A Comparison of Static Architecture Compliance Checking Approaches[1]

Jens Knodel

Fraunhofer Institute for Experimental Software Engineering (IESE), Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany
knodel@iese.fraunhofer.de

Daniel Popescu

Computer Science Department University of Southern California Los Angeles, CA 90089, USA
dpopescu@usc.edu

Abstract

The software architecture is one of the most important artifacts created in the lifecycle of a software system. It enables, facilitates, hampers, or interferes directly the achievement of business goals, functional and quality requirements. One instrument to determine how adequate the architecture is for its intended usage is architecture compliance checking. This paper compares three static architecture compliance checking approaches (reflexion models, relation conformance rules, and component access rules) by assessing their applicability in 13 distinct dimensions. The results give guidance on when to use which approach.

Keywords: access rules, architecture compliance checking, architecture evaluation, conformance rules, SAVE, software architecture, static analysis.

1. Introduction

Software development organizations must address challenges such as faster reaction to market needs, increasing complexity of the system, and frequent changes of the requirements. The software architecture is one of the most crucial artifacts within the lifecycle of a software system. Decisions made at the architectural level have a direct impact on the achievement of business goals, functional and quality requirements [5].

However, once the architecture has been designed and documented, our experiences with industrial partners show that architectural descriptions are seldom maintained. On the contrary, architecture descriptions (if available at all) are often insufficient and/or incomplete [4]. Late changes during implementation, conflicting quality requirements only resolved while implementing the system and technical development constraints often results in the fact that the implemented architecture is not compliant to the planned, intended architecture, even for the first release. During the evolution of a system such effects intensify and the planned and the implemented architecture diverge even more.

A promising approach avoiding such architectural decay is to encode architectural elements into the implementation language. If programming languages provide mechanisms to denote architectural elements like components or connectors (e.g., [2], [14]), they can be more easily kept consistent with the code, and since the mechanisms are part of the code, changes in the code results in changed architectural elements. For example, a design can demand that several components belong to a certain layer of a layered architecture, but most of current programming languages (e.g., C/C++, Java) do not support such mechanisms. So typically developers are not explicitly made aware of architectural decisions while implementing the solutions. Thus, changes or implementation decisions that affect the planned architecture might not be recorded appropriately. The introduction of new components bears the risk of introducing architectural violations. Other potential architectural threats emerge when reusing existing software, in particular when the candidates have not been developed by the same organization or the original developers are no longer available. The case study of Garlan and Ockerbloom [9] describe a number of issues (e.g., excessive code, poor performance, need to modify external packages, need to reinvent existing functionality, unnecessarily complicated tools) caused by architectural mismatch that hampered successful reuse. Each of these issues requires modifications of the software with the risk of deviating from the planned architecture. Thus, there is a high demand for assurance of architectural compliance.

We define architecture compliance as a measure to which degree the implemented architecture in the source code conforms to the planned architecture (i.e., a compliance of 1.0 or 100% means that there are no architectural violations, 0.0 or 0% the opposite). Architecture compliance checking is the means to

[1] This work has been partially funded by the ArQuE project (Architecture-centric Quality Engineering) funded by the German federal ministry of education and research (BMBF) under Förderkennzeichen: 01 IS F14.

measure this. The calculation divides the number of compliant dependencies by the total number of dependencies between components.

The compliance of the architecture can be checked sstatically (i.e., without executing the code) and dynamically (at run-time). In this paper, we identified and compare the three main static architecture compliance checking approaches of a system:

- **Reflexion models**: Reflexion models compare two models of a software system against each other, typically, an architectural model (the planned or intended architecture) and a source code model (the actual or implemented architecture). The comparison requires a mapping between the two models to be compared, which is a human-based task (see [17], [13], [6], [12] for related work on Reflexion models).
- **Relation Conformance Rules**: Relation conformance rules enable specifying allowed or forbidden relations between two components. They can detect similar defects as reflexion models, but the mapping is done automatically for each conformance check (see [18], [20], [10] for related work on relation conformance rules).
- **Component Access Rules**: Component access rules enable specifying simple ports for components, which other components are allowed to call. These rules help to increase the information hiding of components on a level, which might not be possible within the physical architecture of the implementation language (Component Access Rules were inspired by ports in ADL [16] and exported packages in OSGi [19]).

Static architecture evaluation of either one of the three approaches results in the assignment of one of the following types to each relation between two components:

- Convergence – a relation between two components that is allowed or was implemented as intended. Convergences indicate that the implementation is compliant to the planned architecture.
- Divergence – a relation between two components that is not allowed or was not implemented as intended. Divergences indicate that the implementation is not compliant to the planned architecture.
- Absence – a relation between two components that was intended but not implemented. Absences indicate that the relation in the planned architecture were not found in the implementation (please note that evaluation of component access rules can not result in absences).

