A Metamodel for the Support of Semantically Rich Modular Architectures in the Context of Static Architecture Compliance Checking

Leo Pruijt
Information Systems Architecture Research Group
HU University of Applied Sciences
Utrecht, The Netherlands
leo.pruijt@hu.nl

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

ABSTRACT
Architecture Compliance Checking (ACC) is an approach to verify the conformance of implemented program code to high-level models of architectural design. Static ACC is focused on the module views of architecture and especially on rules constraining the modular elements. This paper proposes an approach for support of semantically rich modular architectures (SRMAs) in the context of static ACC. An SRMA contains modules of semantically different types, like layers and components, constrained by rules of different types. Our approach is grounded in a metamodel, which enables support of rich sets of module and rule types and which enables extensive support of the semantics of these types. To validate the feasibility of the metamodel, an open source prototype implementation was developed, tested and applied in practice.

Categories and Subject Descriptors
D.2 [Software Engineering]
D.2.2 Design Tools and Techniques
D.2.11 Software Architecture

General Terms
Design, Verification.

Keywords
Software Architecture; Modular Architecture; Architecture Compliance; Static Analysis; Metamodel

1. INTRODUCTION
Software architecture is of major importance to achieve the business goals, functional requirements and quality requirements of a system. However, architectural models tend to be of a high-level of abstraction, and deviations of the software architecture arise easily during the development and evolution of a system [12]. Architecture Compliance Checking (ACC) is an approach to bridge the gap between the high-level models of architectural design and the implemented program code, and to prevent decreased maintainability, caused by architectural erosion. Architectural erosion is “the phenomenon that occurs when the implemented architecture of a software system diverges from its intended architecture” [22]. The opposing term, architecture compliance, is defined by Knodel and Popescu as “a measure to which degree the implemented architecture in the source code conforms to the planned software architecture” [8].

Many tools and techniques are available to analyze a software system and to reconstruct, visualize, check, or restructure its architecture [5]. In our research, we focus on tool support for static ACC, in which the software is analyzed without executing the code. Tools of this type, which we label as static ACC-tools, focus on the modular structure in the source code.

Although Shaw and Clements included ACC in 2006 in their list of promising areas [21], the adoption of ACC-tools is still limited [7], [22] and research is necessary to advance and improve current methods and tools [3]. Different studies have compared ACC-tools and techniques, and these studies revealed large discrepancies in terminology, approach and performance [5], [8], [14], [18], [22].

Our research builds on these studies, but we focus on ACC support of semantically rich modular architectures (SRMAs). We use this term for expressive modular architecture descriptions, composed of semantically different types of modules, like layers, subsystems and components, which are constrained by rules of different types. These may be explicitly defined rules, like “module A is only allowed to use module B”, but may also be rules inherent to the semantics of a module type, like “a layer is not allowed to use higher-level layers”. In contrast to an SRMA, a semantically poor modular architecture description includes modules of only one module type and rules of only a few rule types, like the two basic types “is not allowed to use” and “is allowed to use”.

Adersberger and Philippsen [1] consider the support of semantically rich architecture models essential for the integration of ACC in model-driven engineering. Furthermore, they make clear that support of semantically rich constructs reduces the number of rules that need to be specified, compared to semantically poorer boxes and lines models. Modules with specific semantics enhance the expressiveness of a modular architecture and support architecture reasoning. A rich set of module types provides a language to express characteristics of the modules in an architectural model. A rich set of rule types provides a language to express constraints on the modules in an architectural model. This language allows architects to define logical rules in a comparable way as expressed in regular language, without the need to translate a logical rule to one or more rules at ACC-tool level.

In a previous study, we compared eight commercial and academic ACC-tools on their support of SRMAs [17]. We concluded that the tested tools were providing useful support for dependency...
checking, but only limited support for SRMAs. Five tools supported no semantic differences between modules. The other three tools provided some specific kind of support of layers, components or façades, but none provided extensive support for more than one type. Furthermore, all tools restricted rule support to dependency rules only, and to simple rule types, like “is allowed to use”, “is not allowed to use”, or “must use”. More complex rules were not supported explicitly and in many cases one logical rule required the combination of several rules to be specified in the ACC-tool. Consequently, a gap has to be bridged between architecture design of SRMAs and ACC tools, with as potential disadvantages, loss of architectural rules, reduced traceability, reduced overview, and reduced productivity.

