Architecture Compliance Checking of Semantically Rich Modular Architectures

A Comparative Study of Tool Support

Leo Pruijt, Christian Köppe
Information Systems Architecture Research Group
HU University of Applied Sciences
Utrecht, The Netherlands
{leo.pruijt, christian.koppe}@hu.nl

Sjaak Brinkkemper
Department of Information and Computing Sciences
University Utrecht
Utrecht, The Netherlands
s.brinkkemper@uu.nl

***Abstract*—Architecture Compliance Checking (ACC) is an approach to verify the conformance of implemented program code to high-level models of architectural design. ACC is used to prevent architectural erosion during the development and evolution of a software system. Static ACC, based on static software analysis techniques, focuses on the modular architecture and especially on rules constraining the modular elements. A semantically rich modular architecture (SRMA) is expressive and may contain modules with different semantics, like layers and subsystems, constrained by rules of different types. To check the conformance to an SRMA, ACC-tools should support the module and rule types used by the architect. This paper presents requirements regarding SRMA support and an inventory of common module and rule types, on which basis eight commercial and non-commercial tools were tested. The test results show large differences between the tools, but all could improve their support of SRMA, what might contribute to the adoption of ACC in practice.**

Keywords—Software Architecture; Modular Architecture; Architecture Compliance; Architecture Conformance; Architectural Erosion; Static Analysis

I. INTRODUCTION

Software architecture is of major importance to achieve the business goals, functional requirements and quality requirements of a system. However, architectural models tend to be of a high-level of abstraction, and deviations of the software architecture arise easily during the development and evolution of a system [1]. Architecture Compliance Checking (ACC) is an approach to bridge the gap between the high-level models of architectural design and the implemented program code, and to prevent decreased maintainability, caused by architectural erosion. *Architectural erosion* is "the phenomenon that occurs when the implemented architecture of a software system diverges from its intended architecture" [2]. Opposing terms are architecture compliance and its synonym architecture conformance. Knodel and Popescu defined *architecture compliance* as "a measure to which degree the implemented architecture in the source code conforms to the planned software architecture" [3].

Many tools and techniques are available to analyze a software system, and to reconstruct, visualize, check, or restructure its architecture [4]. In our study we focus on tools supporting static ACC, which analyze the software without executing the code. These tools, which we label as *static ACC-tools,* focus on the modular structure in the source code and identify structural elements such as packages and classes. In addition, they analyze use-relations between these elements, such as an invocation of a method or access of an attribute. Furthermore, these tools support the definition of rules on the structural elements in the code, or on logical modular elements that are mapped to the code. Finally, ACC-tools check the compliance and report violations to the rules. For example, if a method call from class A to class B in the code corresponds with a not-allowed dependency from a lower layer to a higher layer in the intended architecture, then the tool should report a violation.

Although Shaw and Clements included ACC in 2006 in their list of promising areas [5], the adoption of ACC-tools is still limited [2], [6], and research is necessary to advance and improve current methods and tools [7]. A few studies have compared ACC-tools and techniques, and these studies revealed large differences in terminology and approach. A high level overview of techniques and tools is included in a survey on architectural erosion [2] and in a survey on software architecture reconstruction [4]. Two other studies [3], [8] identified and compared five static ACC techniques at a more detailed level. One of these studies [8] also explored the effectiveness and usability of three tools, each representing one technique, by executing tests on the basis of a small system.

Our research builds on these previous studies, but we focus on ACC-tool support of *semantically rich modular architectures* (SRMAs). We use this term for expressive modular architectures, composed of different types of modules, which are constrained by different types of rules; explicitly defined rules, but also rules inherent to the module types. Kazman, Bass, and Klein have stated the principle that elements in a software architecture should be coarse enough for human intellectual control, but also specific enough for meaningful reasoning [9]. Modules with specific semantics, like subsystems, layers, components or facades, enhance the expressiveness of a modular architecture and support architecture reasoning. Adersberger and Philippsen consider the support of semantically rich architecture models essential for the integration of ACC in model-driven engineering [10]. Furthermore, they make clear that support of semantically rich constructs reduces the number of rules that need to be

defined, compared to semantically poorer boxes and lines models.

We started our study with the following research question: *Do static ACC-tools provide functional support for semantically rich modular architectures?* To answer this question, we identified requirements, developed test-ware based on the requirements, and we tested eight ACC-tools. We restricted our study to the functional support of SRMAs by ACC tools, and consequently we do not focus on other aspects, like usability, scalability or accuracy (in another study, we investigated the accuracy of dependency analysis and violation reporting [11]). Other approaches than ACC that may be supported by the same tools, like architecture reasoning and re-engineering, are outside the scope of this paper as well.

The next section of this paper identifies the information available in semantically rich modular architectures, presents requirements and a classification of common module and rule types. Section 3 describes the test method and introduces the tools, while Section 4 holds the test results. Section 5 discusses the test outcome and compares it to related work, while Section 6 concludes this paper with recommendations, and addresses some issues that require further research.

II. MODULAR ARCHITECTURES

A. Focus of Static ACC

Software architecture compliance checking covers a large field, since software architecture is a broad term. According to Perry and Wolf, software architecture "provides the framework within which to satisfy the system requirements and provides both the technical and managerial basis for the design and implementation of the system" [12]. Static ACC does not cover the full width of software architecture, but only the static structure of the software: the modular architecture. According to the Views and Beyond approach [13], [14], module styles focus on the structure of the units of implementation and not on runtime behavior or the allocation to non-software resources. Different module styles are defined such as the decomposition style, uses style, generalization style, and layer style.