In this paper we will compare the three static architecture evaluation approaches and highlight the commonalities, differences, advantages and drawbacks. We implemented all three approaches as features of the SAVE tool (Software Architecture Visualization and Evaluation, see [12]). All three evaluation implementations use the visualization of SAVE and operate on the same data model (see Section 2). Section 3 explains each of the approaches, while Section 4 illustrates each of the architecture compliance checking approaches by means of an evaluation of the TSAFE. TSAFE is a software system that aids air-traffic controllers in detecting and resolving short-term conflicts between aircrafts and was turned into a testbed for experimentation by Fraunhofer Center for Experimental Software Engineering Maryland (FC-MD) [15]. In Section 5, we will then compare the three approaches, while Section 6 concludes this paper.

2. The SAVE Tool

SAVE (Software Architecture Visualization and Evaluation) is an Eclipse plug-in developed at Fraunhofer IESE for static evaluations of software architectures. The SAVE tool has a powerful engine for visualization of software architectures offering a large number of state-of-the-art graphical elements. A main feature is the configurability of the visualization (i.e., enabling and/or disabling certain graphical elements) allowing users to adapt and customize the visualization of results to their own needs. All evaluation screenshots have been produced with SAVE.

2.1. Data Model

The SAVE data model consists of components and relations. Components can contain subcomponents supporting a hierarchical decomposition of a software system. They describe computation functionality or data store. Relations in the data model indicate a dependency between two components, and thus cannot exist without a component.

A fact extractor identifies the components and their relations using a static analysis of the source code. Components are created according to different extraction strategies. For instance, in Java the component hierarchy is extracted from the package hierarchy. A class can be a component on its own or its elements can be assigned to its directory. In the procedural language C the component hierarchy is extracted from the directory structure. As in Java, an analyst can choose granularity at the directory or file level.

Components and relations maintain information about the source entity from which they were extracted. Therefore, every component can be traced back to its source file supporting detailed analysis and more rigorous evaluation. The same holds for relations: relations are lifted dependencies present in the source code. We

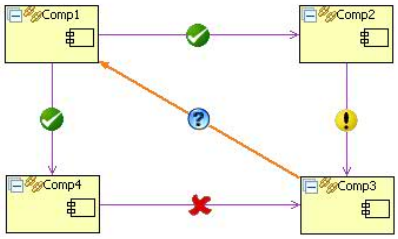

Figure 1: Decoration Icons for Evaluation Results

distinguish between the following types of code dependencies: function calls, direct variable accesses or import statements. Hence, a single relation between two components in the SAVE data model can be an abstraction of hundreds of code dependencies in the source code. Different types of relations can be comprised as an aggregated relation.

The SAVE data model does not support the concept of connectors, but complex connectors can be synthesized into connector components.

2.2. Evaluation Results Visualization

The architecture evaluation results are visualized in SAVE with the help of decoration icons that are placed on top of the relations. There exist four types of decoration icons:

- **Green check mark**: A convergence between two components is decorated with a green check mark.
- **Black exclamation mark**: A divergence relation between two components is decorated with an exclamation mark.
- **Red cross**: Absences are decorated with a red cross.
- **Blue question mark**: Aggregated relations comprising more than one of the above types are denoted with a blue question mark.

3. Architecture Compliance Checking

This section explains the three architecture compliance checking approaches in detail.

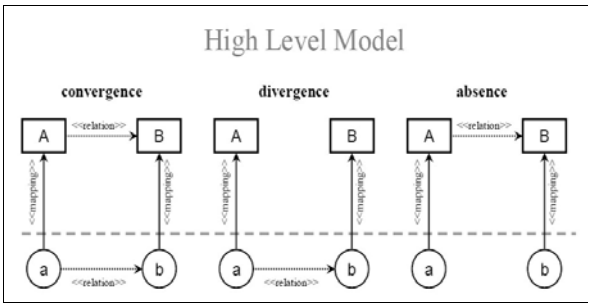

Figure 2: Principle of Reflexion models

3.1. Reflexion Model

The idea of reflexion models was introduced by Murphy [17] and then extended by Koschke [13] to support hierarchies.

Figure 2 depicts the main principle of reflexion models. High-level components are mapped onto source code elements provided in a low-level source code model by human-based mapping. Typically, these mapping are based on regular expressions and a high-level component comprises a high number of source code elements. In addition, the high-level model contains the planned or intended relations among components. At evaluation time, the mappings are resolved and the low level relations among source code elements are lifted to the high-level model. If there is a match of a high-level relation and a lifted low-level relations, then this relation is marked as a convergence, if not it is either a divergence (only a lifted low-level relation present) or an absence (no low-level relation present). Human experts map manually high-level components to source code elements. Recently Christl, Koschke, and Storey proposed a new mapping approach that derives semi-automatic mappings based on similarity clustering of source code files to reduce the expert effort in the mapping step [6].

