual compliance check, we have requested and received a description of the intended modular architecture, including the modules, the rules and the mapping of modules to implemented software units.

Second, we had a preparing interview with the lead developer, who acted as the software architect as well. We were informed on the functionality of the system and on the development organization. Furthermore, we discovered some architectural rules missing in the documentation: two rules that constrain the access of external systems, and one rule that constrains the names of classes in a package.

Third, we installed HUSACCT on one of the organization's laptops, and we analyzed the source code (which took four seconds). Next, we registered the intended architecture into the tool. Finally, we activated the compliance check (which finished within one second), and we studied its results. We performed the two last steps iteratively, to fine-tune the intended architecture and the mapping to the software units.

Fourth, we analyzed older versions of the system's source code, and we performed compliance checks on these versions.

Fifth, we gathered all the data files, needed to perform a detailed analysis afterwards:

- workspace file that contains the intended architecture;
- intended architecture report;
- analysis reports, containing a table with dependencies and a table with an abstracted overview of the dependencies per dependency type and subtype;
- violation reports with an overview of violated and not-violated rules, and with a table with reported violations;
- export files that contained a selection of the analyzed data repository, namely all packages, classes, external systems, and all dependencies.

Sixth, we analyzed the data and we have drawn up a report for the client organization with our findings. The report contained i.e. tables and graphics with the intended architecture, violations per source code version, and for most rules a description of the number and location of the classes that caused the violations.

Seventh, we interviewed the system architect on the validity and the interpretation of our findings.

III. The Case and its Intended Architecture

A. *The Case*

The assessed system is an E-commerce system of a governmental organization in the Netherlands, which is used by citizens and organizations, for example to register or view data, or to apply for a license. The system is developed in C# and follows the .NET common application architecture (MSDN 2009). The E-commerce system may be regarded as one of a multitude of small to medium sized administrative systems with different functionalities but similarities in the modular architecture. The system is composed of: 1) multiple web-based client applications for a variety of products and services; 2) one server-side ServiceComponent, the central component of the application that handles and coordinates service request from web client applications; and 3) multiple server-side plug-ins, which handle the specifics in processing of the different products and services.

The architecture compliance check focuses on the central ServiceComponent, which acts an application specific shell on top of Commerce Server, Microsoft's E-commerce system. The following keywords provide an impression of the responsibilities of ServiceComponent: product catalog management, customer profile management, basket and payment management, order management.

B. *The Intended Architecture*

An overview of the intended architecture of the ServiceComponent is shown in Fig. 2 and Fig. 3. The architecture has been established four years before the SACC. Since then it has remained stable, while the number of products

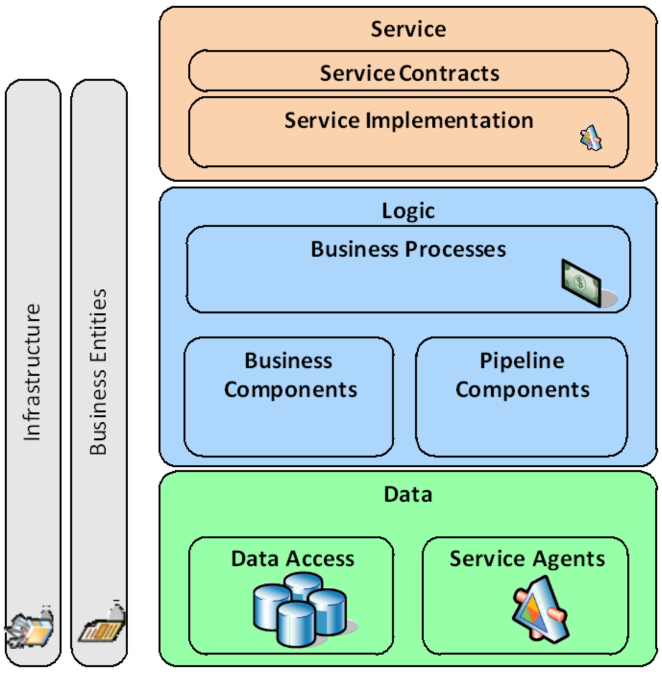


Fig. 2. Overview modular architecture ServiceComponent

and services, provided to customers of the organization via the E-commerce system, has grown from fifteen to sixty.

The intended architecture of ServiceComponent can be labeled as a Semantically Rich Modular Architecture (SRMA), since it contains modules of five different types and rules of eight different types. The first figure provides a high-level overview. Three layers are distinguished, which have the