A modular architecture should describe the modular elements, their form (properties and relationships) and rationale [12]. Modular elements, properties and relationships, are in ACC's center of attention, and should be included in a complete compliance check. A *modular element*, or module, is an implementation unit of software with a coherent set of responsibilities [14]. 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 [12].

B. Requirements Regarding SRMA Support

A semantically rich modular architecture may contain a lot of information about the modules and the rules constraining these modules. Modules may be of types with different semantics, while different types of rules may be used to constrain the modules. A rich set of module types provides a language to express characteristics of the modules in an architectural model, as well as default constraints associated to the type of module. A rich set of rule types provides a language to express constraints on the modules in an architectural model. Provision of a rich rule set allows architects to define logical rules in a comparable way as expressed in regular language, without the need to translate a logical rule to one or more rules at tool level.

Consequently, to support compliance checks of SRMA's, ACC-tools should preferably be able to: a) register common information in SMRAs (modular elements, properties and relationships of different types); b) prevent inconsistencies in the definition of the architectural model; and c) check the rules included in the architectural model and report violations. Inconsistencies in the model, like modules not properly mapped to code, will hamper the accuracy of the actual rule check. Consequently, inconsistencies should be recognized and reported.

In line with these requirements, we focused our research on the following questions. Do ACC-tools provide support for: a) common types of modules and their semantics; b) common types of rules; and c) inconsistency prevention within the defined architecture?

To determine the module types, rules types and inconsistency checks relevant to our research, we studied academic and professional literature, as well as software architecture documents from professional practice and ACC-tool documentation. The following subsections describe the outcome of our study.

C. Common Module Types.

SMRAs may contain modules of different types. We identified six common types of modules relevant for static ACC:

1) Physical clusters are the type of modules that represent a wide variety of software structures or units in the code, like classes, Java packages, or C# namespaces [14]. This type of module does not represent a unit in the design, but in the code.

2) Logical clusters represent units in the system design with clearly assigned responsibilities, but with no additional semantics. Comparable terms are subsystems, or packages.

3) Layers represent units in the system design with additional semantics. Layers have a hierarchical level and constraints on the relations between the layers. The concept of layering can be traced back to the works by Dijkstra [15] and Parnas [16]. Although the layered style is not supported by UML [5], it is one of the most common styles used in software architecture [14], [17]. We cite Larman [18], who summarizes the essence of a layered design as "the large-scale logical structure of a system, organized into discrete layers of distinct, related responsibilities. Collaboration and coupling is from higher to lower layers."

4) Components within a software architecture are designed as autonomous units within a system. The term component is defined in different ways in the field of software engineering. In our use, a component within a

modular architecture covers a specific knowledge area, provides its services via an interface and hides its internals (in line with the system decomposition criteria of Parnas [16]). Consequently, a component differs from a logical cluster in the fact that it has a Façade sub module and hides its internals. Since our definition of component is intended for modular architectures, it does not include runtime behavior, and a module in a module view may turn into many runtime components within the "component and connector view" [14].

5) Façades are related to a component and act as an interface as described under components. We use the term façade, referring to the façade pattern [19], to differentiate with the Java interface, which has not exactly the same meaning as a design-level interface. A façade may be mapped to multiple elements at implementation level, like Java interface classes, exception classes and data transfer classes.

6) External systems represent platform and infrastructural libraries or components used by the target system. Useful ACC support includes the identification of external system usage and checks on constraints regarding their usage [20].

D. Example of an SRMA

An example of an SRMA, with modular elements of different types, is shown in Fig. 1. The model shows a part of a modular architecture of one of the systems at an airport, where it was subject of an ACC. This system is used to manage the state and services of human interaction points where customers communicate with baggage handling machines, self-service check-in units, et cetera.

Various notations for modular architecture diagrams are used in practice [14]. The example in Fig. 1 shows UML icons, but also an identification of the layers, not included in UML. The model combines three modular styles, namely the decomposition style, uses style, and layered style. Examples of modules of different types are visible in Fig.1. such as "Interaction layer", logical cluster "HiWeb", component "HiManager", façade "HimInterface", and external system "Hibernate". The modules are easily identifiable, but the rules are not. In this case, the basic principle is, "no module is allowed to use another module", except when a dependency relation indicates "is allowed to use". Furthermore, the rules related to the layered style are not visible, but the default rules apply: Interaction Layer is not allowed to use Technology Layer (skip call ban); Technology Layer is not allowed to use Service layer or Interaction Layer (back call ban).

E. Common Rule Types.

SMRAs may contain rules of different types, where each rule type characterizes the constraint. Constraints in a software architecture are categorized in literature [12], [14] as properties and relationships. Our inventory of architectural rule types, in principle verifiable by static ACC, resulted in two categories related to properties and relationships: Property rule types; and Relation rule types.

