Using code analysis tools for architectural conformance checking

Jo Van Eyck, Nelis Boucké, Alexander Helleboogh, Tom Holvoet
DistriNet, Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A, 3001, Leuven, Belgium
jo.vaneyck@student.kuleuven.be
{nelis.boucke, alexander.helleboogh, tom.holvoet}@cs.kuleuven.be

ABSTRACT
Software architecture largely determines the quality attributes a system will exhibit. Architecture design decisions are typically shared to developers in an informal, document-oriented way. As a consequence, it often happens that the actual implementation of a system drifts away from the intended architecture. Checking architecture conformance is important to prevent that the system erodes over time and to safeguard the quality.

However, due to the size of most systems, manual code inspections to ensure architectural conformance are costly and time consuming. Today, several existing tools exist that can be used to assist an architect in this task. In this paper, we investigate several code analysis tools that offer support for Java and compare their capabilities for architectural conformance checking: Architecture Rules, Macker, Lattix DSM, SonarJ, Structure101 and XDepend.

Categories and Subject Descriptors
D.2.11 [Software Engineering]: Software Architectures

General Terms
Design, Verification

Keywords
software architecture, conformance checking, architectural constraints, code analysis tool

1. INTRODUCTION
Whether a system meets its non-functional requirements like modifiability and performance depends on its software architecture. Software architecture is commonly defined as "The structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationship among them." [3]

Software architectures are typically described in an architecture description, typically a document containing several diagrams and textual descriptions. When a system is implemented or changed, it is difficult to assess whether the actual implementation conforms to the architecture [12]. Architectural conformance checking is the verification whether a system conforms to its intended architecture, which is essential to safeguard the quality attributes of the system.

The problem of architectural conformance is widely acknowledged in the field [9, 5]. Due to the size of many systems, performing conformance checking by means of manual code inspections is often practically infeasible.

Several approaches exist to support architecture conformance checking. One approach to achieve conformance is to integrate an Architectural Description Language (ADL) within a general-purpose programming language. An example of this approach for the Java programming language is ArchJava [2]. Since architectural entities are first-class elements in the language and serve as a starting point for implementation, architectural conformance is enforced by the language itself. This way, architecture knowledge is preserved within the code, but it requires the use of a dedicated language to build the system. A less invasive approach to achieve architectural conformance is to leave the existing implementation unchanged, but to use external analysis tools to automatically check architectural conformance. A myriad of tools exists, each with its own terminology, benefits and disadvantages. An overview of other approaches to achieve architectural conformance is discussed in [12].

In this paper, we have a deeper look into a number of code analysis tools that can be used to check static conformance of a system to its architecture. Since there is a wide spectrum of tools, we will only discuss a selection of tools that offer support for Java, namely: Architecture Rules, Macker, Lattix DSM, SonarJ, Structure101 and XDepend. We analyze each of these tools according to a set of criteria and compare them.

This paper is structured as follows. Section 2 introduces the notion of architectural conformance. Since each of the tools uses its own terminology, we will define a common terminology in Section 2.1 in order to make comparisons between the tools easier. Section 3 introduces a working set of architectural constraints that will be used to evaluate the tools in the remainder of the paper. In Section 4 we present the criteria we used to evaluate and compare the different
tools. Next, Section 5 presents the reader with the actual evaluation of the tools. For each of the tools we discuss each of the criteria and show how the concepts of each tool map to the terminology we defined. A concise summary of this comparison can be found in Table 1.

2. ARCHITECTURAL CONFORMANCE

Architectural conformance is achieved when the (as-is) implementation adheres to the envisioned (or to-be) architecture. The architecture is defined in order for the system to meet its non-functional requirements. In order for the implementation to achieve these non-functional requirements, it must conform to the documented architecture. Several techniques exist to check structural architectural conformance between an implementation and the architectural description. Some examples (as discussed in [8]) are:

- **Reflexion models [7]:** The architect defines a model of the to-be architecture in the form of modules, interfaces and dependencies between them. Next, the architect maps implementation elements to modules. This is mostly done using regular expressions: every implementation unit that matches the expression belongs to the module. The output for this class of tools are violations of the defined dependencies.

