Automatic Reference Architecture Conformance Checking for SOA-based Software Systems

Rainer Weinreich
Johannes Kepler University Linz
Linz, Austria
rainer.weinreich@jku.at

Georg Buchgeher
Software Competence Center Hagenberg
Linz, Austria
georg.buchgeher@scch.at

***Abstract*—Company-wide reference architectures are an important means for standardization and reuse. Standardization is enforced through reference architecture conformance checking. Manual conformance checking is too time- and resource-intensive to be performed continuously for the various systems that are part of a SOA. We present an approach for automatic reference architecture conformance checking of SOA-based software systems. Reference architectures are defined based on rules consisting of roles and of constraints on roles and role relationships. By mapping the roles to the elements of a software architecture representation, reference architecture specifications are reusable for different software systems. Through automating the whole checking process, including architecture extraction, role mapping, and rule evaluation, the approach can be applied continuously for the different systems that are part of a SOA. The approach has been developed and refined by applying it to a SOA in the banking domain. During its evolution from semi-automatic to a fully automatic approach it has also changed the way of how architecture information is provided as part of the system implementation in this domain.**

***Keywords*—software architecture; reference architecture; conformance checking; service-oriented architecture (SOA);**

I. Introduction

Multiple definitions have been provided for the term reference architecture both in the context of software architecture [1][2][3][4] and beyond [5][6]. While the perspective on what constitutes a reference architecture and how it is represented still varies quite a bit [6], there is a consensus on some defining characteristics. First, there is a focus on reuse [6][2]. Reference architectures are defined to be reused for the definition of system architectures. This means they are more generic [3][5] and sometimes at a higher level of abstraction [3] than system architectures. Second, they encode important design decisions for applications in a particular domain. This means they are usually domain-specific [4][2], though this may depend on how one defines a domain [6]. There has also been a discussion of the difference between reference architecture and product line architecture since both are common architectures for applications in a particular domain. According to Nakagawa et al. [2] the difference is mainly the degree of generality. Reference architectures address a broader domain than product-line architectures, which are addressing a specific subset or family of systems within a domain.

Since reference architectures are generic reusable architectures, they have similar roles as software architecture in general, but a different scope of application. In general, software architecture is a means for risk reduction and quality control, for communication among stakeholders, and a blueprint for the system implementation [1]. Since reference architectures are reusable and typically proven architectures for a particular domain, they already encode design decisions for important system qualities in this domain. For this reason, adhering to a reference architecture is already a means for risk mitigation when designing applications in this domain. Also, reference architectures are blueprints for application architectures in a domain and provide guidance for defining concrete application architectures. Greefhorst and Proper [5] identify three important roles of enterprise architecture, which – from our perspective – also apply to reference architectures: regulative, instructive and informative. In its regulative role, a reference architecture is a prescriptive and normative concept, which restricts the design space of the systems conforming to it. It states how systems must become, instead of how they are. In its instructive role, it provides guidance (i.e., instructions) on how to actually design a system based on a reference architecture. Finally, the informative role of an architecture focuses on enabling decision making by sharing architectural knowledge and rationale. This is the communicative role of architecture. A reference architecture can take all three roles, though it depends on how it is actually represented.

In this paper we present an approach which focuses on supporting the regulative role of reference architectures within service-oriented systems in the banking domain. The main aim is to support software architects in continuous conformance checking of the various systems of a SOA landscape to company-wide reference architectures. To support such a continuous conformance checking for different systems within a SOA the whole process of mapping a reference architecture to a system architecture and checking conformance to the defined rules has been completely automated in our approach.

Main contributions of the presented approach are a model and tool for specifying reusable reference architecture compliance rules, and support for automatically binding and evaluating these rules to/for specific application architectures. An additional important aspect – aside from reusability and automation – is the usability of the approach. This is supported by providing an integrated high-level user interface for the definition of reference architecture rules. The approach is based on previous work on architecture extraction and review support [7]. An overview of an earlier version of the approach, which at the time provided support for semi-automatic conformance checking only, can be found in [8].

978-1-4799-3412-6/14 $31.00 © 2014 IEEE
DOI 10.1109/WICSA.2014.22

The remainder of this paper is organized as follows: In Section II we provide information on previous work, which provides the technical basis, the context, and also the motivation for the development of the described approach. In Section III we describe basic concepts for defining reference architecture rules. In Section IV we give an overview of the main steps involved in conformance checking and describe how these steps are supported in our approach. In Section V we provide information on the application domain, i.e., enterprise applications and services for financial institutions. Section VI provides an overview of tool support for the approach. In Section VII we reflect on the evolution and validation of the approach using action research. In Section VIII we summarize experiences with the approach and outline future work. Related work is discussed in Section IX. A conclusion is presented in Section X.

II. Background

As part of our previous work we have developed an approach for extracting and visualizing architectures of SOA-based software systems [7]. The extracted architecture models can be used for various activities including different kinds of architecture analysis, system design and evolution, SOA governance, and quality control. Extracting the architecture from an existing implementation provides a formalized architecture description that is up-to-date. In our case, the extracted architecture model is an instance of the LISA architecture meta-model, or LISA model for short [9]. The LISA model provides several concepts for describing static architecture structures on different levels of abstraction. Examples are object-oriented concepts like *class*, *interface* and *operation* on a lower abstraction level (code model) and *module, component, configuration*, and *service* on a higher level of abstraction (design model). The provided concepts are similar to what is provided by typical architecture description languages (ADLs) and other architecture meta-models (including UML). The LISA model has been designed to support architecture extraction for heterogeneous distributed systems and provides additional concepts for binding such models to an implementation, for distributing architecture models, for architecture knowledge management, and for variability management. LISA architecture models currently can be visualized and analyzed using the LISA toolkit. More information on the LISA model and the LISA toolkit in the context of service-oriented software systems can be found in [7].

In our previous work we have used architecture extraction to support manual reviews of SOA-based software systems in the banking domain [7]. An important part of such reviews is checking the conformance of a solution-architecture to company-wide standards and reference architectures. Originally software architects had to analyze the system implementation manually to perform such checks. This required the analysis of source code and of XML-based configuration files for extracting the actual configuration of a SOA subsystem. In the work described in [7] we already eliminated the need of manually extracting the actually implemented architecture. Software architects are now able to analyze architecture diagrams, which are generated from an extracted architecture model. In the following, we present an approach to further support architects by automating reference architecture conformance checks. The presented work extends our previous work by adding means for specifying reference architectures, for mapping them to specific (extracted) architectures, and for automatically checking the conformance to the specified reference architecture.