In this study, we focus on the following research question: How can support be provided for SRMAs in the context of static ACC? To answer this question, we followed a process of design research [15]. Iteratively we identified requirements, studied existing tools, designed a metamodel, developed and tested an open-source ACC-tool as prototype, and we applied this tool during ACC’s on professional systems. These iterations spanned three consecutive years in which groups of students in computer science participated.

The contribution of this paper is twofold. First, we present a metamodel for extensive support of SRMAs in the context of static ACC. The metamodel adds to the knowledge base and may be used to enhance existing tools, or to develop new approaches. Second, we introduce an open-source implementation prototype of the metamodel, which illustrates the feasibility of the metamodel.

The next section of this paper describes and illustrates the concept “semantically rich modular architecture”, and it introduces a classification of common module types and common rule types. We use these types in our research to concretize requirements to SRMA support. Section 3 introduces our metamodel for SRMA support in the context of static ACC. Section 4 presents the prototype implementation of our metamodel concisely. Section 5 compares the outcome of our study to related work, and Section 6 concludes this paper and addresses future work.

2. SEMANTICALLY RICH MODULAR ARCHITECTURES

According to Perry and Wolf, software architecture “provides the framework within which to satisfy the system requirements and provides both the technical and managerial basis for the design and implementation of the system” [16]. Static ACC does not cover the full width of software architecture, but only the static structure of the software (planned and implemented); in other words, the module views of architecture [4], or modular architecture.

A planned modular architecture should describe the modular elements, their form (properties and relationships) and rationale [16]. Modular elements, properties and relationships, are in ACC’s center of attention, and should be included in a complete compliance check. A modular element, or module, is an implementation unit of software with a coherent set of responsibilities [4]. Properties and relationships express architectural rules that constrain a modules’ implementation [16].

2.1 Example of an SRMA

A semantically rich modular architecture includes modules of semantically different types, while a variety of types of rules may constrain the modules. As an example of an SRMA, Figure 1 shows a small part of an architecture model of one of the systems at an airport. This system is used to manage the state and services of human interaction points where customers communicate with baggage handling machines, self-service check-in units, etc. Examples in the rest of this document refer to elements in Figure 1.

Figure 1 shows UML icons for three semantically different types of modules: packages, components and interfaces. Layers are the fourth module type in the model (indicated by lines, since layers are not supported by UML). Finally, Spring and Hibernate represent the fifth type of module in the model: external system.

UML dependency relations in this example indicate is-allowed-to use rules; for instance, module HiWebApp is only allowed to use the modules HiForms and HimInterface, no others. Some other rules are not visible in the diagram. For example, rules related to the layered style, like “Technology Layer is not allowed to use Interaction Layer. Other examples of not visible rules are naming rules and rules inherent to components with interfaces.

2.2 Common Module and Rule Types

To enable compliance checks of SRMAs, rich sets of module and rule types should be supported. In a previous study [17], we presented a classification of common module types and common rule types. In this study, we use these common types as functional requirements to SRMA support. The next sub-sections describe these common module and rule types concisely to enhance practical understanding before the metamodel is presented. For a more in-depth discussion of the common module and rule types, we refer to our previous study.

Figure 1. Example of an SRMA model
2.2.1 Common Module Types
SRMAs may contain modules of different types, with very different semantics. We identified five common types of modules relevant for static ACC:

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

Layers represent units in the system design with additional semantics. Layers have a hierarchical level and constraints on the relations between the layers. We cite Larman [11], who summarizes the essence of a layered design as “the large-scale logical structure of a system, organized into discrete layers of distinct, related responsibilities. Collaboration and coupling is from higher to lower layers.”