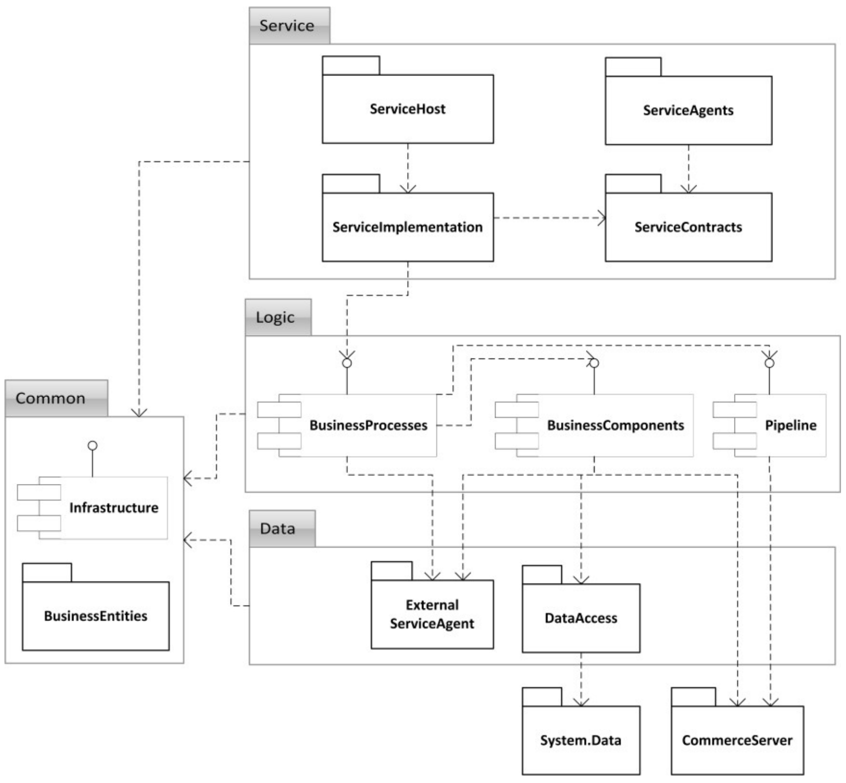


Fig. 3. UML component model ServiceComponent

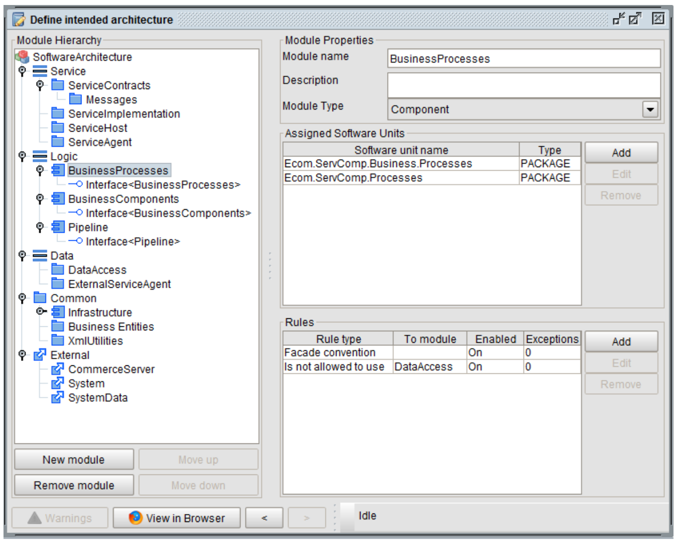
following responsibilities: 1) the Service layer provides the service interface to the web applications; 2) the Logic layer contains the components responsible for the business logic of the application; and 3) the Data layer is responsible for access of the database and communication with infrastructural services. Furthermore, two commonly used modules are visible: Infrastructure, which contains utilities and other shared functionality, and Business Entities, which contains data transfer objects. The rules of a strict layered style apply here: layers are not allowed to make use of higher level layers, and layers are not allowed to skip a layer in their usage relations. Consequently, the Service layer and Logic Layer are not allowed to use infrastructural libraries that are abstracted by the Data Layer.

More rules may be derived from Fig. 3, which provides an overview of the modules and their intended usage relations in the form of an UML component diagram. Identification of the rules based on the component model in the architecture document required interpretation, since an UML component model contains uses dependencies, while constraints need to be derived from the model as rules. The figure presented here is an updated version of the originally received model and a set of specified rules. Based on the first conformance checks it seemed that some uses dependencies were missing in the original component model, which was confirmed by the architect. Conversely, several rules were added, mainly based on additional information obtained in an interview of the system's architect.

The most relevant modules and rules are discussed below. A full specification of the modules, the assigned software units and the checked rules is provided in the next section.

- The Service layer is composed of four submodules, of which only ServiceImplementation is allowed to use the Logic layer, and more specific, only BusinessProcesses. Furthermore, each submodule of Service is allowed to use only one specified other module within Service.
- The Logic layer is composed of three encapsulated modules, BusinessProcesses, BusinessComponents, and Pipelines which may be used only via their interfaces. Furthermore, it is visible that only BusinessComponents and Pipelines are allowed to use Microsoft's CommerceServer.
- The Data layer is composed of two modules, which may be used by a few modules only: DataAccess only by BusinessComponents, and Serviceagent only by BusinessProcesses and BusinessComponents. DataAccess is the only module allowed to use library System.Data.

Nearly all modules in the three layers are allowed to use Common, but module Common.Infrastructure may only be used via its interface. For reasons of clarity, the graphical model is simplified at this point, since (not visible in Fig. 2) ServiceContracts, ServiceHost, Pipeline, and DataAccess are exceptions, which are not allowed to use module Common.