III. Basic Concepts

A reference architecture is defined as a reusable set of rules in our approach. Reusability is achieved by defining the rules of a reference architecture specification on the basis of roles. A reference architecture can then be instantiated and checked for a specific software system by assigning the roles of the reference architecture to specific elements of the software system's architecture and by checking whether the defined rules hold for the elements with the assigned roles.

This means that conceptually, each rule of a reference architecture specification consists of a set of *roles*, required and/or forbidden *relationships* between these roles, and *a set of constraints* on both roles and relationships. Constraints are formulated using logic expressions including universal and existence quantifiers on roles and relationships. Constraints can either be bound to a role or to a relationship between roles.

Each role has a name and a multiplicity specification. A multiplicity of [0..*] means that the role may be assigned to zero or more components in an actual architecture model while a multiplicity of [1..1] states that the role can be assigned only once. Roles may have associated properties. These properties state requirements on the elements a role can be assigned to. For example, roles can be assigned to any element of a LISA architecture model including class, interface, component, and service, provided it is able to deliver values for the properties required by a role. The property values are provided by property functions when binding roles to elements of a specific architecture model[1]. Properties can be arbitrary metadata and even runtime data of a component. Examples are the version of a component, its security domain, the associated service layer, or its implementation technology. The following example shows the specification of a role *Data*, which can be assigned to all elements of an architecture model that have been implemented as EJB entity beans.

r: Data [0..] where technology = "ejb3.entity"*

Aside from role specifications, a rule typically consists of additional specifications of relationships between roles. Allowed and forbidden relationships may be specified using the arrow operator. The following example contains both allowed and forbidden relationships. It defines the two roles *Service* and *DataAccess* and specifies that services (i.e., elements with the role *Service*) may use only local data access components (i.e., within the same module). In addition, data access components are not allowed to use service components[2].

c1: Service[0..], c2: DataAccess[0..*],*

[1] As described in the next section, these property functions can retrieve the property values either by looking up meta-data on the elements in an architecture model or by retrieving the required information from other data sources.

[2] A forbidden relationship is specified by using a multiplicity of [0..0]. A forbidden relationship is actually visualized through a crossed-out arrow in the user interface of the rule editor presented in Section VI.

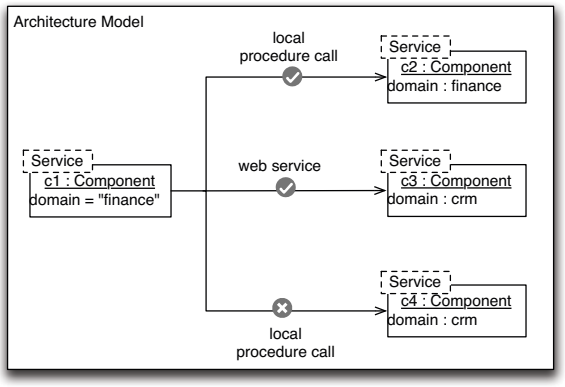

Fig. 1. Role Mapping and Rule Checking Example

c1→c2[0..] where (c1→c2).protocol=”local”,*
c2→c1[0..0]

The kind of relationship this rule applies to in a specific architecture model actually depends on the kind of component a role is (intended to be) assigned to. For example, a relationship between classes and component definitions typically refers to static usage or dependency relationships, while a relationship between component instances and services is a (dynamically) configured communication relationship.

The kind of relationship can be further restricted by assigning properties to a relationship. Properties, like roles, are a general concept. They can be attached to roles and to relationships as well. This is also shown in the example above, where the *protocol* property is required to have a value of ”local” indicating a local connection. In the same way inheritance relationships could be specified as a special case of static dependency relationships. The following example makes use of properties for both roles (property *domain*) and connection (property *protocol*). It specifies that services in different domains should communicate using Web Service protocols:

c1, c2: Service[0..],*
c1→c2[0..] where c1.domain ≠ c2.domain ⇒*
(c1→c2).protocol=”web service”

The rule implicitly specifies a universal quantifier, meaning that for all relations between services, whose domain name is different, the protocol property of the relation needs to provide the value ”web service”, which indicates a Web Service protocol. The actual properties and their potential values depend on the required level of detail for specifying constraints and can be adjusted by providing adequate property functions.

For checking the conformance of a system to a defined reference architecture, roles need to be bound to elements of an architecture model. Roles can be assigned to any element type defined in the LISA meta model (e.g., components, classes, etc). An example for such a role assignment is shown in Figure 1. The figure shows that the role *Service* has been assigned to four components (*c1 – c4*). Component *c1* is connected to the components *c2, c3*, and *c4*. Components *c1* and *c2* communicate through local procedure calls. Since these two components belong to the same domain, this connection is valid. Components *c1* and *c3* belong to different domains. Since they communicate using Web Service protocols this connection is also valid. Finally, the connection between component *c1* and *c4* is invalid because this connection violates the relationship constraint. Component *c1* and *c4* would only be allowed to interact using Web Service protocols (they are located in different domains), but they are connected with a local procedure call connector.

The basic concepts for defining such a reference architecture specification based on rules, roles, and constraints are provided by the LISA constraint model, which is an extension of the LISA architecture model. The resulting reference architecture specification can be stored as part of a LISA architecture module (i.e., a versionable and deployable part of a LISA architecture description) and then be reused for different kinds of architectures by actually importing the module and instantiating the rule set.

IV. Conformance Checking Process

A reference architecture is typically defined only once and can subsequently be reused for checking multiple systems. The basic conformance analysis process is depicted in Figure 2. It is based on a previously defined reference architecture specification, which in our case is a set of rules as described before.

The checking process consists of three mains steps. First, an architecture description needs to be extracted from an existing system implementation. Second, the concepts of the reference architecture need to be mapped onto the extracted architecture. Finally, the rules of the reference architecture can be evaluated with respect to the extracted architecture.