3.2. Relation Conformance Rules

A relation conformance rule defines an allowed or a forbidden relation between two components. In contrast to the reflexion models, leaf level relations of models can be checked without defining the super components of a leaf component. One relation conformance rule can cover multiple relations of different components. Additionally, relation conformance rules can be used to specify allowed or forbidden connections to the context of an analyzed system like third party libraries. A relation conformance rule contains a relation type, a source component, a target component and a relation rule type. For both, the source and target components of the rule, a regular expression must be defined, which matches the name of components. The relation conformance rule type defines either that a connection is forbidden or that a connection must exist between the source component and the target component. The relation type defines if the connection is a general dependency relation, or for example a call, import or access relation.

For example, the relation rule "A* must exist B*" defines that a relation must exist from components, which name starts with an A, to components, which name starts with a B. Further, the relation rule "C* is forbidden D*" defines that the components, which name starts with an C, are not allowed to have outgoing relations to components having a name starting with D.

3.3. Component Access Rules

Component access rules define simple ports of a component, which other components are allowed to call. They help increase information hiding of components on a level, which might not be supported by the implementation language itself. In Java, for example, declaring methods as public within a component is necessary for invoking such methods from other classes and packages located in the same component. All these public methods are also accessible from other components, although typically not all of them were intended to serve as an interface or a port of the component to the outside. The implementation language does not guarantee that entities outside of the component boundary do not call the public methods within a component.

Thus, component access rules encapsulate components and they allow opening only certain ports for component interaction. While relation conformance rules specify a relation between two components, a component access rule concerns only one component.

4. TSAFE Case Study

TSAFE is a prototype of the Tactical Separation Assisted Flight Environment specified by NASA Ames Research Center [7] and implemented at MIT [8]. TSAFE was proposed as a principal part of a larger Automated Airspace Computing system that shifts the burden from human controllers to computers. TSAFE was turned into a testbed by FC-MD to be used for software technology experimentation [15]. The TSAFE prototype checks conformance of aircraft flights to their flight plans, predicts future trajectories, and displays results on a geographical map. We statically evaluated two variants of TSAFE, the original prototype called TSAFE I and a redesigned version TSAFE II. Both variants have different structures but are indistinguishable at the level of their external behavior and identical GUI. The redesign was performed by the FC-MD because they discovered several critical issues affecting the flexibility of TSAFE towards new features. In order to reduce future maintenance effort, they analyzed and resolved quality issues, furthermore they compared the original with the new design in order to evaluate the effect on maintainability and flexibility (please refer to [1] for a detailed description of the redesign).

We illustrate the three architectural compliance checking approaches by the TSAFE case study. The work on TSAFE ([1], [8]) provided us with the necessary information about architectural models, source code, and rules. Furthermore, we interviewed the architects and developers who conducted the redesign for TSAFE II to obtain additional information on relation conformance and component access rules. Hence, we played only the role of analysts in the case study.

We started with the evaluation by applying the reflexion model approach to TSAFE I (see figure 3). The architectural model and the mapping were provided in the documentation of the system. Figure 1 depicts the results. The architecture compliance degree was 88.73% (i.e., the number of convergences divided by the total number of relations). The main divergences were introduced by the *feed* subsystem, violating the architecture by using *engine* subsystem and being used by the *main* subsystem. Additionally, one relation from the *engine* to the *client* subsystem is absent.

As mentioned above TSAFE II (see figure 4) underwent a major restructuring, the architectural style changed to a real client-server architecture. As captured in Figure 4, the implementation is 100% compliant to the architectural style. The main reason for this was the point in time: the evaluation took place directly after the restructuring was completed. Hence, the system did not