Components within software architecture are designed as autonomous units within a system. The term component is defined in different ways in the field of software engineering. In our use, a component within a modular architecture covers a specific knowledge area, provides its services via an interface and hides its internals (in line with the system decomposition criteria of Parnas [13]). Consequently, a component differs from a logical cluster in the fact that it has a Facade sub module and hides its internals. Since our definition of component is intended for modular architectures, it does not include runtime behavior as in the “component and connector view” of architecture [4].

Façades are related to a component and act as an interface as described under components. We use the term façade, referring to the façade pattern [6], to differentiate with the Java interface, which has not exactly the same meaning as a design-level interface. A façade may be mapped to multiple elements at implementation level, like Java interface classes, exception classes and data transfer classes.

External systems represent platform and infrastructural libraries or components used by the target system. Useful ACC support includes the identification of external system usage and checks on constraints regarding their usage [2].

2.2.2 Common Rule Types
Modular architectures may contain rules of different types, where each rule type characterizes another kind of constraint on a module. These constraints are categorized in literature [4], [16] as properties and relationships. Our inventory of architectural rule types, in principle verifiable by static ACC, resulted in two categories related to properties and relationships: Property rule types and Relation rule types. The identified rule types are described and exemplified in Table 1.

Property rule types constrain a certain characteristic of the elements included in the module and their sub modules. Elements et al. [4] distinguish the following properties per module: Name, Responsibility, Visibility, and Implementation information. We identified rule types associated to these properties and named them accordingly, except two types (Facade convention, Inheritance convention), which represent the property Implementation information.

Relation rule types specify whether a module A is allowed to use a module B. The basic types of rules are “is allowed to use” and “is not allowed to use”. However, we encountered useful specializations of both basic types, which we included in the classification. Table 1 shows the two included specializations of “Is not allowed to use” (both specific for layers), and the three specializations of “is allowed to use”.

Table 1. Common rule types

<table>
<thead>
<tr>
<th>Category/Type of Rule</th>
<th>Description (D), Example (E)</th>
<th>Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property rule types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Naming convention</td>
<td>D: The names of the elements of the module must adhere to the specified standard.</td>
<td>[14]</td>
</tr>
<tr>
<td></td>
<td>E: HiDao elements must have suffix DAO in their name.</td>
<td></td>
</tr>
<tr>
<td>Responsibility convention</td>
<td>D: All elements of the module must adhere to the specified responsibility.</td>
<td>[11]</td>
</tr>
<tr>
<td></td>
<td>E: HiForms is responsible for presentation logic only.</td>
<td></td>
</tr>
<tr>
<td>Visibility convention</td>
<td>D: All elements of the module have the specified or a more restricting visibility.</td>
<td>[4]</td>
</tr>
<tr>
<td></td>
<td>E: HiManager classes have package visibility or lower.</td>
<td></td>
</tr>
<tr>
<td>Facade convention</td>
<td>D: No incoming usage of the module is allowed, except via the facade.</td>
<td>[6]</td>
</tr>
<tr>
<td></td>
<td>E: HiManager may be accessed only via HimInterface.</td>
<td></td>
</tr>
<tr>
<td>Inheritance convention</td>
<td>D: All elements of the module are sub classes of the specified super class.</td>
<td>[14]</td>
</tr>
<tr>
<td>Relation rule types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Is not allowed to use</td>
<td>D: No element of the module is allowed to use the specified to-module.</td>
<td>[8]</td>
</tr>
<tr>
<td></td>
<td>E: HiPanels is not allowed to use HiWS.</td>
<td></td>
</tr>
<tr>
<td>Back call ban (specific for layers)</td>
<td>D: No element of the layer is allowed to use a higher-level layer.</td>
<td>[20]</td>
</tr>
<tr>
<td></td>
<td>E: Service Layer is not allowed to use the Interaction Layer.</td>
<td></td>
</tr>
<tr>
<td>Skip call ban (specific for layers)</td>
<td>D: No element of the layer is allowed to use a lower layer that is more than one layer lower.</td>
<td>[20]</td>
</tr>
<tr>
<td></td>
<td>E: Interaction Layer is not allowed to use the Infrastructure Layer.</td>
<td></td>
</tr>
<tr>
<td>Is allowed to use</td>
<td>D: All elements of the module are allowed to use the specified to-module.</td>
<td>[4]</td>
</tr>
<tr>
<td></td>
<td>E: HiWebApp is allowed to use HiForms.</td>
<td></td>
</tr>
<tr>
<td>Is only allowed to use</td>
<td>D: No element of the module is allowed to use other than the specified to-module(s).</td>
<td>[14]</td>
</tr>
<tr>
<td></td>
<td>E: HiForms is only allowed to use HiPanels.</td>
<td></td>
</tr>
<tr>
<td>Is the only module allowed to use</td>
<td>D: No elements, outside the selected module(s) are allowed to use the specified to-module.</td>
<td>[14]</td>
</tr>
<tr>
<td></td>
<td>E: HiDao is the only module allowed to use CorporateWebcore.</td>
<td></td>
</tr>
<tr>
<td>Must use</td>
<td>D: At least one element of the module must use the specified to-module.</td>
<td>[8]</td>
</tr>
<tr>
<td></td>
<td>E: HiDao must use CorporateWebcore.</td>
<td></td>
</tr>
</tbody>
</table>