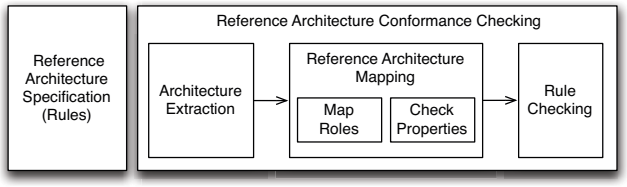

Fig. 2. Reference Architecture Conformance Checking Process

Mapping the concepts of a reference architecture to the elements of the extracted architecture consists of two substeps: assigning roles to architectural elements and assuring that these elements are able to provide values for the properties required by these roles. The latter assures that rules that are using these properties in defined constraints can be evaluated. After these steps, the defined rules can be automatically evaluated based on the actual properties and relationships of the elements in the architecture model.

Assigning roles manually to architectural elements is infeasible for even moderately large software systems. Even smaller subsystems of the SOA we used for validating our approach required hundreds of role assignments to architecture elements. This is why role assignment also needs to be performed automatically. For automatic role assignment we introduced the concept of *role assignment rules* for each role. Each role

can define an assignment rule, which specifies which elements in an architecture model the role is automatically assigned to. Assignment rules use different criteria for assigning roles to elements. Examples are:

- *Element type*: A role is assigned to all architecture elements of a certain type of element in the architecture model. Examples for element types are class, component definition, component instance, and service. This can be used to restrict specific roles to certain element types, e.g., to component instances only.
- *Element name*: A role is assigned to all elements that follow a specified naming pattern, e.g., a role is assigned to all elements with the string "Service" in their name.
- *Implementation technology*: A role is assigned to all elements that are implemented using a specific technology or that use a specific protocol stack for communication. For example, a role can be assigned to all EJB session beans or to all components that can be accessed remotely using Web Service protocols (the implementation technology is assigned to the elements of the architecture model during architecture extraction on the basis of LISA technology bindings [7]).
- *Element properties*: A role is assigned to elements with specific properties. The concept of properties corresponds to the properties concept introduced in the previous section. The above cases are special cases of such properties. Properties are provided by property functions. A number of standard property functions directly extract property values from the architecture model. Examples are element name, version, containing module and the associated technology binding.

An assignment rule can be created by using just one criterion or by combining multiple criteria in a logic expression.

After assigning roles to elements of an architecture, it is checked whether all properties required by roles and relationships can actually be provided by the elements the roles have been assigned to. We check for each element with an assigned role and for each relationship whether the values for the required properties can be provided by property functions. Simple property functions are able to read property values directly from the architecture model, where they have been stored during the architecture extraction process. Examples for such properties have already been listed above. If roles (and constraints on roles and relationships) require properties that are not present in the architecture model, these properties either have to be provided by additional property functions or they have to be provided manually.

The realization and implementation of additional property functions makes sense if the properties can be determined in a predictable and repeatable way in future applications of the reference architecture conformance analysis. In cases, where the realization of specific property functions is too expensive, or where no external source for the required property values exist, they can be provided manually.

The final step is the actual conformance checking. This step is performed after role assignment and after making sure that the required properties for the roles in the specific rules can be provided.

The whole process can be performed continuously to enable conformance checking as the system evolves. Continuous application is supported through the automation of architecture extraction, role assignment, and rule evaluation.

V. Application Context

The presented approach has been developed together with a company providing IT-services for different banks and insurance companies in Austria.

The company is developing large-scale SOA-based systems. Each SOA-based system consists of a number of SOA subsystems (service modules), where each subsystem provides a set of services and uses services provided by other subsystems. SOA-based subsystems are organized in multiple application domains. An application domain comprises multiple applications, and an application is comprised multiple service modules (see Figure 3).

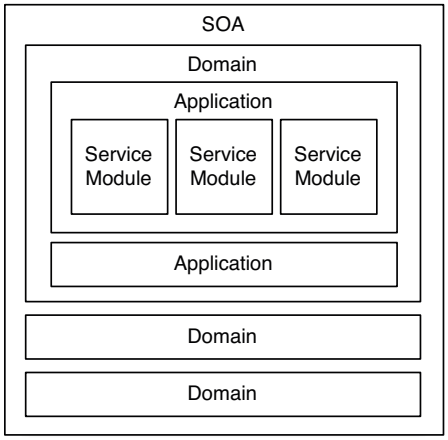

Fig. 3. SOA Landscape

A service module provides one or more services that can be used by other modules, and defines the implementation for the provided service. The implementation consists of a set of "internal components" implementing the provided services. Internal components are responsible for implementing the business logic of the provided services (Business Logic Objects - BLOs), for reading and manipulating data stored in databases (Data Access Objects - DAOs) and in mainframes (Customer Information Control System - CICS Objects), and also for accessing external services provided by other SOA subsystems (see Figure 4).

The core SOA of the company consists of more than 100 service modules. Service modules are developed by multiple development teams at different locations.

The company has defined a company-wide SOA reference architecture for services and applications in their domain. Central parts of this reference architecture can be formulated as rules, as shown in Table 1[3]. Formerly, these rules were defined

[3]While the rules shown in Table 1 are specific to the outlined application domain, the underlying mechanism for specifying such rules is not and could also be applied to other application domains.

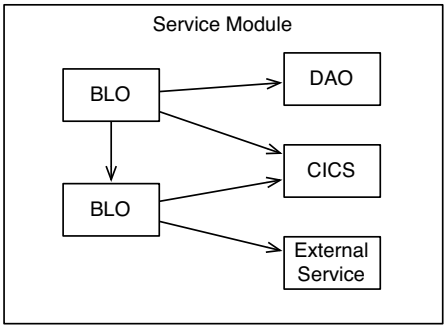

Fig. 4. Service Module Conceptual Overview

TABLE I. REFERENCE ARCHITECTURE RULES FOR ENTERPRISE APPLICATIONS IN THE BANKING DOMAIN

Rule 1: Data access components (DAOs) must not use business logic services and other DAOs
Rule 2: Each host transaction component must only be included once in a module
Rule 3: Host transaction components must not have outgoing connections
Rule 4: Services within one module must communicate via local EJBs
Rule 5: Services within one application (but in different modules) must communicate via local EJBs
Rule 6: Business logic services must only use data access components, host access components and host transaction components if they belong to the same module
Rule 7: Services of different domains must communicate via web services
Rule 8: Different applications should communicate via web services or remote EJBs
Rule 9: Applications may only access components of the service layer of another application

informally and had to be checked manually. An example of how the approach is actually presented to users is shown in the next section.

