HUSACCT: Architecture Compliance Checking with Rich Sets of Module and Rule Types

Leo Pruijt
HU University of Applied Sciences
Utrecht, The Netherlands
leo.pruijt@hu.nl

Christian Köppe
HAN University of Applied Sciences,
Arnhem, The Netherlands
christian.koppe@han.nl

Jan Martijn van der Werf
Sjaak Brinkkemper
University Utrecht
Utrecht, The Netherlands
j.m.e.m.vanderWerf@uu.nl
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 focuses on the module views of architecture and especially on rules constraining the modular elements. This paper presents HUSACCT, a static ACC tool that adds extensive support for semantically rich modular architectures (SRMAs) to the current practice of static ACC tools. An SRMA contains modules of semantically different types, like layers and components, which are constrained by rules of different types. HUSACCT provides support for five commonly used types of modules and eleven types of rules. We describe and illustrate how basic and extensive support of these types is provided and how the support can be configured. In addition, we discuss the internal architecture of the tool.

Categories and Subject Descriptors
D.2 2 [Software Engineering]: Design Tools and Techniques

General Terms
Design, Verification.

Keywords
Software Architecture; Architecture Compliance; Static Analysis

1. INTRODUCTION
A*rchitecture compliance*, is "a measure to which degree the implemented architecture in the source code conforms to the planned software architecture" [2]. Architecture Compliance Checking (ACC) is an approach to bridge the gap between the high-level models of architectural design and the implemented program code. Static ACC does not cover the full width of software architecture, but only the static structure of the software (intended and implemented); in other words, the module views of architecture [1], or *modular architecture*. An intended modular architecture should describe the modular elements, their form (properties and relationships) and rationale, where properties and relationships express architectural rules that constrain a modules' implementation [3]. Modular elements, properties and relationships, are in ACC's center of attention.

Although Shaw and Clements include ACC in 2006 in their list of promising areas [6], the adoption of ACC-tools is still limited [7]. With our research, we intend to contribute to the advancement of current methods and tools. We have focused on ACC support of *semantically rich modular architectures* (SRMAs). We use the term SRMA for an expressive modular architecture description, composed of semantically different types of modules (e.g., layers, subsystems, components), which are constrained by different types of rules, such as basic dependency constraints, constraints related to layers, naming constraints. In practice and literature, many architectures can be labeled as SRMA, since they contain modules with different semantics.

In the last four years, we have iteratively identified requirements regarding SRMA support, studied existing ACC tools, designed a metamodel, developed and tested HUSACCT, and we applied this tool during ACC's on professional systems. In a first publication [4], we presented requirements to SRMA support, and we compared eight commercial and academic ACC-tools on basis of the requirements. We concluded that only limited support was available for SRMAs. Furthermore, that solutions were needed to bridge the gap between modular architectures in software architecture documents on one side, and module and rule models in ACC-tools on the other side.

In a second publication [5], we presented the SRMACC metamodel, whereof the central part regarding SRMA-support is included in Figure 1. It includes the concepts, their attributes and associations, relevant to this paper. As shown in the figure, an SRMA contains *Modules* of different *ModuleTypes*, where *AppliedRules*, each of a certain *RuleType*, may constrain the

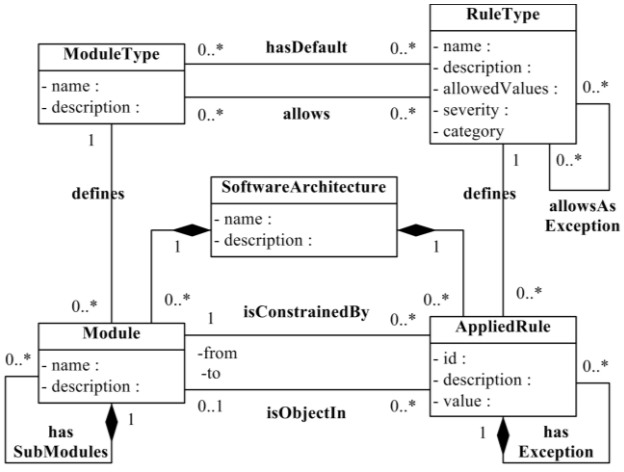

Figure 1. Part of SRMACC metamodel

ASE'14, September 15–19, 2014, Vasteras, Sweden.
 ACM 978-1-4503-3013-8/14/09…$15.00.
http://dx.doi.org/10.1145/2642937.2648624

Modules. For a detailed discussion of the complete metamodel, we refer to [5].