Define intended architecture

Module Hierarchy
```
SoftwareArchitecture
  Service
    ServiceContracts
      Messages
    ServiceImplementation
    ServiceHost
    ServiceAgent
  Logic
    BusinessProcesses
      Interface<BusinessProcesses>
    BusinessComponents
      Interface<BusinessComponents>
    Pipeline
      Interface<Pipeline>
  Data
    DataAccess
    ExternalServiceAgent
  Common
    Infrastructure
    Business Entities
    XmlUtilities
  External
    CommerceServer
    System
    SystemData
```

Module Properties
Module name: BusinessProcesses
Description:
Module Type: Component

Assigned Software Units

Software unit name	Type
Ecom.ServComp.Business.Processes	PACKAGE
Ecom.ServComp.Processes	PACKAGE

Rules

Rule type	To module	Enabled	Exceptions
Facade convention		On	0
Is not allowed to use	DataAccess	On	0

Fig. 4. Intended Architecture as defined in HUSACCT

IV. ARCHITECTURE COMPLIANCE CHECK

A. *Intended architecture in HUSACCT*

Defining an intended architecture starts with the specification of the modules in the view Define intended architecture, visible in Fig. 4. This view shows the modules in the module hierarchy of the intended architecture of ServiceComponent. When a module is added, a module type may be selected. As visible in Fig. 4, all five supported module types are present in the intended architecture: components, interfaces, layers, subsystems, and external systems.

Rule definition is enabled from the same view as well. Fig. 4 shows that two rules are defined for the selected module BusinessProcesses. One rule is of type Facade convention, which forbids usage of the component other than via its interface(s). This rule is automatically generated, when a module of type Component is created. The other rule is of type "Is not allowed to use", and it restricts the usage of module Data.DataAccess.

A table with all rules, including their exceptions is provided below. Table I shows that 17 rules of eight different types of rules are included in the intended architecture. The table is generated as part of the intended architecture report. This report also contains a table with all modules, their type, and the assigned software units per module, but for of reasons of space this table is not included here.

Finally, assignment of implemented software units to the intended modules is supported in this view too. Fig. 4 shows that module BusinessProcesses has two assigned software units in the implemented architecture. Software units can be assigned easily, after source code analysis, by selection of units provided in an overview. If the required knowledge is available, the assignment of software units to modules in HUSACCT can be completed in ten minutes, in a case like this one.

TABLE I. ALL RULES AND EXCEPTIONS

Id	Exception	From module	Rule type	To module	Expression
1		Common.Infrastructure	Facade convention		
2		Common	Is only allowed to use	External	
3		Data.DataAccess	Is the only module allowed to use	External.SystemData	
4		Data	Is not allowed to back call		
5		Logic.BusinessComponents	Facade convention		
6		Logic.BusinessComponents	Is not allowed to use	Logic.BusinessProcesses	
7		Logic.BusinessProcesses	Is not allowed to use	Data.DataAccess	
8		Logic.BusinessProcesses	Facade convention		
9		Logic.Pipeline	Is not allowed to use	Data	
10		Logic.Pipeline	Is the only module allowed to use	External.CommerceServer	
	Exception	Logic.BusinessComponents	Is allowed to use	External.CommerceServer	
11		Logic.Pipeline	Facade convention		
12		Logic	Is not allowed to back call		
13		Service.ServiceAgent	Is only allowed to use	Service.ServiceContracts	
	Exception	Service.ServiceAgent	Is allowed to use	Common	
	Exception	Service.ServiceAgent	Is allowed to use	External.System	
14		Service.ServiceContracts.Messages	Naming convention		*Response
	Exception	Service.ServiceContracts.Messages	Naming convention exception		*Request
15		Service.ServiceContracts	Is only allowed to use	External	
16		Service.ServiceImplementation	Is only allowed to use	Service.ServiceContracts	
	Exception	Service.ServiceImplementation	Is allowed to use	Common	
	Exception	Service.ServiceImplementation	Is allowed to use	Logic.BusinessProcesses	
	Exception	Service.ServiceImplementation	Is allowed to use	External.System	
17		Service	Is not allowed to skip call		

B. *Architecture Violations*

Activation of the compliance check starts up a process that iterates through the rules, and for each rule it checks if there is a class or dependency that violates the rule. Each type of rule has its own checking algorithm that also takes the exceptions to a rule into account.

An overview of the results of the SACC of the ServiceComponent is presented in Table II. The table shows that ten of the seventeen rules are violated, with a total of 654 violations of the rules. Since all violated rules are constraining uses-relations, each violation represents a forbidden dependency of a class on another class.

V. REASONING ON THE ARCHITECTURE AND THE VIOLATIONS

A. *Reasoning on the Modular Architecture*

In general, the ServiceComponent appeared to be well-structured. We noted the following arguments:

- Layers are distinguished.
- Modules within the layers represent different types of functionality, which are in most cases quite well represented by the namespace names.
- The mapping of the intended architecture on the implemented architecture (the implementation units in the source) is in most cases straightforward, since the intended modules map to one or two complete namespaces only.
- In favor of the encapsulation of four modules, interfaces are provided.
- Data.DataAccess, and Data.ServiceAgent are implemented as gateways to reduce dependencies on infrastructural libraries.
- The intended architecture has been stable through the years, although the number of provided user services, which are processed by the system, has grown the last