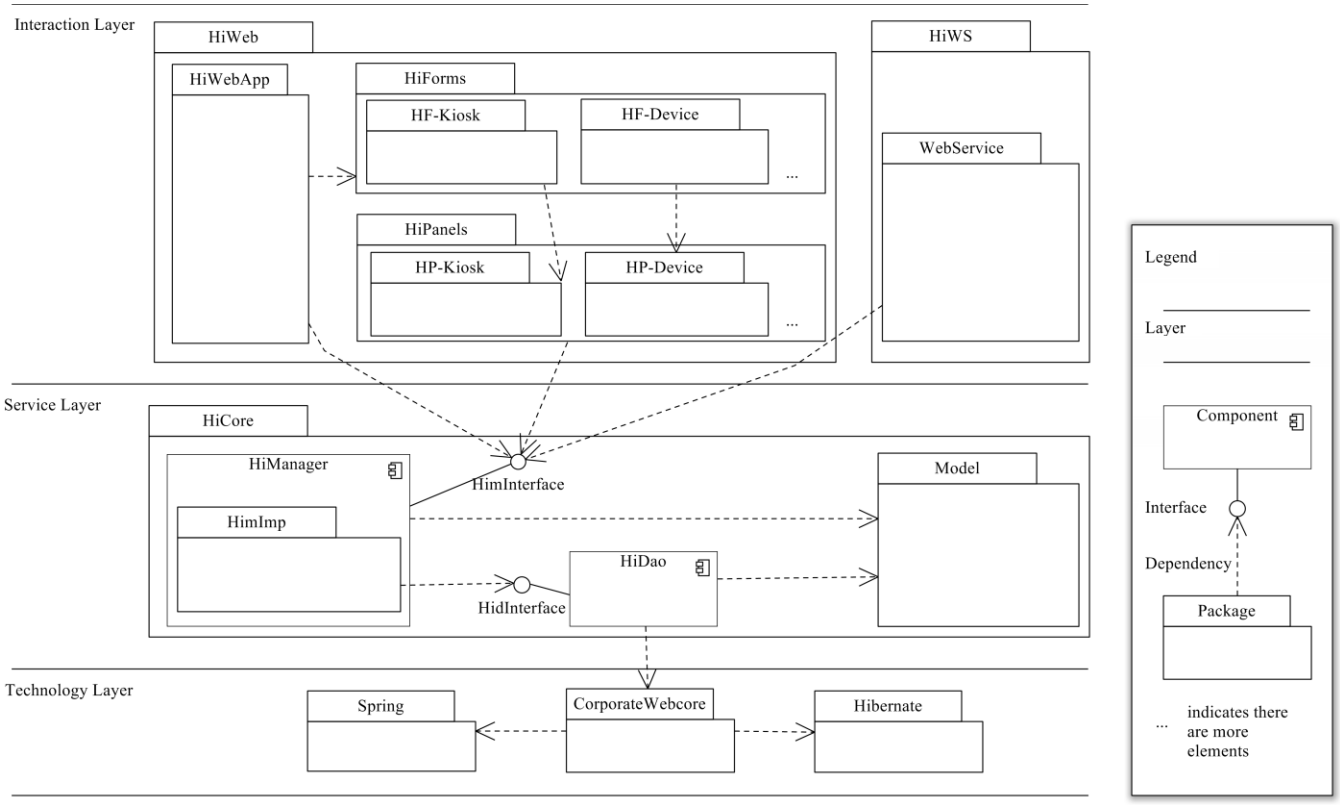

Figure 1. Example of a semantically rich modular architecture model

Property rule types constrain the elements included in the module; their sub modules, et cetera. Clements et al. [14] distinguish the following properties per module: Name, Responsibility, Visibility, and Implementation information. We identified rule types associated to these properties and named them accordingly, except two types (Façade convention, Inheritance convention), which represent the property Implementation information. The identified rule types are shown in Table I. The table contains per rule type: a description, an example, and an exemplary reference to literature covering the topic. The example rules constrain the modules of the modular architecture shown in Fig.1.

Naming conventions may be useful, since names are used by practitioners to unify software architecture and its implementation [21]. Responsibility conventions are useful to preserve the designed distribution of responsibilities over modules. Visibility conventions and Façade conventions can be used to enforce implementation hiding. Inheritance conventions may be used to enforce a selected generalization style. Finally, exceptions to property rules may be useful too. For instance, an exception to the Visibility convention example in Table I is, "HiManager classes have package visibility or lower, except for façade HimInterface."

Relation rule types specify whether a module A is allowed to use a module B. The basic types of rules are "is allowed to use" and "is not allowed to use". However, we encountered useful specializations of both basic types, which we included in the classification shown in Table I. When several rules of the same type are defined on the same from-module, then they should be interpreted as complementary rules; even if the word "only" is part of the name of the rule type.

Some rule types are complex, because they include dependency checks on other modules than only the from-module and to-module. Exceptions to all relation rules are complex, as well as the two following types: "Is only allowed to use", and "Is the only module allowed to use". Complex rule types are very useful in practice, for the following reasons:

- Complex rule types allow architects to define rules in a comparable way as expressed in regular language. Complex rules of type "Is only allowed to use" may constitute a significant part of the total rule set [22].
- Complex rule types help to transform rules in a UML-like diagram to rules in most ACC-tools. For instance, the dependency relationship from module HF-Kiosk to module HP-Kiosk in Fig.1 expresses the rule "HF-Kiosk is only allowed to use HP-Kiosk." Transformation is often necessary. The basic principle underlying UML-like diagrams is restricting (no other than the defined dependencies are allowed), while in most tools, the basic principle is non-restricting (all dependencies are allowed, unless there is a not-allowed-to-use rule).
- Complex rule types may diminish the number of rules, since one complex rule often replaces many "is not allowed to use" rules. For instance, when the "is only allowed to use" rule type is not supported by a tool, than the dependency relationship from module HF-Kiosk to module HP-Kiosk in Fig.1 may have to be translated to many "not allowed to use" rules from HF-Kiosk to all the other modules, except to HP-Kiosk.