VI. TOOL SUPPORT

The approach and the checking process are supported in the LISA toolkit. The toolkit provides high-level editors for defining and using reference architecture specifications and for visualizing the results (i.e., rule violations) of the RA conformance checking. A main architectural driver when developing our approach has been usability, in terms of understandability and efficiency. The definition of the checking rules should be performed by the software architects of our industry partner, and should not require any special formal specification skills. In terms of efficiency, the RA conformance analysis itself should require as little human resources as possible to apply it repeatedly and incrementally, and thus be fully automated in the ideal case. Since these requirements were central to our industrial partner, we designed and reworked our approach multiple times to address these aspects (see next section).

In the following, we describe how RA rules can be specified and checked in the LISA toolkit. As an application example we are using one of the rules described in the previous section. This rule (Rule 1) states that data access components (DAOs) must not use business logic components (BLO) and other DAOs. Figure 5 depicts the corresponding rule definition for this rule. As shown in the figure, rules are presented using a tree structure, with each rule consisting of a set of roles (indicated by the person icon) and allowed or prohibited relationships (indicated by the arrow symbol or the crossed arrow symbol between roles). Roles may provide properties (indicated by the angle brackets symbol) and associated role assignment rules. Roles can be reused across multiple rules and thus need to be specified only once.

The rule in our example (and shown in Figure 5) consists of two roles (*DAO* and *Business Logic Service*) and of two prohibited relationships (*DAO* $\rightarrow$ *Business Logic Service* and *DAO* $\rightarrow$ *DAO*). Each role further defines three properties and an assignment rule for automated role assignment. The role assignment rule for the role *Business Logic Service* is shown in Figure 6. As shown in the figure an assignment rule is defined using simple logical expressions (and, or, not) along with property functions. Property functions use the previously defined role properties. The assignment rule for the Role *BLO* states that the role will be assigned to elements of the type *Component Instance* with a specific component binding and to all elements of the type *Service Instance*.

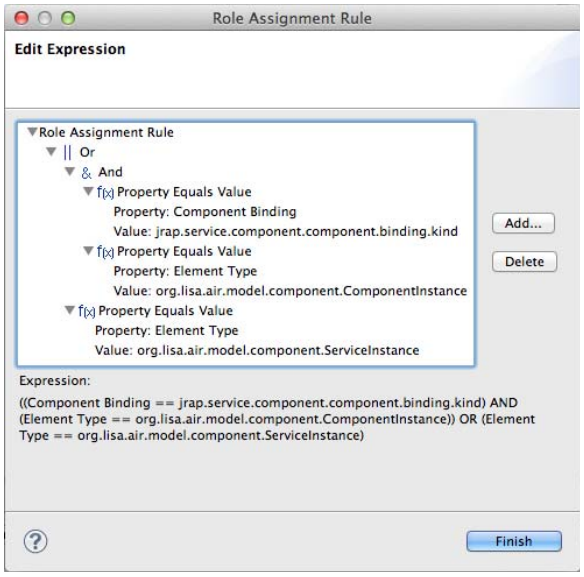

Fig. 6. Building Expressions through Composition of Property Functions

In order to check a system for conformance to a previously defined reference architecture, the defined constraint model needs to be instantiated for a particular architecture model of a system. The instantiation of the constraint model automatically triggers the RA conformance checking process described in Section IV. The result of the checking process (i.e., the rule violations) can be visualized in architecture diagrams. For example, Figure 7 shows a part of an internal configuration of a service module, consisting of components and their runtime dependencies. Dependencies that violate one ore more RA rules are highlighted in red and are annotated with an error marker in the diagram. A details view (next to the diagram) provides a more detailed description of the rule violations of the selected element in the diagram. In addition to visualizing RA violations in architecture diagrams, violations can also be viewed via a web interface providing different searching and filtering capabilities.