TABLE II. ALL VIOLATED RULES WITH THE NUMBERS OF REPORTED VIOLATIONS

Id	Logical module from	Rule type	Logical module to	Violations
1	Common.Infrastructure	Facade convention	Common.Infrastructure	496
2	Data.DataAccess	Is the only module allowed to use	External.SystemData	7
3	Data	Is not allowed to back call	Data	12
4	Logic.BusinessComponents	Facade convention	Logic.BusinessComponents	5
5	Logic.BusinessProcesses	Is not allowed to use	Data.DataAccess	3
6	Logic.BusinessProcesses	Facade convention	Logic.BusinessProcesses	6
7	Logic.Pipeline	Is the only module allowed to use	External.CommerceServer	3
8	Logic.Pipeline	Facade convention	Logic.Pipeline	81
9	Service.ServiceAgent	Is only allowed to use	Service.ServiceContracts	2
10	Service.ServiceImplementation	Is only allowed to use	Service.ServiceContracts	39
			Total:	**654**

three years from 15 to 60 different services.

B. *Facilities to Support Reasoning on Violations*

To support reasoning on the impact of the violations, several options are provided, from high-level overviews to a code viewer to study the code construct that causes a violation. In our experience, an overview of the violated rules, their types, and the number of violations per rule, is a good starting point. Table II provides such an overview. This table is part of the Validate conformance view in the user interface and it is also included in a generated document, the violation report.

The Validate conformance view can be used to study the causes and, if needed, to dig to the source of a violation. If a violated rule is selected in this view, the underlying violations with their details are listed. A double click on a violation activates the code viewer, which shows the source code of the related from class and highlights the line which holds the violating code construct.

In addition, the reported violations may be shown in intended architecture diagrams, which show only modules included in the defined intended architecture with their dependencies, and in implemented architecture diagrams, which show all packages and classes in the source. Intended architecture diagrams are often easier to comprehend. They show the logical type of each module, and they include fewer elements, since one module in the intended architecture may represent several software units in the code. Fig. 5 shows an intended architecture diagram with the top-level modules and the dependency relations between the assigned software units in the implementation. A black, dashed arrow in the diagram represents dependencies only. The related number indicates the number of dependencies. A red, dotted arrow represents violations and dependencies. The first related number indicates the number of violations, while the second indicates the total number of dependencies. Fig. 6 shows an intended architecture diagram of the top level modules with their children. Only the violating dependencies between the child modules are included, since inclusion of all other dependency arrows result in an unreadable diagram.

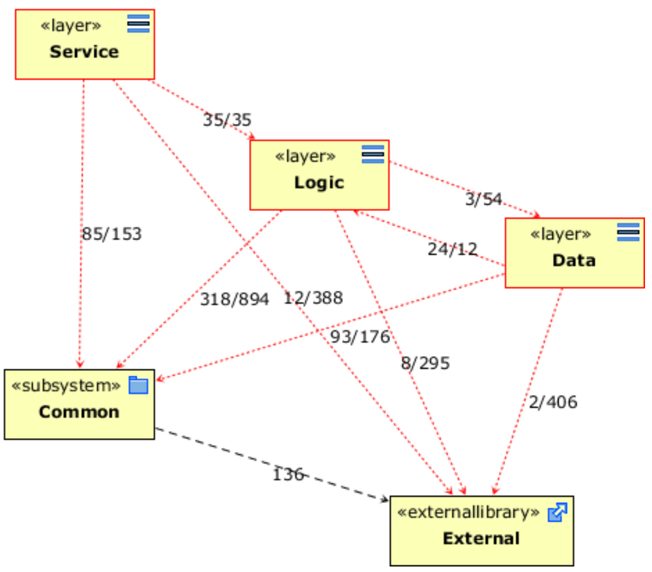
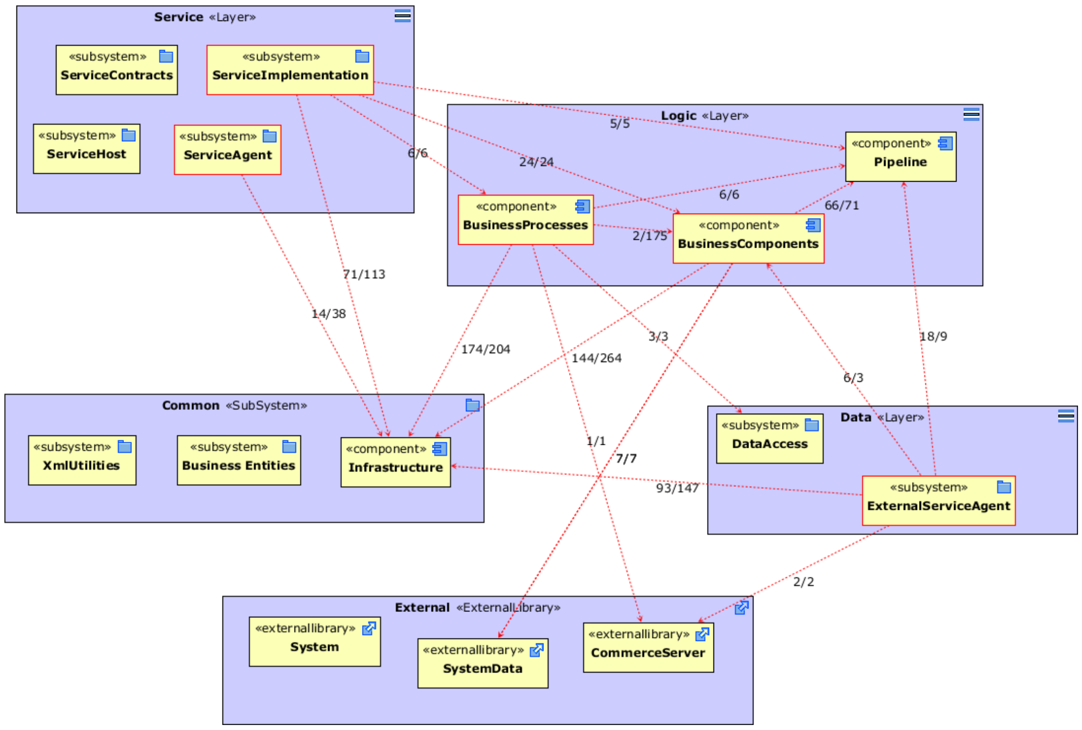