TABLE I. COMMON RULE TYPES (REF= PRIMARY LITERATURE REFERENCE)

Category\Type of Rule	Description (D), Example (E)	Ref
Property rule types		
Naming convention	D: The names of the elements of the module must adhere to the specified standard. E: HiDao elements must have suffix DAO in their name.	[8]
Responsibility convention	D: All elements of the module must adhere to the specified responsibility. E: HiForms is responsible for presentation logic only.	[18]
Visibility convention	D: All elements of the module have the specified or a more restricting visibility. E: HiManager classes have package visibility or lower.	[14]
Facade convention	D: No incoming usages of the module are allowed, except via the façade. E: HiManager may be accessed only via HimInterface.	[19]
Inheritance convention	D: All elements of the module are sub classess of the specified super class. E: HiDao classes must extend CorporateWebCore.Dao.GenEntityDao.	[8]
Relation rule types		
Is not allowed to use	D: No element of the module is allowed to use the specified to-module. E: HF-Kiosk is not allowed to use HP-Device.	[3]
Back call ban (specific for layers)	D: No element of the layer is allowed to use a higher-level layer. E: Service Layer is not allowed to use the Interaction Layer.	[23]
Skip call ban (specific for layers)	D: No element of the layer is allowed to use a lower layer that is more than one level lower. E: Interaction Layer is not allowed to use the Infrastructure Layer.	[23]
Is allowed to use	D: All elements of the module are allowed to use the specified to-module. E: HiWebApp is allowed to use HiForms (including its sub modules).	[14]
Is only allowed to use	D: No element of the module is allowed to use other than the specified to-module(s). E: HF-Kiosk is only allowed to use HP-Kiosk.	[8]
Is the only module allowed to use	D: No elements, outside the selected module(s) are allowed to use the specified to-module. E: HiDao is the only module allowed to use CorporateWebcore.	[8]
Must use	D: At least one elements of the module must use the specified to-module. E: HiDao must use CorporateWebcore.	[3]

F. Associations between Module and Rule Types

Optimal support of SRMAs includes the automatic provision of rule types inherent to the type of module. For instance, layers are inherently associated to a "Back call ban" rule and a "Skip call ban" rule. Furthermore, components are inherently associated to a "Façade convention" rule (and possibly a "Visibility convention" rule, if supported by the implementation language). Options to disable an inherent rule, for instance in case of a relaxed layered model, or to define an exception, will enhance the usability.

III. TEST METHOD AND TESTED TOOLS

A. Test Method

Based on the requirements and classification of module types and rule types described in Section 2, a test was designed to assess the ACC-tools on their SRMA support. For each rule type, at least two test cases were included: one without, and one with violations to the rule. A special test software system was developed in Java. This system included the various module types and separate packages for each rule type, which contained classes with injected violations to a rule and classes without. In addition, a test script was prepared to instruct the tester and to document the test results. The test script and test system are available on request.

After the test preparation, the eight ACC-tools were tested. During the first step of the test of a tool, the intended architecture was entered. Thereafter, the modules were mapped to source code units and the rules were entered into the tool. If a tool did not support a rule type explicitly, then we looked for a workaround; such as a combination of separate rules. The first step was concluded by test actions aimed at the tool's ability to prevent inconsistencies in the architecture definition. During the second step, the outputs of the tool's dependency analysis and conformance check were studied and compared with the expected result. During the third step, reports were prepared, after which the tools could be compared on their SRMA support.

Two iterations of testing and reporting were conducted. The first iteration was performed with 25 bachelor students in the course of a third year specialization semester "Advanced Software Engineering", where each team studied and tested a tool. In a second iteration, the authors studied the tools and verified and refined the results of the students, by using the tools and repeating the tests. ConQAT was added afterwards to our tool set and was tested only by the authors.

B. ACC-Tools Included in the Test

Many tools are available with some facilities to support ACC. Our research focused on tools with explicit support of ACC. We selected eight publicly available tools, which were mentioned in academic work (e.g., [4] [8] [10]), were able to

TABLE II. CHARACTERISTICS OF THE TOOLS IN THE TEST

Tools [1] → **Characteristics ↓**	**ConQAT AA**	**dTangler**	**Lattix**	**Macker**	**SAVE**	**Sonar ARE**	**Sonargraph Architect**	**Structure 101**
General functionalities								
Dependency browsing		√	√				√	√
Dependency visualization			√		√		√	√
Architecture compliance checking	√	√	√	√	√	√	√	√
Architecture refactoring/simulation			√				√	√
Team support							√	√
Code variants								
Java	√	√	√	√	√	√	√	√
Other languages	√		√		√			√
Source file analysis					√	√	√	
Compiled file analysis	√	√	√	√		√	√	√
Licensing								
Free: commercial and non-commercial use	√	√		√		√		
Paid: commercial use			√		√		√	√