1 Ref= primary literature reference
3. SRMACC METAMODEL

In this section, we introduce a metamodel to provide support for SRMAs in the context of static ACC. The metamodel, we labeled it “SRMACC metamodel”, identifies, describes and relates the core concepts needed to address the following objectives regarding SRMA support. The first is to provide basic SRMA support, which includes the provision of sets of common module and rule types and the functionality to check rules of these types. The second is to provide extensive SRMA support, which adds support of the semantics of the common module and rule types. The third is to enable configuration of the provided support.

To enable reuse and different implementations, the metamodel is presented in four UML class diagrams, each focusing on a different aspect. Composition associations without a name in these diagrams, should be read as “isComposedOf”.

3.1 Definition of the Modular Architecture

The metamodel in Figure 2 focuses on the definition of the planned modular architecture. The model shows that an instance of a SoftwareArchitecture (within the context of static ACC) is composed of a set of Modules, the architectural elements, and a set of AppliedRules, constraints on the architectural elements (properties and relationships, in terms of Perry and Wolf [16]). AppliedRules are characterized by their RuleTypes, which are grouped into Categories (e.g., “Property rule” and “Relationship rule” within our classification).

Module represents instances of architectural elements, which may be composed of many sub-modules recursively. In line with the constraints of the decomposition style [4], a module can have only one parent. Basic support of SRMAs includes the provision of a set of ModuleTypes, like the types discussed in Section II, which define the semantic properties of the modules.

AppliedRule represents instances of rules, where each instance constrains a Module; the from-module in the metamodel. An AppliedRule is of a certain RuleType, which defines the kind of constraint applied to the from-module. For example, the AppliedRule “HiManager may be accessed only via HimInterface” of RuleType “Facade convention” constrains Module-from “HiManager”. Some types of applied rules include also a Module-to in their constraint, in which case a relationship is defined. For example, AppliedRule “HiDao must use CorporateWebcore”, of RuleType “Must use”, constrains Module-from “HiDao” in its use of Module-to “CorporateWebcore”. Finally, support of exceptions, is also included within the metamodel. An exception rule is also an instantiation of AppliedRule, however the exception rule is linked to the original rule via association hasException, in order to make the exception traceable to the original rule.

3.2 Support of the Semantics of the Types

Inclusion in the metamodel of ModuleType and RuleType, with their properties and associations, enables support for the semantics of the provided types. First, type-specific properties may be included and configured. For example, RuleType with name “Visibility convention” defines not only the type of constraint of an AppliedRule, but it also defines, inter alia, the Category, the values allowed to include in a rule, and the severity of a violation against a rule of this type.