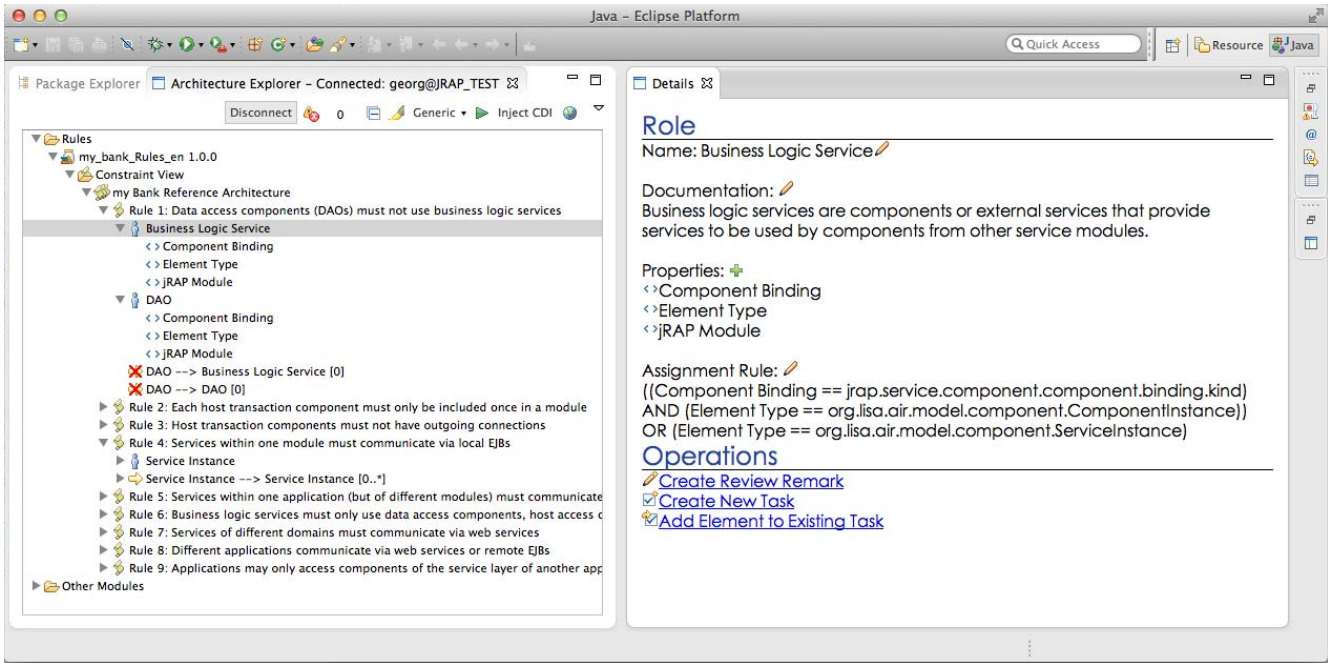

Fig. 5. Rule Specification Editor

VII. Research Approach and Validation

We based our research design on action research [10][11] since this methodology is particularly suited when performing research in an industrial context [12]. In action research (AR) *action* and *research* are combined into a structured process called the AR cycle [13], consisting of five steps: diagnosing, action planning, action taking, evaluating, and specifying learning. As outlined by Kock [14] action research is a qualitative research methodology, which typically involves the researcher in the study design. This results in threats like uncontrollability, contingency, and subjectivity which need to be addressed using different techniques, or antidotes according to Kock [14]. We used two of the mentioned antidotes including unit-of-analysis and multiple-iterations, which are useful for addressing all previously mentioned threats. In particular we followed an iterative study design, with each iteration following the AR cycle mentioned above. In addition, we added different SOA subsystems as units of analysis to subsequent iterations. SOA subsystems used as units of analysis included a customer relationship subsystem (CRM), a subsystem for checking the financial standing of customers (SEC), and a subsystem responsible for the granting of loans (CDT). The three steps of action planning, action taking and evaluation involved concept development and refinement, prototype implementation, architecture extraction (from the different subsystems), application of the approach, and evaluation of the results. We were involved in all steps, except architecture extraction, since we had no access to the implementation of the SOA subsystems. Evaluation was performed through a focused discussion of the results at the end of each iteration. Discussions were held with the software architecture group of our industry partner. This group is responsible for performing RA conformance analysis. Evaluation results influenced action planning for the next iteration. All in all, we performed five iterations over a period of twelve months to reach the currently implemented functionality. In the following we will briefly outline the performed iterations.

In the first iteration, we implemented the rule editor on the basis of on existing architecture analysis framework from our previous work. In addition, we implemented support for the three steps of the conformance checking process outlined in Section IV. We used the CRM system as a case study. Evaluation of the results showed that while the approach worked from a technical point of view, we had several issues with usability. Rules had to be defined on the basis of constraints of a general architecture analysis framework from a rather technical perspective (see [9]). Also the three steps of the conformance checking process were actually realized on the basis of this original analysis framework. Therefore, we aimed at improving usability in the next iteration.

In the second iteration, we developed the rules editor presented in this paper, where the architects could now define the rules in a manner they were familiar with (cf. Table I in Section V and Figure 5 in Section VI). We implemented support for the three steps of the conformance checking process described in Section IV. In the first step, we searched for potential role assignments. In the second step we identified missing property values. In the third step we evaluated the defined RA rules. Each step had to be manually triggered by the user. The result of each step was a list of detected problems, which we directly attached to the elements of the architecture model. Evaluation of this iteration showed that the rule editor worked now in a way that we could actually discuss and modify the rules with the architects of the company. However, the conformance checking process itself was still cumbersome, since detected problems like potential rule assignments, missing property values, and rule violations were attached to the elements of