- **Code Query Languages [11]:** In this approach the architect does not define an architecture in the tool. Instead, he can write arbitrary queries that query the source code directly. For example, he can ask “Give me all the classes Y that use some package X” to check for conformance.

- **Dependency Structure Matrices (DSM) [10]:** In this approach, the existing code-base is analyzed and a DSM is generated. In its most basic form, rows and columns of the DSM represent implementation units (like classes or packages). A marked cell in row $R$ and column $C$ shows that implementation unit $R$ uses implementation unit $C$. The architect can check for architectural conformance by comparing the DSM of the implementation with the DSM of the envisioned architecture.

2.1 Conformance checking terminology

In this section we introduce some self-defined terminology related to conformance checking. We will use these terms in the remainder of the paper to show how common concepts in the various tools relate to each other. Figure 1 contains a class-diagram relating all the concepts. We distinguish the following concepts:

- **ImplementationEntity:** An element of the implementation. Each of these ImplementationEntities is mapped to one or more ArchitecturalEntities.

- **ArchitecturalEntity:** This can either be a Module or an Interface.

- **Module:** An implementation unit of software that provides a coherent unit of functionality[4]. Each Module can expose zero to many Interfaces.

- **Interface:** A coherent set of services a Module makes publicly available.

- **DependencyConstraint:** A usage constraint between ArchitecturalEntities.

- **CanUseConstraint:** A constraint between a Module $X$ and an ArchitecturalEntity $Y$, defining that $X$ can make use of $Y$.

- **CannotUseConstraint:** A constraint stating that one Module cannot use another Module.

ImplementationEntity, Architecturalentity and DependencyConstraint are top-level concepts. Each of these concepts can be specialized: an ImplementationEntity can for example be a class, a java package or a jar-file. ArchitecturalEntities can be Modules, Interfaces, layers or any other structural entity in an architectural description that has a representation in the implementation.

Typical usage of a conformance checking tool will proceed as follows: First, the architect defines the to-be architecture in terms of ArchitecturalEntities and DependencyConstraints. Next, he defines a mapping between ImplementationEntities and ArchitecturalEntities. Finally, the tool will check whether all DependencyConstraints defined between ArchitecturalEntities are satisfied by the corresponding ImplementationEntities.

3. EXAMPLES OF CONSTRAINTS

We will use an implementation of the well-known Model-View-Controller (MVC) architectural pattern as a case
An explicit Interface class. The architectural level. The Module age in the implementation represents a study. Figure 2 shows the used implementation. Each pack-

Figure 2: An illustrative implementation of the MVC pattern

module can only be used through an explicit Interface, which is implemented by the ModuleInterface class. The Module module implicitly uses the View interface through an implementation of the Observer design pattern.

In this example we distinguish several types of constraints:

- C1: Module X can-use Module Y: We adapt the definition of usage from [4].

  "Element A is said to use element B if A’s correctness depends on a correct implementation of B being present”

The example contains only one example of this type of constraint: Controller can-use View.

- C2: Module X can-use Module Y via Interface I: This is a special case of the previous class of constraints. X can use Y, but only through the classes implementing the interface I. Examples are Controller can-use Model via ModelInterface and View can-use Model via ModelInterface.

- C3: Module X can-indirectly-use Module Y: The definition of usage here is more subtle than in C1 and C2. In this case, X can use services of Y at run-time, but X contains no direct references to Y. An example of this kind of constraints are callbacks from subjects to the observers in the Observer pattern [6]. The example contains one such constraint: Model can-indirectly-use View.

- C4: Module X cannot-use Module Y: The example contains several cannot-use constraints: View cannot-use Controller and Model cannot-use Controller.

4. EVALUATION CRITERIA

We used the following criteria to evaluate the tools on their usefulness for architectural conformance checking:

- Classification: Tools are put in the following categories: Reflection models (RM), Code Query Languages (CQL) or DSM-based.

- Input files: Some tools operate on java-, class- or jar-files or a combination of them. Sometimes not all source code is readily available (e.g. third-party software), but the architect also wants to check the dependencies with these pieces of software.

- Used concepts: Every tool uses its own terminology to describe software architectures. In these paragraphs we briefly explain the used terms and relate them to the terms described in Section 2.1.