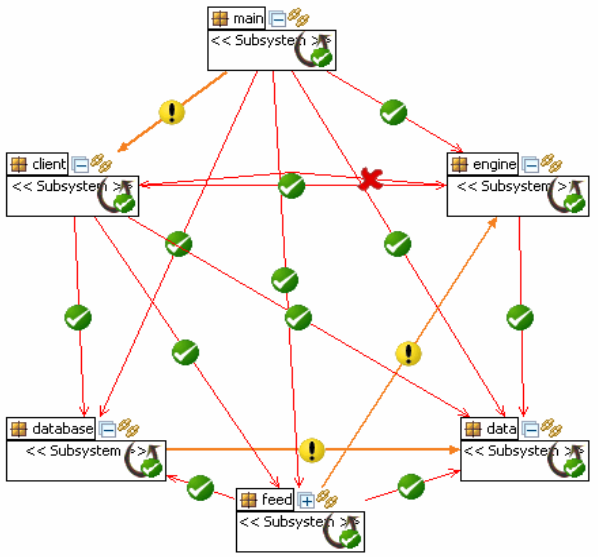

Figure 3: Reflexion Model Evaluation of TSAFE I

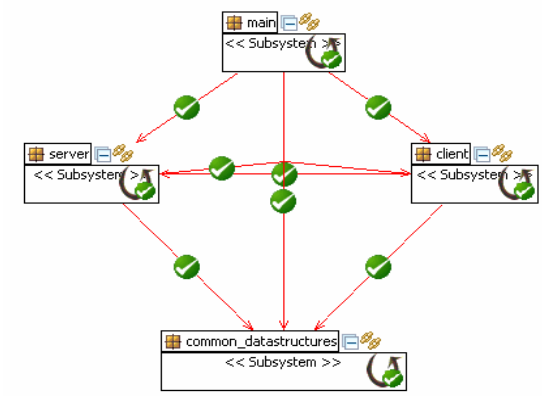

Figure 4: Reflexion Model Evaluation of TSAFE II

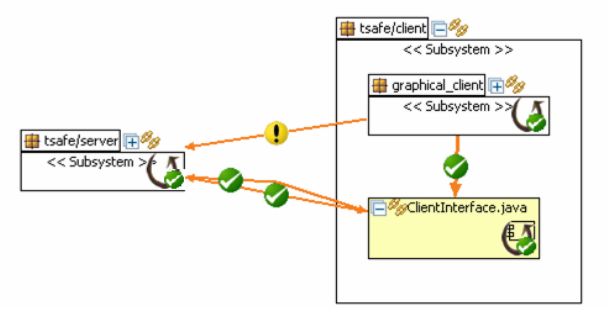

Figure 5: Relation Conformance Rules Evaluation for Client Server Dependencies in TSAFE II

yet evolve further. Nevertheless, Figure 4 lacks one important piece of information: part of the redesign was the introduction of interface classes for each subsystem and components being responsible for the communication with other components.

For instance, the architects stated the following relation conformance rules expressing the client-server dependency: "No classes but the *ServerInterface* class shall access classes in the *client* subsystem" and "No classes other than the *ClientInterface* class shall access classes form the *server* subsystem". The evaluation based on the reflexion model (see Figure 4) does not detect this divergence because lack of information regarding what caused the relations between the subsystems *server* and *client*.

We applied the relation conformance rules to evaluate the two rules mentioned above as well as 20 other similar rules. Figure 5 shows the results of the relation conformance rules evaluation (the figure just depicts an excerpt of TSAFE II) addressing the client server communication. The resulting graph revealed that the subsystem *graphical_client* violates the rule since it has dependencies to the *server* subsystem. The analysis revealed that the subsystem *graphical client* communicates with the server via the *ClientInterface*, but there are also 4 divergences where the interface was not used as intended.

One subsystem contained in the server is the *parser* subsystem. The architects of TSAFE II stated a components access rule for this subsystem: "Only the *ParserInterface* class shall be accessed from any other component". This component's access rule concerns only the *parser* component and no other component so that relation conformance rules are not applicable. The evaluation result of this rule is depicted in Figure 6. The subsystem *server_gui* calls methods contained in classes of the *asdi* subsystem, which violates the component access rule.

Recapitulating, this section reported on a case study where we applied the three architecture compliance approaches. The TSAFE example illustrated the main needs and benefits of each of the approaches: reflexion models allows architects to reflect and refine their views (either documented or mental models) on the structural decomposition of the system, while relation conformance rules enable independent checking of an architectural model with the resolution of the rules at evaluation time. Relation conformance rules allow a fine-grained analysis of the dependencies within a system. Component access rules ensure that a component is used in the intended way, independent from the rest of the system since they are just defined within the scope of single components.

In the next section we will compare the techniques against each other, and partially make use of the examples to denote the comparison criteria.

5. Comparison

5.1. Comparison Dimensions

The main goal of architecture compliance checking is to detect spots in a given system that violate its