Second, more advanced support of the semantics of the types may be provided when logical relationships between the types are included in the model; shown in Figure 3 as three associations. The association hasDefault may be used to create rules (inherent to the type of module) automatically when a module is created. For instance, when a module of type “Layer” is created, a “Back call ban” rule and a “Skip call ban” rule might be generated, based on included instantiations of association hasDefault. The association allows may be used to present to the tool-user a list of RuleTypes, suitable to the ModuleType of the constrained module. For example, a “Back call ban” is allowed only in case of ModuleType “Layer”, and a module of type “External system” is not allowed to be constrained by any type of rule, since it is not the subject of the ACC (conversely, usage of an external system may be constrained).

Finally, the association allowsAsException specifies for a certain RuleType, which RuleTypes are allowed as an exception to an instantiated AppliedRule. For instance, as an exception to a rule of type “Naming convention”, only a rule of the same type is allowed.

3.3 Module Mapping

A Module may represent one or more implementation units of a software application. To enable ACC on various versions of the software, the metamodel in Figure 4 includes the association, Module mapsTo DefinedSoftwareUnit. An instantiation of DefinedSoftwareUnit represents an implementation unit of a certain type (package, class, …) and in case of a composite unit, all
its underlying units. To be able to find the unit when an ACC is performed, attribute uniqueName needs to be set with a string in the form of a path-and-name-combination or in the form of a regular expression. At this point, support to the tool-user is desirable, which may be provided based on analysis data of the current version of the software.

The metamodel in Figure 4 includes also the basics for ACC support of complex Applications, which are subdivided in technical Projects. Each project may have its own class path and programming language and possibly its own SoftwareArchitecture. Our metamodel also features that a SoftwareArchitecture with the same sets of Modules and AppliedRules may be reused in different projects. In that case, only the mapping will differ per project.

3.4 Compliance Checking

The metamodel in Figure 5 shows the concepts needed for the actual compliance check between the planned modular architecture and the implemented architecture. The planned architecture is composed of Modules and AppliedRules. The implemented modular architecture, including all the code-types in the software, relevant properties of these types and the dependencies between the types, is represented by AnalyzedSoftwareUnit and Dependency. In dependency and violation reports it is useful to include the type of the dependency [18]. Reason why DependencyType is included, which stands for the set of dependency types. It enables a standardized presentation of these types to the tool-user, in forms and reports.

An instance of Violation represents an infringement of an AppliedRule by an AnalyzedSoftwareUnit; for instance, caused by a forbidden Dependency. One AnalyzedSoftwareUnit may include many code constructs that infringe the same or different AppliedRules. Each infringement is registered as a separate Violation, to enable detailed violation reporting.

At the beginning of a compliance check, the instantiations of AnalyzedSoftwareUnit and Dependency with their mutual associations need to be provided by a code-analysis process. Next, each AppliedRule can be checked, based on the traced links between the Modules related to the AppliedRule in the planned architecture and the AnalyzedSoftwareUnits in the implemented architecture. The metamodel contains the data to check AppliedRules of all the common RuleTypes in our classification. Since these RuleTypes focus on different constraints, the required data and behavior to check an AppliedRule differ per RuleType.
4. SRMACC PROTOTYPE
We have validated the feasibility of our approach to provide SRMA support in the context of ACC through a prototype implementation, a test of this prototype, and pilot applications of this prototype. Based on our notion of SRMA-support, we have iteratively designed, developed and applied an open source ACC-tool, named HUSACCT (HU Software Architecture Compliance Checking Tool). These iterations spanned three consecutive years. The first year, we focused on layered architectures, the second year on the provision of all the common module and rule types in the classification, and the third year on extensive support of these types. Each iteration, we used the metamodel to consider, discuss and improve our approach.

Students in computer science participated in the project, of which the results, including an introduction video, are attainable via http://husacct.github.io/HUSACCT/. HUSACCT has been developed in Java and analyzes Java and C# code. The tool provides support to define planned SRMAs, to analyze implemented architectures, and to execute conformance checks. We are using the tool to perform ACCs on professional systems, but we are using the tool also in courses on software architecture to introduce the students in architecture reconstruction, and compliance checking. We are continuing our work on the tool to improve on issues like architecture visualization, accuracy, and scalability.