This paper describes and illustrates how HUSACCT provides extensive and configurable SRMA support. The remainder of this paper is structured as follows. Section 2 describes and illustrates the functionality of HUSACCT with the focus on SRMA support. As running example, we use the internal architecture of the tool itself, as it is a suitable example of an SRMA, and it helps to explain how we addressed the most important design challenges. Section 3 describes related work, and Section 4 concludes the paper with the status and outlook of our tool.

2. HUSACCT

HUSACCT (HU Software Architecture Compliance Checking Tool) is a tool that provides support to analyze implemented architectures, define intended architectures, and execute conformance checks. Browsers, diagrams and reports are available to study the decomposition style, uses style, generalization style and layered style [1] of intended architectures and implemented architectures. HUSACCT is free-to-use and open source. It has been developed in Java and analyzes Java and C# source code. The executable and source code are downloadable at http://husacct.github.io/HUSACCT/. An introduction video and documentation are accessible at the same site.

In HUSACCT, an ACC starts with the definition of the modules and rules in the intended architecture. Next, the intended modules are mapped to the implemented software units. Finally, the conformance of the implemented architecture to the intended architecture can be validated. The following subsections follow these steps and explain how HUSACCT provides basic, extensive, and configurable SRMA support. Thereafter, we describe how we addressed some design challenges in the tool's architecture.

2.1 Rich Sets of Module and Rule Types

Basic SRMA support includes the provision of rich sets of module and rule types and the functionality to check rules of these types. In our first SRMA-publication [4], we identified common module and rule types and discussed their grounding in literature. During the development of HUSACCT, we aimed at support of these common types. Currently HUSACCT provides support for five common ModuleTypes and eleven common RuleTypes.

The module and rule types are used in view "Define Intended Architecture", shown in Figure 2. This view supports the creation and maintenance of the intended modular architecture. The panel "Module Hierarchy" shows the *ModuleTypes* currently supported: Component (e.g., *Module* Analyse), Interface (e.g., Interface<Analyse>), Layer (e.g., Presentation), Subsystem (e.g., Common), and External system (e.g., ExternalSystems).

As case, the main part of the architecture of HUSACCT itself is presented. At top-level five components are visible, which all have a layered design internally. As example, three layers are shown within component Analyse. This component is responsible for the analysis of the implemented architecture. The domain layer is responsible for the analyzed data and is designed as a component, with an interface to hide its internals.

The panel "Software Units Assigned" shows that a package and a class are assigned to module Analyse. Inherently, all software units assigned to its submodules are assigned as well. How implemented software units must be assigned to intended modules differs from system to system in practice. Consequently, manual work is required. To enhance the efficiency and accuracy of this work, analyzed software units are made selectable. Once the software units are assigned, defined architecture diagrams can be created, like the ones in Figure 4 and 5, in which defined modules and dependencies (the black, dashed lines) are included.

The panel "Rules" shows that four *AppliedRules* of three different *RuleTypes* are constraining module Analyse. A new rule, together with its exceptions, can be specified in a separate panel that pops up when the Add-button is activated. An exception rule is part of a main rule, as visible in the metamodel. That way it is easy to maintain an overview. For example, the first rule of component Analyse is of type "Façade convention", which bans usage of the component, other than via its interface. Except for a module in component General GUI & Control, that acts as broker.

2.2 Extensive Semantic Support

Extensive semantic support of the module types and rule types prevents inconsistencies in the defined architecture, and it saves work and time. For example, in case of HUSACCT's intended architecture, most rules and all the interfaces are added automatically. HUSACCT provides extensive SRMA support in the following ways.