- Input of the architectural description: Some tools provide functionality to graphically input an architecture, where other tools require the architect to describe his architecture in an XML-file. This criterion has impact on two aspects: ease of use and level of detail. The level of detail determines between which architectural entities dependencies can be described. Some tools allow the architect to define dependencies between modules or high-level layers. Sometimes interfaces can be explicitly modelled so that a higher level of detail can be achieved.

- Supported dependency-constraints: Some tools only support cannot-use dependency constraints, other tools also provide support for can-use dependencies. Tools of the first category often define implicit can-use dependency constraints, namely all possible dependencies that are not covered by the set of cannot-use constraints. In some cases explicit modelling of can-use constraints can provide more insight: when can-use constraints are implicit, there is no clear distinction between intended can-use constraints or cannot-use constraints the architect just forgot to model. We will list the supported types of constraints as defined in Section 3.

- Output: All tools give some representation of violated constraints. This can range from a simple listing of violated constraints to a navigable graphical representation of the as-is architecture that shows all violations. The level of detail can also be a determining factor for this criterion. Some tools only show which constraints are violated, others give the exact line of code that breaks the dependency.

- Mapping between the implementation and architectural description: Most tools support the usage of regular expressions as explained above, but other tools provide an easier to use drag-and-drop system.

- Consistency checking: The possibility of creating detailed architectures comes with a cost: the expressivity of some tools allow inconsistent architectures to be defined. For example, the architect can define contradicting dependency- constraints. In these paragraphs we discuss support for consistency checking. Some tools offer good support, other tools leave this responsibility to the architect.

- Integration in the development cycle: Some tools check for conformance during the build-process. Others are standalone applications and require the architect to manually start the analysis, we will call this option external. Some tools provide integration with
common IDE’s like Eclipse so that developers are continually aware of the to-be architecture and violated constraints are reported at development time.

5. COMPARISON OF THE TOOLS

In this section, we will discuss the criteria presented in Section 4 for each of the following tools: Architecture Rules, Lattix LDM, Macker, SonarJ, Structure101 and XDepend.

5.1 Overview

Table 1 presents an overview of this section. In the column Supported constraints, a regular notation means the constraints can be defined in a straightforward manner. If the constraint is placed between brackets, it is harder but still possible to check this type of constraints in the tool.

5.2 Architecture Rules

Assert Your Architecture! […] Architecture Rules \(^{1}\) leverages an xml configuration file and optional programmatic configuration to assert your code’s architecture via unit tests or ant tasks.

- Classification: RM
- Input files: class-files
- Used concepts: Architectural Rules does not support the notion of ArchitecturalEntities. Instead, it operates on the level of ImplementationEntities, namely Java packages. As for DependencyConstraints, only CannotUseConstraints, which are called rules can be defined. Each rule lists a number of from-packages \(F\) and to-packages \(V\) (called violations). A rule is violated when any package depends on a violation.
- Input of the architectural description: All DependencyConstraints can be defined in an xml-file or programatically configured in Unit tests.
- Supported dependency-constraints: Only constraints of type C4.
- Output: All violated DependencyConstraints.
- Mapping between the implementation and architectural description: There is no need for a mapping since DependencyConstraints are defined in terms of ImplementationEntities.
- Consistency checking: Since there is no notion of ArchitecturalEntities and only CannotUseConstraints can be defined, little room is left to describe inconsistent architectures. Only one inconsistency can be defined, namely ImplementationEntity \(X\) cannot-use itself. Architecture Rules detects this inconsistent DependencyConstraint.
- Integration in the development cycle: Architecture Rules is run by implementing a Unit test (development- or build-time), and as such it might be used in a test-driven environment.

Architecture rules was specifically defined to check architectural conformance. However, the lack of ArchitecturalEntities forces the architect to define constraints in terms of lower-level ImplementationEntities. Also, the range of supported constraints is limited, since only cannot-use constraints can be defined.

5.3 Macker

Macker \(^{2}\) is a build-time architectural rule checking utility for Java developers. It’s meant to model the architectural ideals programmers always dream up for their projects, and then break – it helps keep code clean and consistent.