Support of SRMAs confirm the metamodel does not have to be implemented in the same way as in HUSACCT. For example, the presentation to the user may vary. HUSACCT supports the definition of the planned architecture via a GUI-form, though support via an architecture diagram editor is possible too. As another example, the outcome of a conformance check may be presented in terms of violations, but also in terms of Murphy’s Reflection Model [12] (convergence, divergence and absence).

4.1 Metamodel Implementation
4.1.1 Definition of the Modular Architecture
Figure 6 shows the view “Define Architecture”, used for the creation and maintenance of the planned modular architecture; in this case of the example system depicted in Figure 1. The panel “Module Hierarchy” shows Modules of different ModuleTypes: Layers (e.g., Interaction Layer), Logical clusters (e.g., HiWeb), Components with Facades (e.g., HiManager with HimInterface), and External systems (e.g., Hibernate).

The panel “Rules” shows two generated AppliedRules attached to layer “Service Layer”. These two rules are of the RuleTypes “Back call ban” and “Skip call ban”. Existing rules can be edited and new rules can be specified in a separate panel that pops up when the Edit or Add-button is activated. Exceptions to a rule are also specified in this pop-up panel. To enable traceability, an exception rule is linked to the main rule, as shown in the metamodel by association AppliedRule hasException.

4.1.2 Support of the Semantics of the Types
Extensive support of the semantics of the module and rule types is provided in several ways. First, when a rule is created, only rule types are selectable, which suit the type of the constrained module, (association ModuleType allows RuleType in the metamodel). Second, when an exception is created, only rule types are selectable, which suit to the type of the main rule (association RuleType allowsAsException in the metamodel). Third, when a module is created, applied rules inherent to the module type will be created automatically (association ModuleType hasDefault RuleType in the metamodel). Here, the support is made configurable. For example, to configure that by default layers are allowed to skip call, but not to back call. Fourth, when a module is created of type component, a sub-module of type facade is created automatically, in line with our definition of component.
4.1.3 Module Mapping

Mapping Modules to DefinedSoftwareUnits is supported in panel “Software Units Assigned” within view “Define Architecture”. In the example in Figure 6, package “service”, an AnalyzedSoftwareUnit within the analyzed code, is assigned to Module “Service Layer”. Available software units in the analyzed code are shown and selectable when the button “Add” is activated.

4.1.4 Compliance Checking

HUSACCT is able to check AppliedRules of eleven different RuleTypes. The result of a conformance check are presented in a GUI-browser, in reports, and in diagrams. Conceptually, conformance checks are executed in line with the SRMACC metamodel, but technically, there are differences. An important one is that the analyzed code in HUSACCT is stored conform the FAMIX model [24]. When needed, the concepts AnalyzedSoftwareUnit and Dependency in the metamodel are extracted from the data in the FAMIX model.

4.2 SRMA Test of HUSACCT

As part of a previous study [17], we have designed and implemented a test to assess ACC-tools on their SRMA support. The test includes all common module and rule types from our classification. HUSACCT has been tested with the same testware. The test results demonstrate that HUSACCT provides explicit support for all the module types and for eleven of the twelve rule types (the rule type “Responsibility convention” is not supported, since it requires human interpretation). However, graphical support and support of “External systems” is limited, currently.

4.3 Pilot Applications of HUSACCT

At the end of each of the three development iterations, we performed ACCs with our tool on professional systems at governmental and commercial organizations. In total, six different business information systems of four organizations were subject of an ACC, which we performed with the students participating in the project. The ACCs have yielded interesting results for the customer organizations and have been important for our research. This way, we were able to test our concepts and the tool’s performance in practice, which resulted in new insights and new requirements for the next development iteration of our metamodel and tool.

Some general findings are of interest here. First, semantically rich module types were present in all cases; a confirmation of the relevance of rich sets of ModuleTypes in ACC. Layers dominated the modular architecture of all six case systems, while internal components with access restricting facades were included in two case systems. Second, we encountered and tested rules of nine different rule types; a confirmation of the relevance of rich sets of ModuleTypes in ACC. Third, the customers appreciated the introduction of ACC in their organization, even though in five of the six cases violations were detected (up to 1500).