Fig. 5. Intended architecture diagram, top level modules

The diagrams provide an overview of the modules and their types, the origin of the violations and the numbers of violations. However, they do not show the cause(s) of a violation, since violations of different rules, from different types may be represented by a same red, dotted arrow. For



Service «Layer»
«subsystem» ServiceContracts
«subsystem» ServiceImplementation
«subsystem» ServiceHost
«subsystem» ServiceAgent
Logic «Layer»
«component» Pipeline
«component» BusinessProcesses
«component» BusinessComponents
Common «SubSystem»
«subsystem» XmlUtilities
«subsystem» Business Entities
«component» Infrastructure
Data «Layer»
«subsystem» DataAccess
«subsystem» ExternalServiceAgent
External «ExternalLibrary»
«externallibrary» System
«externallibrary» SystemData
«externallibrary» CommerceServer
5/5
6/6
24/24
6/6
66/71
2/175
71/113
14/38
174/204
3/3
144/264
18/9
6/3
1/1
7/7
93/147
2/2

Fig. 6. Intended architecture diagram, top level modules with child modules; violations only

example, the arrow in Fig. 5 from Service to Logic might represent violations to six different rules of two different rule types. To browse relevant information, an arrow may be selected in the diagram to activate a pop-up that lists the represented violations (or dependencies) with their properties.

C. *Analysis of the Reported Violations*

Ten out of 17 architectural rules were reported to be violated in the implemented architecture. We analyzed the violations per rule to get an impression of the number of modules and number of classes that caused the violations. Below, we list some of our findings, sorted per type of rule.

- Facade conventions are related to the facade pattern [8]. Violations to this type of rule form the largest group of violations (rule 1, 4, 6, and 8 in Table II). This type of violations compromises the encapsulation of components by direct usage of internal classes, thus bypassing the component's interface. In this case, the encapsulation is compromised of all three components in the Logic layer and of component Common.Infrastructure.
  - In case of component PipeLine, the interface is bypassed in 81 occasions by seven classes in three different modules (BusinessProcesses, BusinessComponents, and ExternalServiceAgent).
  - The interface of component BusinessComponents is bypassed in 5 occasions by two classes in module BusinessProcesses and one in ExternalServiceAgent.
  - The interface of component BusinessProcesses is bypassed in 6 occasions by one class in module ExternalServiceAgent.
  - The interface of component Infrastructure is bypassed in 496 occasions. A high number, so we studied the causes in more depth. Twelve of the 31 classes of Infrastructure are used in the violations. Five of the twelve classes appeared to be shielded by an interface, but seven were not. These seven were subject in 322 violations. Of these, 139 violations were usages of three exception classes; each one specific for a component in the logic and data layer.
- Back call rules are related to the layer pattern [9]. Violations are limited to rule 3 only. The twelve back calls from layer Data to layer Logic are caused by one class in Data.ExternalServiceAgents, which is using two different classes of module Logic.Pipeline and two different classes of module Logic.BusinessComponents.
- Is the only module allowed to use rules indicate the application of the gateway pattern [10], a refinement of the adapter pattern [8]. Two rules (2 and 7 in Table II) are of this type and restrain usage of System.Data and CommerceServer respectively. Rule 2 is violated by seven different classes in Logic.BusinessComponents. Rule 7 is violated by one class in Logic.BusinessProcesses and one class in Data.ServiceAgent. The reported number of violations is lower than in reality, since, in case of external systems in combination of C#, only dependencies caused by using statements are reported.
- Rules of the types Is only allowed to use, and Is not allowed to use, restrict the responsibility of a module [3]. The violations to rule 5, 9, and 10) indicate that a module has more implemented responsibilities than designed responsibilities, with the risk of duplications and reduced maintainability. The numbers of violating classes is respectively 1, 1, and 5. Especially module Service.ServiceImplementation requires attention, since it exceeds its designed responsibilities substantially.

D. *Interpretation of the Violations*

We discussed the results of the SACC with the architect of the system, based on the SACC report. We were interested in his opinion on the validity and interpretation of our findings.

To start with the validity: the architect approved our findings. Moreover, he asked if it is possible to include HUSACCT in the build procedure of the software development process, in order to prevent violations as reported. Since HUSACCT can perform an SACC in batch mode too, the answer was positive. Furthermore, the architect expressed the intention to solve the reported violations when the ServiceComponent would be extended or adjusted.