[1] ConQat AA– version 2011.9 – www.conqat.org;
dTangler - GUI version 2.0 - web.sysart.fi/dtangler;
Lattix LDM - version 7.2 - lattix.com;
Macker - version 0.4.2 - sourceforge.net/projects/macker;
SAVE - version 1.7 - iese.fraunhofer.de;
Sonar ARE - version 3.2 - docs.codehaus.org/display/SONAR/Architecture+Rule+Engine;
Sonargraph Architect (fusion of Sotograph and SonarJ) - version 7.0 - hello2morrow.com;
Structure101 - version 3.5 - structure101.com.

analyze Java, and provided evaluation or research licenses (two vendors rejected and one did not respond). We excluded tools that focus mainly on architecture visualization, metrics and/or architecture refactoring. The eight tools included in our study are shown in Table II, which also gives an overview of functionalities, code variants and licensing.

The tools provide their support of ACC in various ways. The eight tools can be subdivided in four categories of tools. 1) Macker and Sonar Architecture Rule Engine (Sonar ARE) are text-based tools, which support relation conformance rules. These tools provide HTML-based violation reports. 2) dTangler and Lattix are based on the Dependency Structure Matrix (DSM) technique, complemented with text-based editors to define rules. The DSM is used to select modules and to show dependencies and violations. Lattix is also able to visualize architectures graphically, and provides extensive reporting facilities.
3) ConQAT Architecture Analysis (ConQAT AA) and SAVE are strictly based on the Reflexion Model (RM) technique [1], and both tools provide a graphical editor to define the intended architecture and to show violations after the evaluation. Textual reports are generated at request. 4) Sonargraph Architect and Structure101 are diagram-based too, but these tools are not based on the RM-technique. To define modules and rules, these tools provide diagrams in which the horizontal and vertical position of a module implies rules. Violations are shown in these diagrams, but textual reports are provided in addition.

IV. TEST RESULTS

A. Support of Common Module Types

In Section 2 we identified six common types of modules, relevant for static ACC. The results of our tests concerning the support of these module types are shown in Table III, and the most interesting findings are described below.

Clusters are supported by all tools. Five of the eight tools support *physical clusters*. The advantages to use them are that they allow fast, ad hoc rule checking; for instance, when there is no formal modular architecture. The disadvantage is the diminished or lost traceability to the formal modular architecture, if there is one. Sonar ARE is the only tool that supports only this type of modules. *Logical clusters* are supported by seven tools. Although in very different ways, these tools provide support to register logical clusters and to map the logical clusters to code units. Furthermore, support is provided to define rules constraining logical clusters and to check these rules at code level.

Layers are supported by only one tool, Structure101, on all indicators: modules can be marked as layers; back call and skip call rules are reported; and layers are visualized. Two other tools support the definition and visualization of layers, but do not provide inherent support of the related rules.

Components and *Facades* are supported by SAVE and Sonargraph Architect, on the following indicators: modules can be marked as component; facades can be defined. SAVE visualizes components and facades, but does not actively

TABLE III. TOOL SUPPORT OF COMMON MODULE TYPES (+ = EXPLICIT SUPPORT; ± = PARTIAL SUPPORT; - = NO SUPPORT)

	ConQAT AA	dTangler	Lattix	Macker	SAVE	Sonar ARE	Sonargraph Architect	Structure 101
Clusters								
Physical cluster	-	+	+	+	-	+	-	+
Logical cluster	+	+	+	+	+	-	+	+
Layers								
A module can be marked as layer	-	-	-	-	+	-	+	+
Back call violations are reported	-	-	-	-	-	-	-	+
Skip call violations are reported	-	-	-	-	-	-	-	+
Components and Facades								
A module can be marked as component	-	-	-	-	+	-	+	-
Facade can be defined	-	-	-	-	+	-	+	-
Facade-skip violations are reported	-	-	-	-	±	-	+	-
External systems								
A module can be marked as external system	-	-	-	-	-	-	+	-
A module can be mapped to an external system	+	+	+	+	+	+	+	+
Rules constraining their use are checked	+	+	+	+	+	+	+	+
Visualization								
Clusters are visualized	+	-	+	-	+	-	+	+
Layers are recognizable visualized	-	-	-	-	+	-	+	+
Components are recognizable visualized	-	-	-	-	+	-	-	-
Facades are recognizable visualized	-	-	-	-	+	-	+	-
External systems are recognizable visualized	-	-	-	-	+	-	+	-

support any of their semantics. Sonargraph Architect visualizes facades and supports their semantics; it reports facade-skip violations automatically when a facade is associated to a module. ConQAT AA seems to support components at first glance, since it depicts all modules as UML components. However, it does not provide any other icons and does not support the semantics of a component; reason why we classified ConQAT's components as logical clusters.

External systems are not designated as a special module type by all tools, except Sonargraph Architect, but all enable conformance checks on modules mapped to external libraries.

Five tools support visualization of modular architectures. However, only two tools offer three or more different icons. A notable observation is that the tools that support semantically rich modules all have their own terminology, icons, rules and ways to visualize the architecture. SAVE provides an UML-like notation, while Sonargraph Architect and Structure101 position the modules horizontally and vertically. SAVE discerns five module types, while Sonargraph Architect discerns six types (which are only partly overlapping with those of SAVE), whereas Structure101 does not show the logical meaning of a module, but uses an icon to show the type of the related physical item.