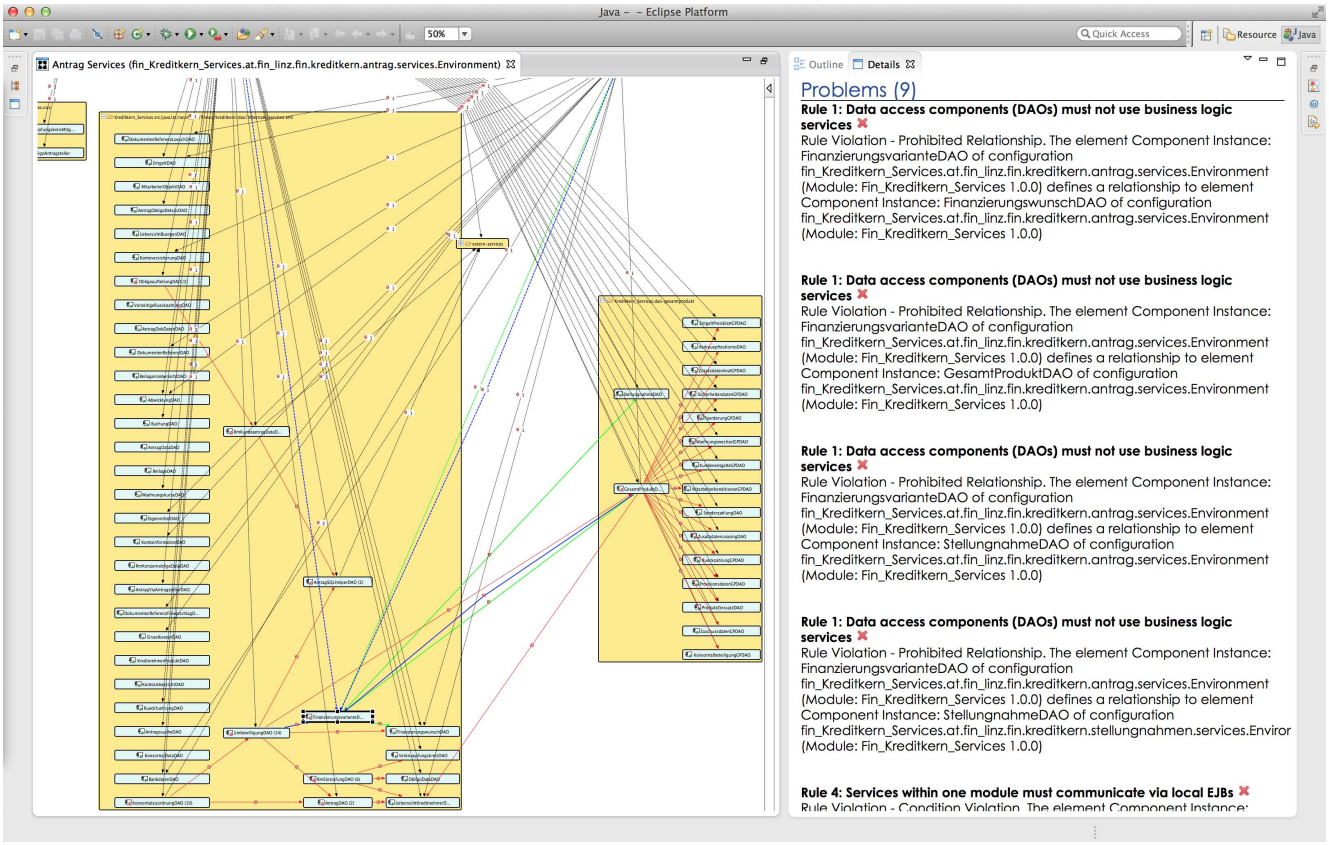

Fig. 7. Visualization of Rule Violations in Configuration Editors

the architecture model and had to be confirmed respectively resolved manually by the architects. As a result, we aimed at improving usability of the conformance checking process in the next iteration.

In the third iteration, we developed a dedicated conformance checking process view supporting the single steps of the RA checking process. The view should guide the user through the checking process and summarize the results (i.e., the detected problems) of the different steps to eliminate the need for inspecting the individual elements of the architecture model. The resulting view is shown in Figure 8. The view lists the single steps of the checking process, which can be trigged directly in the view. As shown in the figure, the the view also lists the results of each step (in this case proposals for role assignments) and enables an architect to confirm or reject the proposed actions in each step. We also introduced the SEC and the CDT system as additional case studies in this iteration. Evaluation of this iteration showed that the conformance checking view now supported the process as expected. However, the required manual intervention for confirming role assignments and providing property values was considered to be too high. This became especially evident when using the additional systems for testing, which required a higher effort for confirming roles and providing missing property values. When discussing the results of this iteration, we decided that the required manual efforts should be reduced and ideally eliminated at all.

The goal of the fourth iteration was to eliminate the need for the manual provisioning of property values. This should be achieved by fetching required properties from external systems. In our context we required information about the application, the domain, and the service module of single services (see Section V). This information is available in service registries, which are part of the enterprise architecture management infrastructure. Therefore, we intended to retrieve missing values from these external resources. However, it turned out that the required information was often not available or out of date. This triggered a major change of how such additional architectural information is provided, which was realized in the next iteration.

In the fifth iteration, our industry partner decided to introduce a new software component model for future development of the different SOA services. In particular, they decided to provide the required information through specific component type metadata and provide this information as a mandatory part a system's implementation. This included the information on different component types, which we previously determined through naming heuristics. This eliminated the false positives in role assignments and enabled us to perform role assignments completely automatically. Further, required property values like application and domain are now part of the metadata, which is deployed with service modules. This eliminates the problem of missing property values at this stage. We now check the relevant information already during architecture

TABLE II. CLASSES, COMPONENTS, AND ROLE ASSIGNMENTS FOR THREE SOA SUBSYSTEMS

SOA Subsystem	Number of Classes	Number of Components/ Services	Number of Role Assignments
Subsystem 1 CRM	466	65	182
Subsystem 2 SEC	1022	104	284
Subsystem 3 CDT	2100	151	389

extraction, which ensures that the required information is actually provided by the development team.

The result is that the whole RA checking process can now be performed completely automatically because of the extensions to the component model and the service implementation. Now, a user only has to select an architecture model and a reusable reference architecture (rules) specification. The mapping of roles to the elements of the architecture model and the retrieval of the required properties are performed automatically. The evaluation of defined RA rules has been performed automatically since the first iteration of the AR cycle.

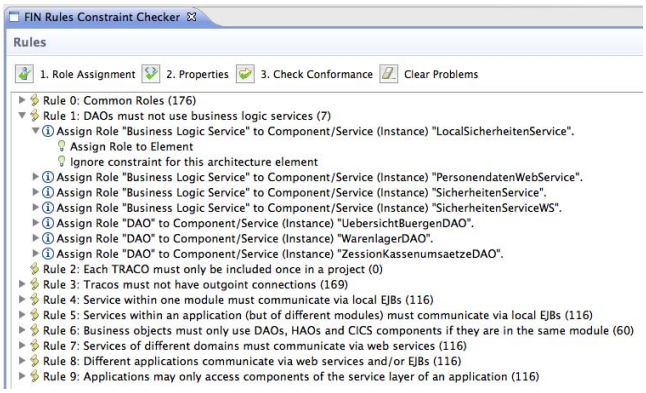

Fig. 8. Conformance Checking Process View (obsolete)

VIII. EXPERIENCES AND FUTURE WORK

During the five iterations we applied our approach to three different SOA subsystems of our industry partner. An overview of classes, components and role assignments in the three case studies is provided in Table II. For describing the 9 rules of the reference architecture depicted in Table I, we required 6 different roles, 13 role properties, 8 role assignment rules, and 10 relationship constraints (roles can be shared between different rules). The number of role assignments required for the individual subsystems as shown in Table II indicate that manual assignment of roles is not feasible even for smaller projects.

The aim of the approach described in this paper was to provide a framework for defining and checking rules where rules can be defined by software architects without the need of programming such rules. This goal has only partially been reached. While rules can be defined without extending the LISA toolkit programmatically, other elements of the approach, i.e., the retrieval of property values, and certain model checks, might still require programmatic extensions in the future. Due to the modular structure of the analysis framework, an additional property retrieval component (technically a property provider) can be implemented in under an hour. Realizing additional providers is rather the exception than the rule and should only be necessary if the approach is applied to a significantly different context or type of system, or if the kinds of analysis change significantly and require additional information from the architecture meta-model and other information resources.

Currently the approach has been designed to be used as part of quality gate reviews in which software architects analyze the architecture of single SOA subsystems. In such a setting violations are detected rather late in the development process. In the future we plan to support reference architecture conformance checking as part of continuous integration build processes. Integrating reference architecture checking with the build process permits the early detection of architecture violations and continuous quality monitoring. Integration with build processes will require further extensions of our work like the incremental extraction of the architecture models as part of daily build processes to ensure that the analysis is performed with the latest and most current architecture model.

IX. RELATED WORK

Software architecture conformance checking is addressed by numerous approaches in both research and practice. Most of the available approaches focus on the conformance of (an intended) system architecture and code, which does not address the requirement of reuse that is inherent to reference architectures and also lacks the extra level of abstraction that is required for addressing a family of systems.