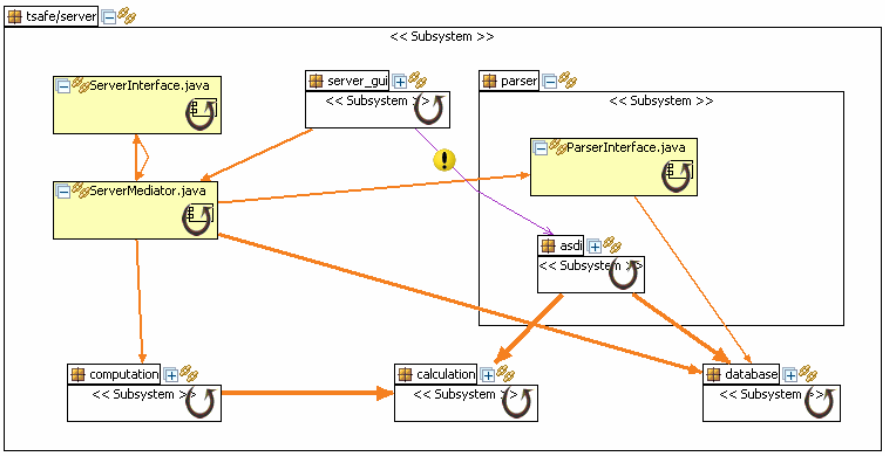

Figure 6: Component Access Rule Evaluation for Parser in TSAFE II

architecture. The detection of such spots enables the architects to plan, and track counteractive measures. Guided by the GQM approach (Goal-Question-Metrics [3]) we derived criteria dimensions for the comparison of the three architecture compliance checking approaches. The dimensions have then been reviewed and refined by two other members of our group. In total, we defined 13 dimensions:

Input: the input dimension describes the artifacts that are necessary in order to apply the approach.

Involved Stakeholders: stakeholders are the persons or roles that are involved in the evaluation and their tasks.

Manual Steps: describes the main activities or steps the analyst has to carry out with the involved stakeholders but are not (semi)-automated by tools. The automated tasks, such as fact extraction or computation of the results, are ignored here since all are automated by the SAVE tool.

Evaluation Performance: the time required to compute the evaluation results.

Defect Types: the defect types describe classes of defects that can be detected by the evaluation approach. In particular, we consider the classes of architecture violations: unintended dependencies, context exploration (the context captures libraries and components where the source code is not available during analysis time), broken information hiding, misusage of interfaces, misuse of patterns, or violation of architectural styles.

Probability of False Positives: the probability dimension captures the likelihood for false positives in the analysis results measure on an ordinal scale with the values low, medium and high.

Maintainability: the maintainability dimension captures the robustness of the architecture compliance checking approach with respect to code evolution. In particular, we consider the addition, modification or removal of components or code entities. Changes to components hereby means a major change that result in a high number of code entity changes, which are considered as having a rather limited impact.

Transferability: the transferability dimension describes how the work products created in the compliance checking approach can be applied or reused when evaluating another version of the system, a distinct variant of the system, the system after major restructurings, or a different system. The first two are closely related to the system (i.e., mostly the same components, the same architecture), while the third uses similar components but has a different architecture, and the fourth is a completely different system with limited reuse.

Scalability: scalability captures to which degree the approach scales up to handle large systems.

Ease of Application: the ease of application captures our subjective experiences on three levels: intuitiveness, iterations, and learning curve. We rate intuitiveness (how easy and intuitive an analyst can apply the approach) on an ordinal scale with the values low, medium and high. The learning curve captures our subjective experiences on how much training is needed for a new analyst to be able to conduct an evaluation and to produce useful results. This dimension ranges from high (few training required) to low (a lot of training required). The scale for iterative refinements is yes, limited, and no.

Multiple View Support: typically, there is not only one view of the static structure of the system; the architects often have different views, sometimes overlapping or even conflicting. This dimension captures how the approach is able to handle such multiple views and how easy it is to find out about the commonalities and variabilities in the created work products.

Restructuring Scenarios: restructuring scenarios capture the exploration of what-if analyses of the actual system, how would the architecture compliance change for different structural decompositions.

Architectural Decision Support: this dimension captures how the architecture compliance checking approach supports decision making, trade-off analyses or scenario-specific analyses when reasoning about architectural or non-functional qualities. This dimension ranges from almost no, limited, medium to high support.

5.2. Comparison Results

Figure 7 shows the comparison results for the 13 criteria as defined above for the three architecture compliance checking approaches. Bullets in parenthesis indicate optional items or activities. Rows with only one entry for all approaches indicate a commonality among them. The table assumes the general case of the application of the compliance checking approaches, however there might be exception where cells of the table might be different. We derived the table based on our practical experiences gained in several industrial and academic case studies.

The main **input** for all three approaches is obviously the source code, for the reflexion model an architectural model (usually obtained from architecture documentation) is mandatory. Component access rules can, in addition, be based on component specifications or interface documentation. The main **stakeholder** involved in defining the access rules is the component engineer. For the other two approaches, the architect has the main responsibility; all three may involve developers in order to obtain detailed information about the source code.

Section 3 already explained the **manual steps** of the three approaches were in detail. The automation in our case provided by the SAVE tool are the same for all three approaches, they involve the fact extraction, abstraction, visualization (e.g., layouting) and the support for results navigation (e.g., filtering).