B. Support of Common Rule Types

In section 2 we identified twelve common types of rules, relevant for static ACC. The results of our tests concerning the support of these rule types are shown in Table IV. Explicit support of a rule type is depicted by a "+", meaning that one logical rule can be registered as one rule in the tool. Partial support, depicted by "±", means that it is possible to register a rule of this type, but only via a workaround; often a combination of several rules. The most interesting findings from the test are described below.

1) Property rule types

Property rule types are poorly supported. No tool provides facilities to specify and check conventions regarding naming, responsibility, or inheritance. Although names are used, in combination with regular expressions, to map modules to the code, no facilities are provided to check all the packages and/or classes contained by a module on conformance to a naming convention.

Only rule types to enforce implementation hiding are supported by some tools. Visibility convention rules are partly supported by Sonargraph Architect and Structure101. These tools provide a property to restrict the accessibility of a module, but do not check at code level on accessibility settings; reason why they did not score a "+". However, when a module is marked as hidden or private, violation messages are reported, when dependencies to the module are detected from outside.

Façade convention rules are supported explicitly only by Sonargraph Architect. Four other tools enable the definition of this type of rules by default means, resulting in a combination of separate rules, so their support is scored with "±".

2) Relation rule types

Relation rule types are supported by all the tools, but no

TABLE IV. TOOL-SUPPORT OF COMMON RULE TYPES (+ = EXPLICIT SUPPORT; ± = PARTIAL SUPPORT; - = VERY WEAK OR NO SUPPORT)

Support is provided for	ConQAT AA	dTangler	Lattix	Macker	SAVE	Sonar ARE	Sonargraph Architect	Structure 101
Property rule types								
Naming convention	-	-	-	-	-	-	-	-
Responsibility convention	-	-	-	-	-	-	-	-
Visibility convention	-	-	-	-	-	-	±	±
Facade convention	-	±	±	±	±	-	+	-
Superclass inheritance convention	-	-	-	-	-	-	-	-
Relation rule types								
Is not allowed to use	+	+	+	+	+	+	+	±
Back call ban (inherent to layer)	-	-	-	-	-	-	-	+
Skip call ban (inherent to layer)	-	-	-	-	-	-	-	+
Is allowed to use	+	+	+	+	±	-	+	+
Is only allowed to use	±	±	±	±	±	-	±	±
Is the only module allowed to use	±	±	±	±	±	-	±	±
Must use	-	-	-	-	+	-	-	-
Exception (to relation rules)	±	±	±	±	-	-	±	±
Visualization of rules and violations								
Rules are visualized	+	-	+	-	+	-	+	+
Violations are visualized	+	+	+	-	+	-	+	+

more than three rule types are explicitly supported per tool.

Complex rule types (Is only allowed to use, Is the only module allowed to use, Exceptions to a relation rule) are not-explicitly supported, or not at all. Without explicit support, workarounds are needed, for instance for the rule "HF-Kiosk is only allowed to use HP-Kiosk". In Lattix, dTangler and Macker, two combined rules are needed such as: "HF-Kiosk Cannot-Use $Root" + "HF-Kiosk Can-Use HP-Kiosk". Since these rules are not related to each other, they form a threat to the maintainability and traceability of the set of rules. Sonargraph Architect and Structure101 may require the specification of more than two rules or property settings for complex rules, and sometimes many rules are needed, depending on the number and position of other modules. Sonar ARE provides no support at all to check complex rules. ConQAT AA and SAVE work quite differently from the other tools, since no transformation is required of rules in UML-like diagrams to rules in the tool. SAVE supports only the "Must use" rule type explicitly, while ConQat AA supports "Is allowed to use" and "Is not allowed to use" rule types. Complex rules can be checked, but this requires interpretation of the architecture model and the conformance check output.

3) Visualization

Six tools are able to visualize rules and violations. Lattix and dTangler show colors in a DSM. ConQAT AA, SAVE, Sonargraph Architect, and Structure101 use lines in diagrams to define and show rules, and to show violations. However, not all rules are visible in these diagrams.

C. Support of Inconsistency Prevention

In section 2 we defined the requirement, "ACC-tools should prevent inconsistent definitions of modules and rules." The results of our tests concerning this requirement are shown in Table V. Most tools allow, without a warning, incomplete or contradictory definitions of modules and/or rules. ConQAT AA scored best and prevented six out of six types of inconsistency included in our test. Lattix prevented five out of six types, while the other tools prevented or warned for upmost three types. Six of the tools start the compliance check without a warning when the defined modules and rules model is inconsistent. In such a case, the tool does not check all the rules as intended by the user, and consequently the outcome of the check may be unreliable.