Reflexion models as introduced by Murphy [15] is an approach for mapping a high-level architectural model to the source model extracted from an implementation. The resulting reflexion model shows where the high-level model does or does not agree with the low-level source model. This way, reflexion models are an effective way of architecture/implementation conformance analysis. Examples for approaches supporting reflexion models are JITTAC [16], SAVE [17] and ConQUAT [18]. Reflexion models have some similarities to our approach, but lack some fundamental properties for supporting reference architectures. The main issues are reusability and scope. Reflexion models mainly target architecture/implementation conformance. As such they require a direct and typically 1:1 mapping of high-level model elements to implementation elements like packages. This is different to our approach, were a role is typically assigned to several hundred elements of an architecture model. Also the mapping of roles is based on general enterprise-wide rules and is thus not application-specific. As a result the defined reference architecture can be reused across all systems of the domain. Also note that roles are automatically mapped to elements of a high-level architecture model (including a component and service model) rather than to elements of a source model as in reflexion models and the mapping does not need to be adjusted for different systems. Finally, the concept of properties is not present in reflexion models. While typed reflexion models enable the analysis of different kinds of relationships, properties as supported in our approach allow to check for additional properties on both relationships (in addition to relationship types) and the elements themselves.

Dependency analysis approaches like Lattix [19], Structure 101 [20], and Sonargraph [21] can be seen as a specific implementation of the concept of reflexion models and thus also support architecture/implementation conformance checking. Architectures are typically described as a graph of namespaces (packages) and allowed/disallowed dependencies among these namespaces at a rather low level of abstraction. The approach described in this paper works mainly at the component abstraction level, though our role-based approach supports other abstraction levels as well. As with reflexion models, dependency analysis provide insufficient support for reusing an architecture specification across a family of systems, i.e., the intended architecture typically has to be defined specifically for each individual system. Finally, dependency analysis approaches are restricted to the source model (classes and dependencies). Additional architectural information on a higher level of abstraction like the protocol used for a specific communication relation are not supported by such approaches.

In [22] Selonen and Xu present an approach for validating UML models against architectural profiles. An architectural profile defines stereotypes and constraints on how stereotypes are related to each other. Stereotypes are then mapped onto elements of an UML model using an architecture profile. Finally, elements with stereotypes are validated against the constraints defined by the architecture profile to find mismatches. Conceptually the approach is similar to what is offered in our approach. Stereotypes that are mapped to UML elements correspond to roles in our approach. Architecture profiles are reusable for systems of a particular domain similar to our reference architecture rules. However, both the creation of UML design models and the assignment of stereotypes have to be performed manually. We automate these activities, which is necessary to perform reference architecture conformance checking continuously for the various systems that are part of a SOA.

In [23] Schmerl and Garlan describe an approach for defining and automatically analyzing architectural styles based on the ACME/Armani architecture description language [24]. Their approach consists of two separate activities. (1) The definition of an architectural style and (2) the definition of architecture design models based on a previously defined architectural style. An architectural style consists of a set of architecture element types, a set of design rules, a set of design analyses, and a set of minimal required structures. Design rules specify heuristics, invariants, and composition constraints on architecture elements. These rules are described in the Armani predicate language, which consists of standard predicate logic expressions with composable terms, primitive functions, and quantification capabilities. In addition to primitive functions, user-definable functions (called design analysis) can be used within an expression. While the basic concepts of the approach presented by Schmerl and Garlan are similar to what is offered in our approach, our constraint model is simpler than the Armani predicate language. For example, we also use predicate logic expressions and provide primitive functions for determining element properties. In addition, external (user defined) functions can be integrated. But quantification is directly tied to an element like a role or a relationship between roles in our approach. Also nested quantifications like in ACME/Armani are not supported in our approach. This simplification has been a deliberate decision to increase the usability for architects. Another important difference is that elements defined by an architectural style are directly instantiated in the architecture model in ACME/Armani, while in our approach reference architectures and architecture models are initially decoupled from each other (the relationship between a reference architecture and an architecture model is established by defining role mappings as part of the checking process). This is mainly due to the intended usage scenario of both approaches. The approach by Schmerl and Garlan is mainly targeted at supporting the design process where a new architecture design is based on a reference architecture from the beginning. Our approach, however, has been mainly developed to support the continuous checking of the conformance of already implemented and evolving systems.

Deiters et al. [25] present a rule-based approach for architectural compliance checking of enterprise information systems. They use first order logic for describing the intended architecture, an existing architecture design, and the mapping between the two. Each of the three descriptions is represented as a single knowledge base containing (Prolog-like) facts and potentially logical rules. The checking of architectural compliance is technically realized by executing architectural rules as queries upon the union of knowledge bases. The provided rules are restricted to layer checking (as stated in [26]). An extension of this approach is presented in [26] where the knowledge base is partly generated from source and meta models through components called wrappers in their approach. The rules for checking conformance are, however, still first-order logic formulas. The main drawback of their approach is usability because of the need of writing rules as Prolog-like facts. They thus list model-based rule specification (as supported in our approach) is part of their future work [26]. Another extension of the work presented in [25] aims at supporting the design process through architectural building blocks (ABBs) [27][28]. ABBs provide concepts like roles, relationships, and assurances which are similar to our concepts of roles, relationships, and constraints. The architecture model used in this approach is less expressive (entities and dependencies) than the LISA model used in our approach. Also, the use of element properties in logical expressions, the inclusion of external property functions, and the support for automatic role assignment seem not to be supported in ABBs. Usability is still a concern, since rules and constraints are fact bases which are defined textually in their approach, while we use a projectional editor on a constraint model. Finally, their main target is support for the design process while we aim to support reference architecture conformance checking as part of quality control.

X. Conclusion

We have presented an approach for automatic reference architecture conformance checking of different systems within a SOA landscape. The approach has been developed to support software architects during quality control of service-oriented software systems in the banking domain.

Reference architectures rules can be defined an a high-level of abstraction using a projectional rules editor and can be reused and applied to different SOA subsystems. The whole process of architecture conformance checking including the extraction of an architecture model from an existing software

system (as part of our previous work), the mapping of the concepts of the reference architecture specification (i.e., roles) to the elements of the architecture model, the retrieval of additional element properties needed for rule evaluation, and the evaluation of the (conformance) rules themselves, can be performed automatically in our approach.