With respect to the interpretation of the findings, we focused on the severity of the reported violations and on possible adjustments of the intended architecture. The most interesting opinions of the architect are summarized below.

1) *Severity of the Violations*

The violations of the two rules of type Is the only module allowed to use (rule 1 and 8 in Table II) are severe. For instance, effort has been devoted to enable replacement of CommerceServer, so the usages by BusinessProcesses and ServiceAgents are undermining this intention. In general, skipping a gateway to an external system is severe; more serious than bypassing an interface of an internal component.

The back call from Data to Logic (rule 2) is unexpected. Violations of the layered model undermine the core of the architecture and might have serious consequences. However, the number of violations is small and in this case possibly relatively simple to analyze and repair.

The violations caused by module ServiceImplementation are very serious and will take a lot of effort to repair. The module exceeds its responsibility by far, indicated by 39 violations to rule 10 of type Is only allowed to use. ServiceImplementation makes use of BusinessComponents and Pipeline, bypassing BusinessProcesses, with as consequence that it duplicates responsibility of these components. Furthermore, it violates the facade convention (rule 5) of component BusinessProcesses. The violations may be the result of convenience. For example, in a situation where a new business process or a new business component is needed, a developer might choose a short track.

Bypassing the interfaces of the three components in the Logic layer (rule 4, 6, 7 in Table II) is problematic, but the severity is not equal in all cases. In general, bypassing an interface of a component in another layer is more serious than bypassing an interface of a component in the same layer. Consequently, the violations by Service.ServiceImplementation and Data.ExternalServiceAgent are more serious than those caused by the components within the Logic layer.

2) Adjustment of the Intended Architecture

The architecture is really intended, so will not be adjusted, based on the SACC. However, an exception applies: the facade convention of module Common.Infrastructure. The interface classes of this component were added for reasons of testability and should not be removed, but the facade convention rule seems to be valid for only a part of the contained classes.

E. Architecture Erosion

To answer research question 3, we analyzed another version of the source code, nearly three years older. Version 2 in Table III represents the current version at the moment of the SACC (May 2015), while version 1 was nearly three years older. Table III shows that architecture compliance has decreased in these three years, indicating architecture erosion. The number of violated rules has increased (from seven in version 1 to ten in version 2) as well as the number of violations (from 586 in version 1 to 654 in version 2; an increase of 12 percent). Exclusion of rule 3, because of its relatively very high number of violations, results in an increase of 80 percent going from version 1 to version 2.

Interestingly, nearly all rules show an increase in the number of violations in version 2. For nine rules, the number of violations increased, and for one rule the number of violations stayed equal.

The system architect could not remind causes for the increase in violations over the versions, since they had taken place over the years. Finally, we looked for possible correlations between rule type and fluctuations in the number of violations, but we did not find a meaningful pattern.

VI. Discussion

A. Answers to the Research Questions

The case study of the ServiceComponent within the Ecommerce system has provided answers to the research questions described in Section I.

1) Is SRMA support suitable within the context of the case?

To answer this question, we studied the fit between the types of modules and rules in the intended architecture of the case system and the sets of types provided by HUSACCT. We concluded with an affirmative answer, based on the following arguments.

- All five types of modules supported by HUSACCT are included in the intended architecture of the case system: Layer, Components with Interface, Subsystem, and External system.
- Eight of the eleven types of rules supported by HUSACCT are included in the intended architecture of the case system, and six of them occur more than once. The three rule types not used in this case are: Inheritance convention, Visibility convention, and Must use. On further consideration, Must use rules could have been added to all rules of the types Is only allowed to use and Is the only module allowed to use .
- We did not encounter logical rules that could not be included in the intended architecture within the tool, nor logical rules that required a lot of rules at tool level.
- We did not encounter module types with semantics that do not fit within the provided set of five module types. However, we have put a potential new module type on our think-list: Gateway, with as default rule type Is the only module allowed to use.

2) Is SRMA support useful in the SACC process?

An affirmative answer to this question is based on the arguments below.

- The semantical differences between the different modules helped to comprehend the intended architecture and to specify the architectural constraints. The module types and rule types helped to express the main principles and patterns of the case system's modular architecture: layering, implementation hiding of internal components, and hiding of relevant external systems (gateway pattern).
- The rule types of the reported violated rules assisted to get an impression of the severity of the violations. The case system's technical leader was able to express expectations on the impact of certain violations, based on the type of rule and the position and type of the affected module; without studying the code.
- The provided extensive SRMA support saved time during the definition of the intended architecture. For instance, seven of the seventeen main rules were defined automatically, based on the type of a created module.

3) Is architecture erosion identifiable in this case?