- Classification: RM
- Input files: class-files
- Used concepts: Macker does not directly use concepts such as Module or Interface. Instead, Macker makes use of regular expressions called Patterns. The architect can for example define Patterns that match all classes in an Interface or Module. In Macker DependencyConstraints are called Access Rules. An Access Rule defines a Can- or CannotUseConstraint between two Patterns.
- Input of the architectural description: The Patterns and Access Rules are defined in an xml-file.
- Supported dependency-constraints: Macker explicitly supports constraint types C1 and C4. If the architect makes clever use of the Pattern mechanism, constraints of type C2 can also be checked. There is no support for constraints of type C3.
- Output: Macker generates a xml-report which contains all the violated Access Rules. For every violation, the two classes that cause the violation are shown.
- Mapping between the implementation and architectural description: The mapping is achieved through regular expressions called Patterns. Since Patterns provide a general way to group ImplementationEntities on an architectural level, constraints can be defined in a great level of detail.
- Consistency checking: Macker supports very little consistency checking. For example, it is possible map the same ImplementationEntity to multiple ArchitecturalEntities and define a CannotUseConstraint between them, hereby creating a DependencyConstraint that will always be violated.
- Integration in the development cycle: Macker can be added as an ant-task, so violations are checked at build-time.

Macker is also specifically designed to check architectural conformance. The use of Patterns give the architect a great level of flexibility. The lack of consistency checking shifts this responsibility back to the architect. Development of Macker has stopped since 2003.

\(^{1}\)http://wiki.architecturerules.org/

\(^{2}\)http://www.innig.net/macker/


5.4 Lattix LDM

Lattix LDM \(^3\) enables you to create Dependency Models of your software systems [...]. With Lattix LDM, you can analyze your architecture in detail, edit the structure to create what-if and should-be architectures, and then create Design Rules to formalize and communicate that architecture to your entire development organization.

- **Classification:** DSM
- **Input files:** class- and jar-files
- **Used concepts:** LDM uses the term Subsystem for Modules. It has no concept to explicitly describe an Interface. DependencyConstraints are called Rules.
- **Input of the architectural description:** The architect is initially presented with a DSM generated from the existing implementation. The architect can modify this DSM by grouping ImplementationEntities in user-defined ArchitecturalEntities. The architect can define a DependencyConstraint between every two ArchitecturalEntities in the modified DSM.
- **Supported dependency-constraints:** Constraints of type C1 and C4 are easily defined using Can- and Cannot Use Rules respectively. By putting all the classes belonging to an Interface in a separate Subsystem, constraints of type C2 can be checked. LDM has no support for constraints of type C3.
- **Output:** Lattix colors cells whose DependencyConstraints are violated red in the DSM. It is also possible to generate a so-called violations report that lists all the DependencyConstraints that are violated in the implementation.
- **Mapping between the implementation and architectural description:** The mapping is achieved by manually assigning ImplementationEntities to user-defined ArchitecturalEntities using a drag-and-drop system.
- **Consistency checking:** There is no need to apply any form of consistency checking since inconsistent architectural descriptions cannot be created. Every ImplementationEntity can be assigned to exactly one ArchitecturalEntity and only one DependencyConstraint can be defined between two ArchitecturalEntities.
- **Integration in the development cycle:** External: Lattix LDM is used as a stand-alone application.

Lattix LDM is not specifically developed to check architectural conformance, but instead offers functionality to perform a DSM analysis of existing implementations. In order to create a to-be architecture in LDM an existing implementation is required, since the DSM of the implementation acts as a starting point. This is a rather unnatural order, since usually the to-be architecture is defined before actual implementation starts.

\(^3\)http://www.lattix.com/products/ldm-ldv

5.5 SonarJ

SonarJ \(^4\) is designed to simplify software architecture and dependency management. You begin by defining the intended logical architecture of your system and map it to your code. [...] Once defined, our Eclipse plugin checks every code change for rule compliance.