The approach evolved from a semi-automatic approach to a fully automatic approach during its development. Feedback cycles with the company architects and the application to several SOA subsystems targeted especially the usability of the approach according to the architect's needs and eventually led to changes of the component model of our industry partner to enable a consistent provisioning of architectural information as part of the system implementation. The approach has been tested with several SOA subsystems during its development and is currently deployed as part of architecture and development quality gates within the company. Future work includes the integration with company build infrastructures and other review activities. We are also exploring the use of the approach for checking the conformance to security patterns addressing specific security threats.

REFERENCES

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice (2nd Edition)*, 2nd ed. Addison-Wesley Professional, 4 2003.

[2] E. Y. Nakagawa, P. Oliveira Antonino, and M. Becker, "Reference architecture and product line architecture: A subtle but critical difference," in *Software Architecture*, ser. Lecture Notes in Computer Science, I. Crnkovic, V. Gruhn, and M. Book, Eds. Springer Berlin Heidelberg, 2011, vol. 6903, pp. 207–211. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23798-0_22

[3] S. Angelov, J. J. M. Trienekens, and P. Grefen, "Towards a method for the evaluation of reference architectures: Experiences from a case," in *ECSA '08: Proceedings of the 2nd European Conference on Software Architecture*. Springer Berlin Heidelberg, 2008, pp. 225–240.

[4] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley, 1 2010.

[5] D. Greefhorst and E. Proper, *Architecture Principles: The Cornerstones of Enterprise Architecture*. Springer, 2011, vol. 4.

[6] R. Cloutier, G. Muller, D. Verma, R. Nilchiani, E. Hole, and M. Bone, "The concept of reference architectures," *Systems Engineering*, pp. 14–27, 2009.

[7] R. Weinreich, C. Miesbauer, G. Buchgeher, and T. Kriechbaum, "Extracting and facilitating architecture in service-oriented software systems," in *2012 Joint 10th IEEE/IFIP Working Conference on Software Architecture & 6th European Conference on Software Architecture (WICSA-ECSA 2012), DOI 10.1109/WICSA-ECSA.212.16*. IEEE, 2012.

[8] G. Buchgeher and R. Weinreich, "Towards continuous reference architecture conformance analysis," in *7th European Conference on Software Architecture (ECSA 2013)*, ser. Lecture Notes in Computer Science, K. Drira, Ed., vol. 7957. Springer Berlin Heidelberg, 2013, pp. 332–335. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-39031-9_32

[9] R. Weinreich and G. Buchgeher, "Towards supporting the software architecture life cycle," *Journal of Systems and Software*, vol. 85, no. 3, pp. 546 – 561, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121211001361

[10] N. Kock, *Information systems action research: An applied view of emerging concepts and methods*, ser. Integrated Series in Information Systems. Springer, 2007, vol. 13.

[11] A. Lee, *Action is an Artifact: What Action Research and Design Science Offer to Each Other*, ser. Integrated Series in Information Systems. Boston, MA: Springer US, 2007, vol. 13, ch. 3, pp. 43–60.

[12] D. E. Avison, F. Lau, M. D. Myers, and P. A. Nielsen, "Action research," *Commun. ACM*, vol. 42, pp. 94–97, 1 1999.

[13] G. I. Susman and R. D. Evered, "An assessment of the scientific merits of action research," *Administrative science quarterly*, pp. 582–603, 1978.

[14] N. Kock, "The three threats of action research: a discussion of methodological antidotes in the context of an information systems study," *Decision support systems*, vol. 37, no. 2, pp. 265–286, 2004.

[15] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: bridging the gap between design and implementation," *Software Engineering, IEEE Transactions on*, vol. 27, no. 4, pp. 364–380, 4 2001.

[16] J. Buckley, S. Mooney, J. Rosik, and N. Ali, "Jittac: a just-in-time tool for architectural consistency," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1291–1294. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486987

[17] J. Knodel, D. Muthig, M. Naab, and M. Lindvall, "Static evaluation of software architectures," in *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 279–294.

[18] F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juergens, "Flexible architecture conformance assessment with conqat," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 247–250. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810343

[19] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," *SIGPLAN Not.*, vol. 40, no. 10, pp. 167–176, 10 2005.

[20] R. S. Sangwan, P. Vercellone-Smith, and P. A. Laplante, "Structural epochs in the complexity of software over time," *IEEE Software*, vol. 25, no. 4, pp. 66–73, 7 2008.

[21] Sonargraph. (2013) http://www.hello2morrow.com/products/sonargraph. [Accessed: Jan. 31, 2014]. hello2morrw GmbH.

[22] P. Selonen and J. Xu, "Validating uml models against architectural profiles," in *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-11. New York, NY, USA: ACM, 2003, pp. 58–67. [Online]. Available: http://doi.acm.org/10.1145/940071.940081

[23] B. Schmerl and D. Garlan, "Acmestudio: supporting style-centered architecture development," in *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, 2004, pp. 704–705.

[24] R. T. Monroe, *Capturing Software Architecture Design Expertise with Armani*. Carnegie-mellon univ pittsburgh pa school of computer Science, 10 2001.

[25] C. Deiters, P. Dohrmann, S. Herold, and A. Rausch, "Rule-based architectural compliance checks for enterprise architecture management," in *2009 IEEE International Enterprise Distributed Object Computing Conference*. IEEE, 9 2009, pp. 183–192.

[26] S. Herold, M. Mair, A. Rausch, and I. Schindler, "Checking conformance with reference architectures: A case study," in *17th International EDOC Conference (EDOC 2013), Vancouver, Canada*. IEEE, 9 2013.

[27] C. Deiters and A. Rausch, "A constructive approach to compositional architecture design," in *ECSA '11: Proceedings of the 5th European Conference on Software Architecture*, ser. Lecture Notes on Computer Science. Springer Berlin Heidelberg, 9 2011, pp. 75–82.

[28] ——, "Assuring architectural properties during compositional architecture design," in *Proceedings of the 10th International Conference on Software Composition (SC 2011)*, ser. Lecture Notes in Computer Science, vol. 6708. Springer Berlin Heidelberg, 7 2011, pp. 141–148.