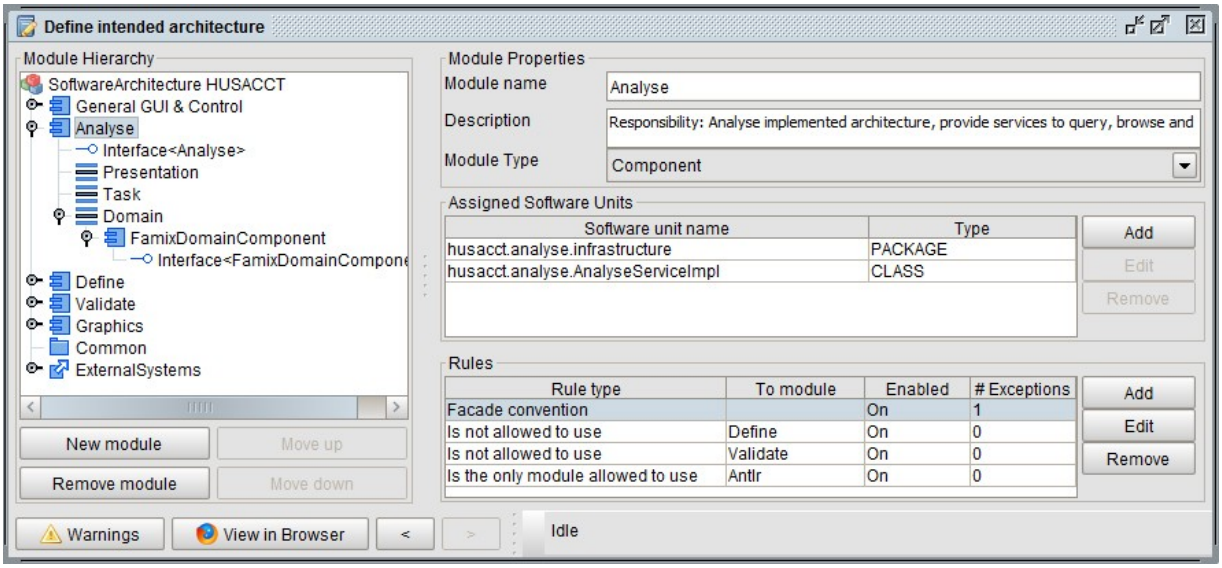

Figure 2. Define intended Architecture, with as case the software architecture of HUSACCT itself

First, when a rule is created, only rule types are selectable which are allowed for the type of the constrained module. For example, in case of module type Layer, all rule types are allowed, except a rule type specific for Components, and rule type "Is allowed to use", which is reserved for exceptions. The list of allowed rule types for module type Layer is shown in Figure 3.

Second, when an exception rule is created, only rule types are selectable which suit to the type of the main rule. For instance, an exception to a rule of type "Façade convention" may only be of type "Is allowed to use".

Third, when a module is created of type Component, a sub-module of type Interface is created automatically; in line with our definition of component.

Fourth, when a module is created, zero, one or more applied rules will be created, based on the associated default rule types. For example, in case of module type Component, an accompanying default rule of type "Facade convention" is generated automatically.

2.3 Configurable Support

ACCs with other tools taught us that non-configurable tool support may result, in certain situations, in invalid violation messages. Reason why we made all rules accessible and incorporated the following configuration options: 1) generated default rules may be disabled (just as user defined rules); 2) exceptions to generated default rules may be specified (just as exceptions to user defined rules); 3) tool-users may configure the default rule types per module type. Figure 3 serves as an example for the third option. It shows that two rule types are assigned as default for module type "Layer". These two rule types together enforce a strict layered model. However, a tool-user is able to configure that in his software architecture a relaxed layered model is standard. Consequently, only an "Is not allowed to back call" rule will be generated when a module of type Layer is added.

2.4 Conformance Checking

Within HUSACCT, the component Validate is responsible for conformance checking. The results of a conformance check are presented in a GUI-browser, in reports, and in diagrams.

Figure 4 and 5 show *Intended architecture diagrams* with the results of a conformance check on the rules of the intended architecture in Figure 2. Violations are shown as red, dotted lines, where the number indicates the number of violations between the two related modules. Details about these violations (like rule type, involved classes, or dependency type) are shown when a line is selected. For example, of the 194 dependencies in Figure 4 from Define to General GUI & Control (the black, dashed line), 26 are violating (the red, dotted line). In this case, all are violating a rule of type "Façade convention". It concerns dependencies to classes within component Analyse, which pass the interface.