- **Classification:** RM
- **Input files:** class- and java-files
- **Used concepts:** SonarJ offers a wide variety of options to represent ArchitecturalEntities. It offers concepts such as Subsystems which correspond to Modules as defined in Section 2.1. It further supports grouping of these Subsystems into Layers (horizontal) or Vertical slices (vertical groupings). SonarJ is the only discussed tool that explicitly supports Interfaces. DependencyConstraints are called Dependencies.
- **Input of the architectural description:** The architect graphically enters the to-be architecture by defining ArchitecturalEntities and defines DependencyConstraints between them.
- **Supported dependency-constraints:** Constraints of type C1 and C4 can be defined in a straightforward way by using Dependencies. In SonarJ, C2 constraints can also easily be defined because of the explicit support for Interfaces.
- **Output:** A list of all the violated DependencyConstraints.
- **Mapping between the implementation and architectural description:** Mapping is achieved by defining regular expressions for every ArchitecturalEntity, or by simply drag-and-dropping ImplementationEntities into ArchitecturalEntities.
- **Consistency checking:** SonarJ explicitly offers good support for consistency checking. The tool reports contradicting or redundant DependencyConstraints, warns the architect when not all ImplementationEntities are mapped to an ArchitecturalEntity, etc.
- **Integration in the development cycle:** SonarJ can be used as a stand-alone application (external), as an Ant-tasks or Maven plugin (build-time) or even as an Eclipse-plugin (development-time). The Eclipse-plugin warns the developer during development when DependencyConstraints are violated.

SonarJ was specifically developed to support architectural conformance checking. It offers the broadest range of supported ArchitecturalEntities.

5.6 Structure101

Structure101 \(^5\) lets teams develop simpler code-bases with defined architectures, so they get more done in less time. Simple as that.

\(^4\)http://www.hello2morrow.com/products/sonarj
\(^5\)http://www.headwaysoftware.com/
• Classification: RM, DSM

• Input files: class- and java-files

• Used concepts: Structure101 uses the term Cells for ArchitecturalEntities. Cells can be hierarchical. These Cells are grouped in layers. DependencyConstraints are not explicitly defined by the architect, but are instead derived from the layered architecture. Cells can only use Cells in a lower layer. Exceptional upward CanUseConstraints can be defined and are called overrides. Violations of DependencyConstraints are called dependency breakouts.

• Input of the architectural description: The architect defines a layered architecture by drawing ArchitecturalEntities and placing them in a diagram. DependencyConstraints are derived from this diagram in the following way: Placing cell A above cell B defines the following DependencyConstraints: A can-use B and B cannot-use A. The architect can explicitly allow usage of higher-level cells by defining overrides. An override from lower cell B to cell A on a higher level represents the following constraint: B can-use A.

• Supported dependency-constraints: Constraints C1 and C4 can be easily defined by positioning the relevant ArchitecturalEntities in the architectural diagram. Constraints of type C1 are defined by placing the to-cell below the from-cell or by defining an override. C4 can be defined by placing the to-cell above the from-cell. Constraints of type C2 are less straightforward to define: a separate cell can be defined that is implemented by all ImplementationEntities representing the Interface. By placing this new cell between the from- and to-cell in the diagram constraints of type C2 can be defined. There is no support for constraints of type C3.

• Output: Structure101 shows all dependency breakouts on the architectural diagram. The tool also creates a DSM of the implementation.

• Mapping between the implementation and architectural description: The mapping is achieved through regular expressions. This process can be simplified by manually dragging ImplementationUnits into their corresponding ArchitecturalEntities.

• Consistency checking: Structure101 allows several regular expressions to match the same ImplementationEntity, but assigns it to only one ArchitecturalEntity. The user receives no warning about this inconsistency.

• Integration in the development cycle: Versions of Structure101 include a stand-alone application (external), as part of the build-process (build-time) or as an IDE plugin (development-time).

Structure101 is specifically developed to support architecture conformance checking. It is one of the few tools that support multiple approaches (RM and DSM) to check conformance. Since not all software architectures are organized in a layered fashion, this can sometimes lead to strange representations in Structure101 as the tool only supports layered architectures.

5.7 XDepend

XDepend \(^6\) lets you extract, visualize, seek and control the structure of your applications and frameworks.

• Classification: DSM, CQL

• Input files: class-files in jars

• Used concepts: Since XDepend does not support the Reflexion Model approach, the architect cannot explicitly model the to-be architecture. Consequently, there is also no concept of ArchitecturalEntities or mappings. Instead, the architect has to reason directly about ImplementationEntities.

• Input of the architectural description: No to-be architecture can be defined.