Approach / Dimension	Reflexion Model	Relation Conformance Rules	Component Access Rules
Input	• source code • architecture documentation	• source code • (architecture documentation)	• source code • (component specification) • (interface documentation) • (architecture documentation)
Involved Stakeholders	• architect • (developer)	• architect • (developer)	• (architect) • Component engineer • (developer)
Manual Steps	• creating of the architectural model • mapping of architectural model elements to code • reviewing results	• creating and formalizing of relation conformance rules • reviewing results	• creating and formalizing of component access rules • reviewing results
Evaluation Performance	equivalent		
Defect Types	• unintended dependencies • misusage of interfaces • (misuse of patterns) • violation of architectural styles	• unintended dependencies • context exploration • misusage of interfaces • (misuse of patterns) • violation of architectural styles	• unintended dependencies • broken information hiding • misusage of interfaces
Probability of False Positives	• low	• high	• low
Maintainability	• added components: refinement of architectural model and mapping necessary • modified components: refinement of mapping might be necessary • removed components: refinement of architectural model necessary • added code entities: no consequences • modified code entities: no consequences • removed code entities: no consequences	• added components: refinement might be necessary • modified components: no consequences • removed components: no consequences • added code entities: no consequences • modified code entities: no consequences • removed code entities: no consequences	• added components: new component access rules necessary • modified components: refinement might be necessary • removed components: no consequences • added code entities: component access rule refinement might be necessary • modified code entities: component access rule refinement might be necessary • removed code entities: component access rule refinement might be necessary
Transferability	• version: rework mapping • variant: partial refinement of architectural model and rework mapping and • major restructuring: rework • different system: not possible	• version: no consequences • variant: no consequences • major restructuring: review rules • different system: yes, if naming conventions are applied	• version: no consequences • variant: no consequences • major restructuring: review rules • different system: yes, if component is reused
Scalability	equivalent		
Ease of Application	• intuitiveness: high • iterative refinements: yes • learning curve: high	• intuitiveness: high • iterative refinements: limited • learning curve: medium	• intuitiveness: medium • iterative refinements: limited • learning curve: medium

Approach / Dimension	Reflexion Model	Relation Conformance Rules	Component Access Rules
Multiple View Support	• yes, each model mapping pair can capture a different view. • it is possible to have mapping that crosscut the decomposition hierarchy • commonalities, overlaps, and variation can be achieved by comparing the mappings	• yes, each rule set can capture a different view • no hierarchy crosscutting possible since the rules depend on the hierarchy	• no, multiple views are not supported • no hierarchy crosscutting possible since the rules depend on the hierarchy
Restructuring Scenarios	• restructuring scenarios supporting by modifications to model and mapping pairs • monitoring and tracking possible	• no support for what-if analyses • monitoring and tracking possible	• no support for what-if analyses • no support for monitoring and tracking of restructurings
Architectural Decision Support	• limited		

Figure 7: Comparison Table

The **evaluation performance** is obviously dependent on the implementation and on the number of elements and dependencies that have to be checked. We conducted a series of evaluations with different systems (up to 500 KLoC) and varying configurations (mappings, number of rules, etc.) on a typical computer (processor 1,2 GHz, 1 GB RAM, Microsoft Windows XP Professional, Eclipse 3.1.2). The time required to compute the evaluation results for the three approaches was less than 5 minute for all systems, other steps in the analysis like fact extraction and visualization of results took significant more time, so we consider all approaches as equivalent in this dimension.

All three approaches are of course able to identify the **defect type** unintended dependencies. Misuse of interfaces is detectable by all three, but the reflexion model approach requires the definition of special interfaces component that act as a broker between the actual component and rest of the system. To detect the misuse of design patterns statically is only possible in special cases since pattern instances can be ambiguous. By their nature component access rules are able to detect broken information hiding since this is the main application of such rules. Both reflexion models and relation conformance rules could be used to identify violations of architectural styles [21] (e.g., layered architecture, blackboard, client sever) assuming that the architects provide the necessary information for formalizing the style. Relation conformance rules are the only technique that is able to explore the context. Context exploration thereby means that it is possible to define rules that take the context (i.e., libraries, third party software, components off the shelf where the source code is not available) into account. The relation rules define certain ways on how the dependencies of the system to its context should be or can violate the intended usage. In Java for instance, there are libraries such as JFace and SWT providing elements for graphical user interfaces. A relation conformance rule could forbid the usage of such libraries to all components not related to the user interface of the system under examination.

The **probability of false positives** (e.g., a computed divergence that is actually a convergence or vice versa) for reflexion model is low since the resolution of regular expressions can be reviewed and adjusted by the analyst and architect before the evaluation. It is high for relation conformance rules since the evaluation is only based on regular expressions which are only resolved at evaluation time. Component access rules score low since the interfaces are typically intended to be accessible from all other components and refinements by regular expressions are less likely.