5. RELATED WORK.

Other studies on ACC have mentioned or proposed the inclusion of support for a specific semantic module type; for instance for layers [14], components [1], [8], or external systems [2]. However, to the best of our knowledge, other studies on ACC have provided neither a comprehensive set of requirements regarding SRMA support, nor a foundational metamodel to address these requirements. Moreover, no metamodel on ACC (with or without SRMA support) has been published before, which is as comprehensive and detailed as our metamodel, which enables support for four modular styles [4]: the decomposition style, uses style, layer style, and generalization style. Koschke [9] has presented an interesting metamodel on ACC according to the Reflection Model approach [12]. The concepts and associations in this metamodel can be mapped to the concepts in our metamodel, but compared to our SRMACC metamodel, Koschke’s model is very abstract, with a smaller set of concepts and without attributes. Furthermore, it is restricted to dependency rules only, and it does not enable differentiations of module and rule types. Other studies (e.g., [10], [19]) present metamodels with even fewer concepts, since they focus on one specific aspect of ACC.

In a previous paper [17] we reported on the results of an SRMA-test on eight academic and commercial ACC-tools. We demonstrated that the SRMA support of these tools was limited: up to explicit support of the semantics of only one module type and up to explicit support of only a few rule types. Consequently, we concluded that all eight tools could improve their support of SRMAs. The same SRMA test has been used to test HUSACCT, as described in the previous section. The test results show that extensive SRMA support is possible on the base of the SRMACC metamodel.

Support of Common Module Types

Five of the eight tested tools in our previous study were not at all supporting semantic differences between modules. Three other tools1 were providing some kind of support for layers, components and facades. SAVE supported the graphical definition of subsystems, layers, components and interfaces, but provided no support of the semantics of these module types. Sonargraph Architect supported the facade pattern and imposed a “Facade convention” rule on defined interfaces. Structure101 supported the concept of layering by imposing “Back call ban” and “Skip call ban” rules on vertically positioned modules. Compared to Sonargraph Architect and Structure101, our approach adds combined support (basic and extensive) for all common module types, and in a consistent way, which allows extension of the set of types. Furthermore, it adds configuration options to tune the semantic support.

Support of Common Rule Types

All eight tested tools in our previous study restricted rule support to dependency rules only, and to simple rule types, like “is allowed to use”, “is not allowed to use”, or “must use”. Compared to these tools, our approach adds explicit support for complex dependency rules and for property rules, including traceable exceptions. These additions are relevant. Rules of the added types are used in practice, like “Naming convention” [25], and may even constitute a significant part of the total rule set, as in case of rule type “Is only allowed to use” [23].

6. CONCLUSION

Architecture compliance checking (ACC) relies on the support of tools to compare the planned architecture with the implemented architecture. We focused our research on support of semantically rich modular architectures (SRMAs) in the context of static ACC. In a previous study, we studied eight ACC-tools and concluded

---

1 SAVE - version 1.7 - iese.fraunhofer.de;
Sonargraph Architect - version 7.0 - hello2morrow.com;
that the support of SRMAs was limited. In this paper, we have presented the SRMACC metamodel for extensive support of SRMAs in the context of static ACC. The metamodel provides the fundamental concepts for: a) the definition of the planned modular architecture, including common module and rule types; b) extensive semantic support of these types; c) module mapping; and d) conformance checking. Therefore, the metamodel may be helpful to enhance existing tools or to develop new approaches. We have validated the feasibility of our approach and metamodel through an open source prototype implementation, a test of this tool (HUSACCT), and pilot applications of this tool.

Future work on SRMA support in the context of static ACC includes ongoing improvement of our HUSACCT prototype, case study research, research on visualization techniques of SRMAs in the context of ACC, research on improvement advice on planned SRMAs, and research on the inclusion of support of other module views and architectural patterns.

In conclusion, our study shows that extensive SRMA support is possible in the context of static ACC. SRMA support widens the scope of ACC and enhances the architectural process. Furthermore, we believe that SRMA support may contribute to the adoption of ACC and consequently to the effectiveness of software architecture in practice and education.

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

8. REFERENCES