Figure 5 shows the violations between the layers within the component Analyse. Five back call violations are visible from

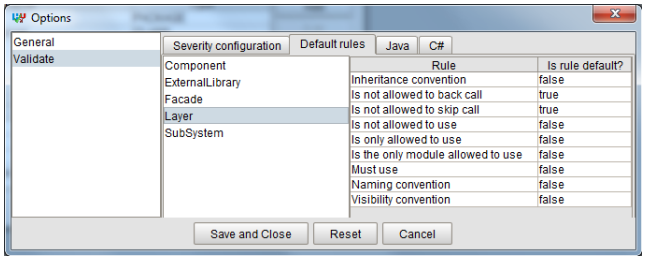

Figure 3. HUSACCT: Configuration of default rule types

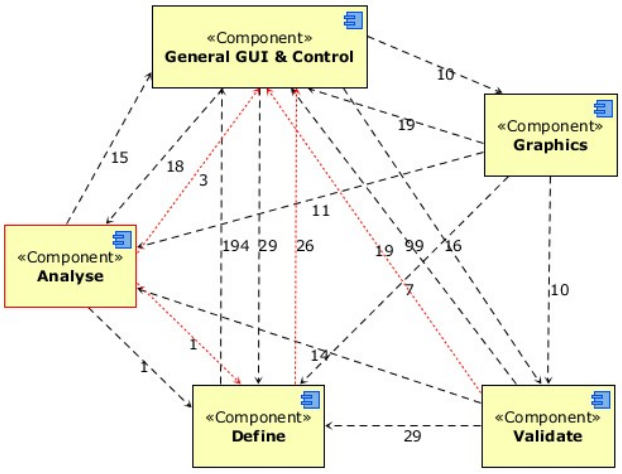

Figure 4. Intended architecture: Top-level components

layer Task to Presentation. The other 17 violations, from Task to Domain, are violations against a "Façade convention" rule. These violations from Task to Domain are shown in more depth in Figure 6, an *Implemented architecture diagram* (zoomed-in on these two layers; some classes and packages are hidden). It shows that two implemented classes make use of the service implementation class and pass the interface class of the FamixDomainComponent. Even worse are the violating dependencies from package analyser directly to package famix.

2.5 Design Challenges

The development of HUSACCT started after a phase of requirement analysis, in which two organizations were involved; the Dutch Tax Administration and InfoSupport. Based on the requirements and the team structure, we had to address design challenges, like: 1) the sets of module and rule types had to be extendible; 2) the tool should work in GUI mode, but also in batch (e.g., daily build process); 3) six development teams had to work concurrently (students in computer science contributed to the development during the first two releases); 4) the set of supported OO programming languages had to be extendible.

To address the first challenge, the SRMACC metamodel was developed, and during the implementation of the concepts, hard-wired dependencies to individual types were prevented as much as possible; for example, by usage of the strategy pattern.

To address the second and third challenges, HUSACCT's software is divided into five components, where each component covers a knowledge area. The components hide their internals, offer services to other components, and exchange data only via data transfer objects. That way, services may be activated via a

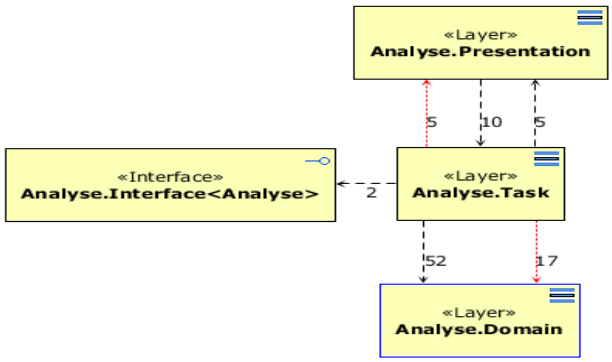

Figure 5. Intended architecture: Analyse component

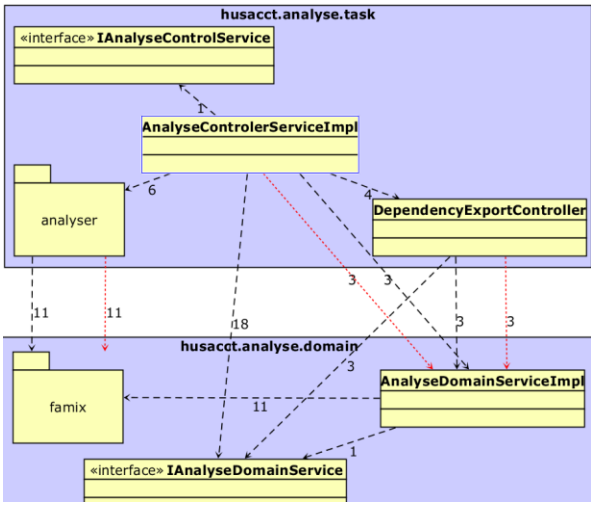

Figure 6. Implemented architecture: Analyse package

GUI or in batch (challenge 2), and each component can be assigned to a separate development team (challenge 3).