The **maintainability** dimension addresses the ability of the compliance checking approach to evolve with the source code. We distinguish thereby between two levels of evolution: major changes resulting in addition, modification or removal of components, and minor changes affecting only code entities. Relation conformance rules are only affected by the addition of new components since they might have a different naming convention not yet taken into account by the rules. Obviously, when adding new components, new component access rules are required and thus, have to be defined. When modifying components the component access rules have to be reviewed with respect to potential refinements. Changes to components imply for reflexion models that both the architectural model and the mapping have to be at least reviewed and if not updated. Code entity changes did not affect the reflexion models nor the relation conformance rules, but they can affect component access rules, if they accessible entities were changed or entities to be were added.

The **transferability** differs among the compliance checking approaches: component access rules can be

reused whenever the component itself is reused. Reflexion models require at least rework of the mapping, even for variants the architectural model might need a refinement, and it is not possible to transfer work products of reflexion models when evaluating a different system. The restructuring of TSAFE as described in section 4 required rework to both the mapping and the architectural models. Conformance rules allow or forbid certain kinds of relations. The rules can be applied organization-wide if naming conventions are applied consistently. However, their applicability has to be reviewed with respect to their usefulness in the context of a restructured or different system.

The three evaluation approaches use SAVE features as fact extraction and visualization affecting the **scalability** more than the evaluation computation, thus we consider the approaches as equal.

The **ease of application** is a subjective measure based on our experience. We decomposed it into three aspects. We rated intuitiveness of reflexion models and relation conformance rules as high since both can be applied straight-forward without requiring any special training: natural-languages rules can mostly be formalized and architectural model and the mapping can be done easily without any special training based on given documentation. We consider component access rules as medium easy since they require some knowledge of implementation details (i.e., which methods constitute the interface) is required to be able to define the access rules. Reflexion models and conformance rules support incremental refinements well; they allow trials of models and rules. For instance, the application of the two client server rules as applied in Section 4 can be started with one rule only, or reflexion models allow evaluations of the system under analysis with slightly different mappings, so the analyst can start with basic mapping (where the mapping confidence is high, leaving out parts of the system) and expand the mapping over time.

Reflexion models provide good support for **multiple views**. For example, consider a system with layered architecture composed partially of reusable components distributed over the layers. The architects would be interested on the one hand in the compliance of the layering and on the other hand in whether or not the reusable components have dependencies on the system-specific parts. With reflexion models, this is easily checked by creating two distinct architectural models and mapping pairs, each capturing one of the two evaluation scenarios (refer to [11] for an example). Component access rules do not support such multiple views. Relation conformance rules are able to reflect different views by different rules sets but the rule sets are all dependent on the decomposition hierarchy. Thus, views that crosscut hierarchies are indefinable with relation conformance rules.

Restructuring scenarios are supported by reflexion models in two different ways: by creating artificial components being part of the architectural model and by defining a target architecture towards the implementation of the systems should be refactored to. Evaluating the implementation at constant intervals enables monitoring and tracking the progress towards reaching the restructuring goals. Since relation conformance rules and component access rules just operate on one model, they do not support what-if analyses. However, once a target architecture has been established a new set of relation conformance rules can be created to check the compliance of the implementation towards the target and thus, the monitoring and tracking is supported.

The **architectural decision support** offered by each approach itself is rather low. There is no guidance on how to address architectural violations that have been detected. The reasoning of the architects is dependent on their interpretation and decisions are not derived by the evaluation results only but use a significant amount of additional context information and rationales.

5.3. Discussion

As the case study in Section 4 and the comparison in the previous section show each of the three approaches has its benefits and advantages. Common to all approaches are the criteria evaluation performance and scalability, both are highly dependent on the SAVE tool. Since the realization of each technique was conducted within our group (partially by the same developers), the selected algorithms and data structures created for the three approaches are similar to some extent. Thus, we did not expect a major difference here. However, when using other tools, the approaches might rank differently in both dimensions.

Although the three approaches are based on the same principle of statically evaluating the dependencies of a system under investigation, the approaches differ in many ways. The main differences concern the dimensions defect types, maintainability, transferability, multiple view support, and restructuring scenarios. Due to these differences the architects have to decide which alternative fits better to their goals and the application context for architecture compliance checking. The criteria dimensions as defined in Section 5.1 and the result of the comparison in Section 5.2 can serve as the foundation for such decisions.

Thus, we recommend a goal-driven selection approach to decide whether to apply reflexion models, relation conformance rule, component access rules, or a combination of them. The goal-driven selection approach has to be based on the comparison dimensions. Based on their needs in each dimension the architects can decide which compliance checking approach fits best or which

constraints exclude one of the approaches. We believe that the comparison presented in this paper is a first step to derive decisions on objectified criteria rather than to just rely on the experiences of the architects and analysts being expert for the approaches. The TSAFE example showed a case where all three approaches have been applied in combination.