V. DISCUSSION

To our opinion, all tested tools are providing useful functionality to support ACC or ad hoc rule checking. Apart from our laboratory experiments described in the paper, we used all eight tools to analyze an open source system. Furthermore, we performed ACCs on professional software systems with use of Lattix, Sonargraph Architect, and Structure101. Based on these experiences we can conclude that these tools are of great help for architecture reconstruction and ACC. However, our tests show that all eight tools could improve their support regarding SRMAs, though in varying degrees. Not one of the tested tools is able to support all the module types and rule types included in our classification. However, we encountered interesting examples of partial support. SAVE supports the graphical definition of modules of nearly all the types in our classification; only physical clusters are missing. However, SAVE's rule language is very limited, and the semantics of the modules are not supported. ConQAT provides the same types of diagrams, but complements the rule setting capabilities considerably. Furthermore, ConQAT checks the consistency of the defined architectural model accurately. However, ConQAT provides one type of module only, and does not support any semantics. Sonargraph Architect and Structure101 are the only tools that actively support the semantics of two module types in our classification. Sonargraph Architect supports the definition of Facades and relates the "Façade convention" rule to a defined façade. Structure101 supports the definition of Layers and relates a layer to the "Back call ban" and the "Skip call ban" rules. Combination of these examples of partial support builds an image of the provision of full functional SRMA support.

Another observation during our study is that the combination of visualization, rule definition, and rule checking appears to be challenging. Lattix, dTangler, and Macker provide no support to define the architecture via a graphical editor, but enable the definition and checking of quite a diversity of rules, including complex rules. ConQAT, SAVE, Sonargraph Architect and Structure101 provide graphical support to define and check the architecture, but lack the freedom of rule definition, as provided by the

TABLE V. PREVENTION OF INCONSISTENCIES (+ = SUPPORTED; - = NOT SUPPORTED; N/A = NOT APPLICABLE)

	ConQAT AA	dTangler	Lattix	Macker	SAVE	Sonar ARE	Sonargraph Architect	Structure 101
Modules must have (unique) name or ID	+	+	+	+	-	n/a	+	-
A module may have only one parent.	+	-	+	-	+	n/a	-	-
Modules must be mapped to code file(s)	+	-	-	-	-	n/a	-	+
Mapped code files must exist	+	-	+	-	-	n/a	-	-
Rules must be completely specified	+	-	+	+	+	-	+	+
Rules cannot be contradictory	+	-	+	-	+	-	+	+
Tool checks model prior to conf. check	+	-	+	-	-	-	-	-

textual-rule based tools. Furthermore, sometimes we experienced serious problems related to the graphical models. Defining sub-subsystems, exceptions, and other complex rules in the graphical models, is hard in some tools, and impossible in others. Furthermore, it may result in many lines, which makes the diagrams unreadable. Structure101 and Sonargraph Architect have introduced additional rule-setting techniques to reduce the number of required rule-lines. In these tools, the module type, the horizontal and vertical position, and the value of a visibility property per module may imply dependency rules. On top of that, Sonargraph Architect provides a "transversal access" variable per module as well. To our opinion, the combination of all the rule-setting techniques increases the complexity considerably, and it reduces the transparency of the set of defined rules.

A. Limitations

Our study can be characterized as a quasi-experiment, according to Wohlin et al. [24], since we did not work with a randomized selection of tools. Consequently, our findings may not be generalized to other tools, even though we tested eight tools in a small market.

Furthermore, we do not claim that our classification of common module types and common rule types is complete, since *common* is not a qualified term. We aimed to cover the most used types of modules and rules, reasoning from the functional point of view; the architect's view, not the tool builder's view. Creating the classification proved a valuable step in our study. The classification was used as a basis for our tests and will be used as starting point for our future work.

B. Related work.

Requirements regarding the functional support of ACC can be derived from quite a number of sources, like general literature on software architecture and design, and studies on ACC. In Section 2 we described the most relevant sources used for our requirements and classification. Several studies on ACC propose the inclusion of support for some specific module and/or rule types, for instance [10] [20] [3] [8]. However, to the best of our knowledge, none of these studies or other studies on ACC have provided and substantiated a broad inventory and classification of module and rule types. We intentionally did not include very specific or detailed module or rule types, but kept the set of requirements broad and not too specific. However, some interesting studies elaborate on particular types. For instance, Adersberger and Philippsen [10] describe the constraints and checks regarding components in detail. Furthermore, Terra and Valente [22] identified different types of dependencies (accessing methods and fields, declaring variables, creating objects, extending classes, implementing interfaces, throwing exceptions, and using annotations), and based fine grained rule types on these dependency types. Lattix, SAVE and Structure101 provide support to define or configure rules at this level of detail.

Not much comparative research on ACC-tools has been performed, as described in the Introduction section. Only Passos et al. [8] presented similar work. They evaluated three tools, including Lattix and SAVE, on the basis of a very small system. During our study no findings have arisen that contradict their tool evaluations. Our study adds a substantiated set of requirements focused on SRMA support, as well as test results of eight tools.

VI. CONCLUSION

Architecture compliance checking (ACC) relies on the support of tools to define modules and rules, to analyze the code, to check the compliance, and to report violations to the rules. In this study, we have investigated the support of semantically rich modular architectures (SRMAs) provided by static ACC-tools. We identified requirements to the support of SRMAs and classified module types and rule types relevant for static ACC. Furthermore, we prepared a test, and we tested eight tools on their support of SRMAs.

We started our study with the following research question: Do static ACC-tools provide functional support for semantically rich modular architectures? We focused our test on the support of: a) common types of modules and their semantics; b) common types of rules; and c) inconsistency prevention within the defined architecture.