Yes, comparison of the recent version with a code version

TABLE III. All violated rules with the numbers of reported violations in two different versions

Id	Logical module from	Rule type	Logical module to	Violations vs. 2	Violations vs. 1
1	Data.DataAccess	Is the only module allowed to use	xLibraries.System.Data	7	3
2	Data	Is not allowed to back call		12	0
3	Infrastructure.Infrastructure	Facade convention		496	489
4	Logic.BusinessComponents	Facade convention		5	2
5	Logic.BusinessProcesses	Is not allowed to use	Data.DataAccess	3	0
6	Logic.BusinessProcesses	Facade convention		6	0
7	Logic.Pipeline	Facade convention		81	77
8	Logic.Pipeline	Is the only module allowed to use	xLibraries.CommerceServer	3	1
9	Service.EdsServiceAgent	Is only allowed to use	Service.ServiceContracts	2	2
10	Service.ServiceImplementation	Is only allowed to use	Service.ServiceContracts	39	12
			Total:	**654**	**586**

nearly three years older, revealed an increase of 12 percent of violations. The average increase of nine out of ten rules (excluding rule 3) was even 80 percent.

Furthermore, we did not find a meaningful correlation between the rule types and the fluctuations in violations.

B. Points for Improvement of HUSACCT

During the conduction of the SACC, we noted some points for improvement of HUSACCT. The following are implemented.

- An intended architecture report was missing. We added one that provides tabular overviews of all the modules, their assigned code, and the defined rules with their exceptions.
- The violation report contained a table with all the violation messages, but missed an overview of violations per rule. We added an overview of the violated rules (as in Table II) and an overview of the not-violated rules; also important.
- We missed the option in diagrams to view violations only, so we added it to improve the comprehensibility of diagrams with many elements.

The presented intended architecture diagrams are useful to get an overview of the modules (with their types) that cause the most violations. But, we missed an intended architecture diagram that shows the modules (with their types) and the rules (with their types). Such a type of diagram requires research, so we have added it to our list of future work.

C. Threats to Validity

To reflect on the limitations of our study, we have made use of the validity threats as described by Wohlin et al. [11]. It seems to us that the most relevant limitations of our work are related to the following types of threats: internal validity, and external validity.

With respect to the internal validity, we need to mention that we cannot guarantee that all dependencies and violations present in the case system's source code are reported. We have ensured the validity of the C# code analysis functionality by means of extensive automated tests, including (though exceeding) the test cases within the benchmark test [12]. However, since many code variations are possible, some variations may not be reported (especially not in case of usage of external systems, as discussed before). Even though not all individual usages of a class within another class will be reported, chances are much smaller that not one of these dependencies will be reported. One of the reasons why we discussed most violations at rule level, or class level, and not at the level of individual usages.

The validity can be threatened also by false positives; incorrect violation messages. However, we think that is quite unlikely. To ensure validity at this point, we have checked some of the reported dependencies in code of the case system, but not all (by far). Moreover, in accuracy tests with SACC-tools [12], no false positives were detected at all.

With respect to the external validity, it is clear that our findings cannot be generalized; e.g. the number of architectural violations present in a software system. We have discussed one case system of one organization in one country, written in C#, and tested with HUSACCT. However, in our experience, and as described in related work, it is quite common that implemented architectures diverge from intended architectures.

D. Related Work

A survey of Nugroho and Chaudron [13] on design-code correspondence revealed that design incompleteness is an important source of non-correspondence. In line with their finding, we noted that the architecture was not documented during the main development of the system; only afterwards.

Several other case studies on the application of SACC are published. Most interesting for comparison to this case study are studies that mention different types of modules, rules, or patterns. Buckley et al. [14] describe the results of five case studies in four organizations. In all five cases, violations were detected. In support of our approach, the paper describes that the participating architects were trying to check the conformance to architectural patterns like layers, and the usage of facades to attain implementation hiding. Furthermore, the participants expressed their wish to include the usage of external components in the analysis.

Herold et al. [15] describe an interesting rule-based approach to check if a system conforms to six architectural patterns described in the reference architecture for the German public administration. Two of their six patterns are related to the uses style [3], and comparable to two rule types in our approach (Facade convention, and Is the only module allowed to use). All eight violations found in this case study are violating rules of these two patterns. Two other patterns are related to the decomposition style (a component should have: a facade; an exception facade). In our approach, violations to these types of rules are detected manually during the definition of the intended architecture. Finally, two patterns focus on restrictions regarding the implementation of component facades (e.g., each method should be surrounded by a try-catch clause). Conformance to these two patterns cannot be detected in our approach. Although layers are present in the reference architecture, no rules related to layers are checked in this case study. In contrast, another case study from this research group [16] focused on compliance to the layered architecture of a case system. The compliance check revealed that five rules were violated. In all cases because of back calls.

A comparison between our SRMACC-based approach and other SACC approaches, like [17], [18], [19], [20] and [21] is discussed in [6]. Furthermore, in [5] we reported on the results of an SRMA-test on eight academic and commercial SACC-tools. We demonstrated that the SRMA support of these tools was limited. Only three[1] of the eight tested tools in this study were providing some kind of support for layers, components, and facades. SAVE supported the graphical definition of different types of modules, but provided no further support of their semantics in the SACC process. Sonargraph Architect supported the facade pattern explicitly. Structure101 supported the concept of layering explicitly. Compared to these tools, our approach adds extensive and configurable support of five

[1] SAVE - version 1.7 - iese.fraunhofer.de;
Sonargraph Architect - version 7.0 - hello2morrow.com; Structure101 - version 3.5 - structure101.com.

common module types and eleven rule types in a consistent way, which allows extension of the set of types. Furthermore, in our approach violations are communicated per rule with an explicit rule type, to aid architecture reasoning.