To address the fourth challenge, two design decisions were taken. First, ANTLR (www.antlr.org) was selected to read and process the source code, because grammars are available for many programming languages. Second, the FAMIX model [8] was selected to store analyzed code data internally, in a language independent format. Since, after the analysis, all services acquire their data from the FAMIX model, language dependencies are minimized.

3. RELATED WORK

In a previous study [4], we reported on the results of an SRMA-test on eight academic and commercial ACC-tools. We concluded that the tested tools were providing useful support for dependency checking, but only limited support for SRMAs.

Five of the eight tested tools in our previous study were providing only one type of module. Three other tools[1] were providing more types of modules, but only with limited support of their semantics. One tool, SAVE, supported the graphical definition of four module types, but provided no support of their semantics. The two other tools provided semantic support for one type of module: Sonargraph Architect for Interface; and Structure101 for Layer. Compared to these tools, HUSACCT adds semantic support for all its types of modules in a consistent way, which allows extension of the set of module types. Furthermore, it adds configuration options to tune the semantic support.

All eight tested tools in our previous study restricted rule support to dependency rules only, and to simple rule types. Compared to these tools, HUSACCT adds support for property rules (e.g., "Naming convention", "Inheritance convention"), complex dependency rules (e.g., "Is only allowed to use", "Is the only module allowed to use), and exceptions (exceptions are presented as parts of a main rule, not as independent rules).

4. STATUS AND OUTLOOK

HUSACCT provides support to analyze implemented architectures, define intended architectures, and execute conformance checks. HUSACCT distinguishes itself from other ACC tools in its extensive and configurable support of rich sets of module and rule types.

HUSACCT is a free-to-use open source tool, but it is not intended to compete licensed tools. In contrast, we want to contribute to the adoption and quality of ACC. HUSACCT is intended for: 1) introduction of ACC within software development organizations; 2) practical support in courses on software architecture. We use the tool to introduce our students in software architecture, architecture reconstruction, and compliance checking. The tool helps them to relate abstract models to code and to understand the different types of modules and rules.

HUSACCT is in its fourth year of development and each year we performed ACCs with our tool on open source systems and professional systems. The ACCs yielded interesting results for customer organizations and helped us to test and improve the tool. Furthermore, they confirmed the relevance of SRMA support, since in many cases semantically rich module types were present.

Last year, we have worked on the improvement of the accuracy, performance, and usability of the tool, and with considerable results. For instance, analysis and processing time of the source code of HUSACCT version 1.0 (136K lines of code) was reduced from hours in version 2.0 to less than 20 seconds in version 3.2. Future work will focus at first on further improvements of existing functionality, such as the architecture diagrams. Thereafter, we plan to extend the tool with more options for ACC and architecture reconstruction.

In conclusion, HUSACCT shows that extensive and configurable SRMA support is possible. SRMA support widens the scope of ACC and enhances the architectural process. Furthermore, we believe that SRMA support will contribute to the adoption of ACC and consequently to the effectiveness of software architecture in the practice of software engineering.

5. ACKNOWLEDGMENTS

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

6. REFERENCES

[1] Clements, P. et al. 2010. *Documenting Software Architectures: Views and Beyond*. Pearson Education.

[2] Knodel, J. and Popescu, D. 2007. A Comparison of Static Architecture Compliance Checking Approaches. *Working IEEE/IFIP Conf. on Software Architecture* (2007), 12–21.

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

[4] Pruijt, L. et al. 2013. Architecture Compliance Checking of Semantically Rich Modular Architectures: A Comparison of Tool Support. *2013 IEEE International Conference on Software Maintenance* (2013), 220–229.

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

[6] Shaw, M. and Clements, P. 2006. The golden age of software architecture. *IEEE Software*. 23, 2 (2006), 31–39.

[7] De Silva, L. and Balasubramaniam, D. 2012. Controlling software architecture erosion: A survey. *Journal of Systems and Software*. 85, 1 (Jan. 2012), 132–151.

[8] Tichelaar, S. et al. 2000. Famix and xmi. *Proceedings Workshop on Exchange Formats*. (2000), 296–299.

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