6. Conclusion

Architecture compliance checking is a sound instrument to detect architectural violations (i.e., deviations between the intended architecture and the implemented architecture). In this paper we presented three architecture compliance checking approaches (namely reflexion models, relation conformance rules, and component access rules) that evaluate architectures statically. Based on 13 criteria dimensions, we assessed the applicability and usefulness of each approach supporting the goal-driven selection of architects on which approach to apply.

We are continuously seeking new approaches that check the architectural compliance statically. Furthermore, we are planning to extend the comparison towards dynamic architecture compliance checking, which might lead to a refinement or extension of the comparison dimensions.

In ongoing and future projects we will statically evaluate the given architectures by either one of the three approaches or a combination of them. So far we applied just one out of the three approaches. We plan to analyze in more detail the benefits of combinations and we want to continue our work towards a systematic, customizable selection approach for architecture compliance checking.

References

[1] Ackermann, C., Lindvall, M. Dennis, G., (2006) “Redesign for Flexibility and Maintainability: A Case Study “, Technical Report, Fraunhofer Center for Experimental Software Engineering Maryland

[2] Aldrich, J.; Chambers, C. & Notkin, D. (2002), “ArchJava: connecting software architecture to implementation”, Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA, 187-197.

[3] Basili, Victor R.; Caldiera, Gianluigi; Rombach, H. Dieter, (1994) “The Goal Question Metric Paradigm”, Encyclopedia of Software Engineering (Marciniak, J.J., editor), Volume 1, John Wiley & Sons, pp. 578-583

[4] Bayer, J., (2004): “View-Based Software Documentation”, PhD, Fraunhofer IRB Verlag.

[5] Bosch, J. (2000), “Design and Use of Software Architectures”, Addison-Wesley Professional.

[6] Christl, A., Koschke, R., & Storey, M.-A. (2005). Equipping the Reflexion Method with Automated Clustering. Working Conference on Reverse Engineering, Pittsburgh, USA.

[7] Erzberger H. (2001), “The automated airspace concept”. 4th USA/Europe Air Traffic Management R&D Seminar.

[8] Dennis G., (2003), "TSAFE: Building a Trusted Computing Base for Air Traffic Control Software." Masters Thesis MIT.

[9] Garlan, D. A. & Ockerbloom, J. R., (1995) “Architectural mismatch: Why reuse is so hard”. IEEE Software, 12(6):17–26, Nov 1995.

[10] Holt, R.C., (2005), “Permission Theory for Software Architecture Plus Phantom Architectures”, School of Computer Science, University of Waterloo, September 2005.

[11] Kolb, R., John, I., Knodel, J., Muthig, D., Haury, U., & Meier, G. (2006). Experiences with Product Line Development of Embedded Systems at Testo AG. Submitted to: The 10th International Software Product Line Conference (SPLC 2006), Baltimore, USA

[12] Knodel, J., Lindvall, M., Muthig, D., & Naab, M. (2006). Static Evaluation of Software Architectures. 10th European Conference on Software Maintenance and Reengineering, Bari, Italy.

[13] Koschke, R. & Simon, D. (2003), “Hierarchical Reflexion Models”, 10th Working Conference on Reverse Engineering (WCRE 2003), 13-16 November 2003, Victoria, Canada, 36-45.

[14] Lam, P. & Rinard, M. (2003), “A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information” Proceedings of the 17th European Conference on Object-Oriented Programming, pages 275--302, July 2003

[15] Lindvall M., Rus I., Shull F., Zelkowitz M. V., Donzelli P., Memon A., Basili V. R., Costa P., Tvedt R. T., Hochstein L., Asgari S., Ackermann C., and Pech D., "An Evolutionary Testbed for Software Technology Evaluation," Innovations in Systems and Software Engineering - a NASA Journal, vol. 1, no. 1, pp. 3-11, 2005

[16] Medvidovic, N. Taylor, R. N. (2000) "A Classification and Comparison Framework for Software Architecture Description Languages," IEEE Transactions on Software Engineering, vol. 26, no. 1, pp. 70-93, Jan., 2000.

[17] Murphy, G.C.; Notkin, D. & Sullivan, K.J. (2001), “Software Reflexion Models: Bridging the Gap between Design and Implementation”, IEEE Trans. Software Eng. 27(4), 364-380.

[18] van Ommering, R., Krikhaar, R., Feijs, L. (2001), “Languages for formalizing, visualizing and verifying software architectures”, Computer Languages, Volume 27, (1/3): 3-18

[19] OSGi: Open Services Gateway Initiative (2006), http://www.osgi.org/

[20] Postma, A. (2003), “A method for module architecture verification and its application on a large component-based system”, Information & Software Technology 45(4), 171-194.

[21] Shaw, M. Garlan, D. (1996), “Software Architecture: Perspectives on an Emerging Discipline”, Prentice Hall