VII. CONLCUSION

In this case study, we have applied a Software Architecture Compliance Checking (SACC) approach that acknowledges semantical differences between different types of modules and rules in the intended architecture. In the previous sections, we introduced the case system (a governmental application), presented the intended architecture of this system, and we described the number and causes of the detected violations of ten of the seventeen rules. In addition, we described the interpretation of the violations by the system's software architect.

Furthermore, we have demonstrated how the different rule types aid architecture reasoning, and how the types of rules indicate architectural measures or patterns in the intended architecture. This knowledge is valuable during the interpretation of the severity of violations.

We started the case study with a number of research questions, which are finally answered as follows.

1) Is SRMA support suitable within the context of the case? Yes, a suitable module or rule type could be assigned to all modules and rules of the intended architecture of the case system. Furthermore, the case's intended architecture appeared to contain modules of all five supported types, and rules of eight of the eleven supported types.

2) Is SRMA support useful in the SACC process? Yes, the differences in type between the modules and rules help to comprehend the intended architecture, and they aid reasoning on the severity of the detected violations. Furthermore, since a part of the modules and rules is generated automatically, based on semantic relations, time is saved during the registration of the intended architecture.

3) Is architecture erosion identifiable in this case? Yes, comparison with a nearly three years older code version revealed an increase in the number of violated rules and in the number of violations per rule.

In the course of the SACC process, we noticed a number of points for improvement of HUSACCT, the used SACC-tool. Several of these improvements have been implemented, but one (an additional type of intended architecture diagram that visualizes the different types of rules that constrain the modules) requires proper research, so we have added it to our list of future work. This list contains more items. For example, we intend to perform more case studies, and we intend to study how architects use HUSACCT. Furthermore, we intend to study the effect of the inclusion of an SACC-tool in the software development process of a professional organization.

REFERENCES

[1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[2] D. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Softw. Eng. Notes*, vol. 17, pp. 40 – 52, 1992.

[3] P. Clements, F. Bachmann, L. Bass, D. Garlan, P. Merson, J. Ivers, R. Little, and R. Nord, *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2010.

[4] S. Ducasse and D. Pollet, "Software Architecture Reconstruction: A Process-Oriented Taxonomy," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 573–591, 2009.

[5] L. Pruijt, C. Köppe, and S. Brinkkemper, "Architecture Compliance Checking of Semantically Rich Modular Architectures: A Comparison of Tool Support," in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 220–229.

[6] L. Pruijt and S. Brinkkemper, "A metamodel for the support of semantically rich modular architectures in the context of static architecture compliance checking," in *WICSA 2014 Companion Volume*, 2014, pp. 1–8.

[7] L. Pruijt, C. Köppe, J. M. van der Werf, and S. Brinkkemper, "HUSACCT: Architecture Compliance Checking with Rich Sets of Module and Rule Types," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, 2014, pp. 851–854.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissedes, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1995.

[9] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns, Volume 1*. John Wiley & Sons, 1996.

[10] M. Fowler, D. Rice, M. Foemmel, E. Hieatt, M. Mee, and R. Stafford, *Patterns of enterprise application architecture*. Addison-Wesley, Boston, MA, USA, 2003.

[11] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer, 2012.

[12] L. Pruijt, C. Köppe, and S. Brinkkemper, "On the Accuracy of Architecture Compliance Checking: Accuracy of Dependency Analysis and Violation Reporting," in *21st International Conference on Program Comprehension*, 2013, pp. 172–181.

[13] A. Nugroho and M. R. V. Chaudron, "A Survey of the Practice of Design -- Code Correspondence amongst Professional Software Engineers," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 2007, pp. 467–469.

[14] J. Buckley, N. Ali, M. English, J. Rosik, and S. Herold, "Real-Time Reflexion Modelling in Architecture Reconciliation: A Multi Case Study," *Inf. Softw. Technol.*, vol. 61, pp. 107–123, 2015.

[15] S. Herold, M. Mair, A. Rausch, and I. Schindler, "Checking Conformance with Reference Architectures: A Case Study," in *Enterprise Distributed Object Computing Conference (EDOC)*, 2013, pp. 71–80.

[16] C. Deiters, P. Dohrmann, S. Herold, and A. Rausch, "Rule-based architectural compliance checks for enterprise architecture management," in *Proceedings - 13th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2009*, 2009, pp. 183–192.

[17] G. C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models," *ACM SIGSOFT Softw. Eng. Notes*, vol. 20, no. 4, pp. 18–28, Oct. 1995.

[18] J. Knodel and D. Popescu, "A Comparison of Static Architecture Compliance Checking Approaches," in *Working IEEE/IFIP Conference on Software Architecture*, 2007, pp. 12–21.

[19] J. Adersberger and M. Philippsen, "ReflexML: UML-based architecture-to-code traceability and consistency checking," in *5th European Conference on Software Architecture*, 2011, pp. 344–359.

[20] R. Koschke and D. Simon, "Hierarchical Reflexion Models," in *Working Conference on Reverse Engineering, WCRE*, 2003, pp. 36-45.

[21] R. Rahimi and R. Khosravi, "Architecture conformance checking of multi-language applications," *Int. Conf. Comput. Syst. Appl.*, pp. 1–8, May 2010.