Our tests regarding the *support of common module types* show that five tools support non-semantic clusters only. The three other tools distinguish also one or more semantically rich module types from our classification. SAVE supports the graphical definition of five types of modules, but does not support their semantics. Sonargraph Architect supports the semantics of a Façade actively, while Structure101 supports the semantics of Layers actively. However, no tool provides the combined support of layers, components, and facades.

Our tests regarding the *support of common rule types* show that per tool only a few rule types are explicitly supported. Complex relation rules are by no tool explicitly supported. Consequently, complex relation rules at logical level require workarounds at tool-level, which often result in two or more unrelated rules; a threat to the maintainability and traceability of the set of rules. Furthermore, only two of the five property types are supported, and only partially, not explicitly.

Our tests regarding the *support of inconsistency prevention* show that only two tools, ConQAT and Lattix, score high on the prevention of inconsistencies in the module and rule model, while inconsistent models may result in an unreliable outcome of the compliance check.

Based on our study and experiments, we present the following recommendations to ACC-tool developers: 1) *Widen the scope of the tools from dependency checking to software architecture compliance checking, including SRMAs.* Provide explicit support for semantically rich module types with their related rule types. The requirements and the classification of common module and rule types, presented in this paper, may be used as a starting point. 2) *Minimize the difference between logical rules, as perceived by the architect, and the technical implementation in the tool.* Offer rule types that match with logical rule

types, including exceptions, and support each type explicitly. 3) *Provide one method to define and edit rules.* Do not mix several rule setting mechanisms. Keep it simple to the user. 4) *Provide several, best adaptable, views* on the modular structures, the rules, and the violations against the rules: reports, browsers, and diagrams. Do not mix too much types of information into one view.
5) *Check on inconsistencies in the architecture definition*, and inform the user when it is incorrect or incomplete.

Not all issues identified in this study can be solved easily. The provision of SRMA support calls for further research. Techniques need to be identified, and support needs to be designed and tested on effectiveness by means of prototypes and case studies. Specific topics deserve attention too. For instance, visualization, rule definition and rule checking appeared to be a challenging combination. Furthermore, automatic recognition of responsibility at code level, needed to check against the defined responsibility of a module, is an unresolved issue, though responsibility is an important property of a module at design level.

In conclusion, the eight tested tools provide useful support for ACC, but all could improve their support of SRMAs. Solutions need to be found to reduce the gap between documented modular architectures in software architecture documents on one side, and module and rule models in ACC-tools on the other side. More-complete functional support of SRMAs might contribute to the adoption of ACC and ACC-tools, and consequently could improve the effectiveness of software architecture in the practice and education of software engineering.

ACKNOWLEDGMENT

The authors would like to thank the students of the specialization "Advanced Software Engineering" at the HU University of Applied Sciences, but also colleagues and reviewers for their contributions to this study.

REFERENCES

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

[2] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, no. 1, pp. 132–151, Jan. 2012.

[3] 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.

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

[5] M. Shaw and P. Clements, "The golden age of software architecture," *IEEE Software*, vol. 23, no. 2, pp. 31–39, 2006.

[6] M. Gleirscher and D. Golubitskiy, "On the Benefit of Automated Static Analysis for Small and Medium-Sized Software Enterprises," *Software Quality. Process Automation In Software Development*, 2012.

[7] G. Canfora, M. Di Penta, and L. Cerulo, "Achievements and challenges in software reverse engineering," *Communications of the ACM*, vol. 54, no. 4, p. 142, Apr. 2011.

[8] L. Passos, R. Terra, M. T. Valente, R. Diniz, and N. Das Chagas Mendonca, "Static Architecture-Conformance Checking: An Illustrative Overview," *IEEE Software*, vol. 27, no. 5, pp. 82–89, 2010.

[9] R. Kazman, L. Bass, and M. Klein, "The essential components of software architecture design and analysis," *Journal of Systems and Software*, vol. 79, no. 8, pp. 1207–1216, 2006.

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

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

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

[13] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Third Edit. Addison-Wesley, 2012.

[14] 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.

[15] E. W. Dijkstra, "The structure of the 'THE'-multiprogramming system," *Communications of the ACM*, vol. 11, no. 5, pp. 341–346, 1968.

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

[17] N. B. Harrison and P. Avgeriou, "Analysis of Architecture Pattern Usage in Legacy System Architecture Documentation," in *Seventh Working IEEE/IFIP Conference on Software Architecture*, 2008, pp. 147–156.

[18] C. Larman, *Applying UML And Patterns*. Prentice Hall PTR, 2005.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vissedes, *Design Patterns: Elements of Reusable Object-Oriented Software (Google eBook)*. Pearson Education, 1995.

[20] N. Ali, J. Rosik, and J. Buckley, "Characterizing real-time reflexion-based architecture recovery: an in-vivo multi-case study," *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures*, pp. 23–32, 2012.

[21] E. Woods and N. Rozanski, "Unifying software architecture with its implementation," in *Proceedings of the Fourth European Conference on Software Architecture*, 2010, pp. 55-58.

[22] R. Terra and M. Valente, "A dependency constraint language to manage object □oriented software architectures," *Software: Practice and Experience*, no. June, pp. 1073–1094, 2009.

[23] S. Sarkar, G. Rama, and R. Shubha, "A method for detecting and measuring architectural layering violations in source code," in *APSEC*, 2006.

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