• Supported dependency-constraints: Constraint types C1, C2 and C4 can be checked using CQL queries or by inspection of the DSM. Since there is no representation of ArchitecturalEntities however, the constraints have to be specified on the lower level of the ImplementationEntities themselves. This can make conformance checking more labour-intensive, as the following example illustrates: Violations of the constraint Controller can-use Model via ModelInterface can be checked by the following CQL queries:

CQL1: SELECT PACKAGES WHERE IsDirectlyUsing "model"

CQL2: SELECT TYPES WHERE IsDirectlyUsing "model.ModelPrivate"

CQL1 returns all the package which directly use the package model. It returns the following result: \{controller, view\}. CQL2 returns all classes or interfaces that use the ModelPrivate class. It returns \{ModelInterface\}. Both view and controller make use of the model package (CQL1) and they don’t use any classes private to the package (CQL2), thus the constraint is satisfied. Constraints of type C3 cannot be checked with XDepend.

• Output: The DSM of the implementation can be navigated to manually inspect dependencies between ImplementationEntities. Constraints can be automatically checked by using specific CQL queries as discussed above.

• Mapping between the implementation and architectural description: Not applicable for this tool.

• Consistency checking: Since there is no representation of the to-be architecture in XDepend, consistency checking does not apply.

• Integration in the development cycle: Analysis can be done at build-time or by choice of the architect (external): XDepend is a standalone tool that can be used through a GUI or by command-line interface. A Maven plugin (build-time) is also developed.

\(^6\)http://www.xdepend.com/
XDepend was not developed specifically for architectural conformance checking, but instead offers a broader range of code analysis. The lack of a representation for ArchitecturalEntities forces the architect to perform conformance checking on a lower level of abstraction. Both a deeper understanding of the implementation and strict naming conventions regarding package structure are required when using XDepend for conformance checking.

6. DISCUSSION AND CONCLUSION

Architectural conformance is crucial to ensure that a system’s implementation respects the constraints imposed by its architecture. Due to the size and frequent evolution of many software systems, checking architecture conformance is a tedious and costly task when performed manually.

In this paper we compared a number of available code analysis tools that can be used to check architectural conformance. Table 1 presents a summary of the discussed tools. The tools we investigated can alleviate some of the burden of manual code inspections. We applied the tools to a limited example. An interesting avenue for future research is to investigate the scalability how the various tools scale when being applied to larger systems. So far, we have only focused on the structural aspect of an architecture, namely dependencies between entities. Tool support for other architectural aspects that can be derived from the source code (like deployment) could also be investigated. From our investigation we draw the following observations. First, the analysis tools we evaluated can be used to check static conformance only: none of the tools is capable of checking dependency constraints of type C3, which can occur quite often in object-oriented designs. Work on checking architectural conformance between the implementation and its runtime architecture has been done by Aldrich[2, 1]. Second, the tools we investigated have limited support to represent custom architectural entities. The tools mostly rely on predefined concepts such as modules or layers, which may or may not correspond to the abstractions used to express the system’s architecture. Finally, most of the tools only show which constraints are violated, but fail to pinpoint the precise location in the code where the violation occurs. This means that, even if a large part of the conformance checking is automated, an architect still has to rely on some degree of manual inspection of the code to trace the violations.

7. REFERENCES

<table>
<thead>
<tr>
<th>Criterium</th>
<th>Macker</th>
<th>Lattix LDM</th>
<th>SonarJ</th>
<th>Structure101</th>
<th>XDepend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classification</td>
<td>RM</td>
<td>DSM</td>
<td>RM, DSM</td>
<td>RM, DSM</td>
<td>CQL, DSM</td>
</tr>
<tr>
<td>Input files</td>
<td>class</td>
<td>class, java (workspace)</td>
<td>class, java</td>
<td>class</td>
<td>jar</td>
</tr>
<tr>
<td>Used concepts</td>
<td>package</td>
<td>package, class, interface</td>
<td>package, class</td>
<td>package, class</td>
<td>package, class, interface</td>
</tr>
<tr>
<td>- ImplementationEntities</td>
<td>package</td>
<td>package, class, interface</td>
<td>package, class</td>
<td>package, class, interface</td>
<td>package, class, interface</td>
</tr>
<tr>
<td>- ArchitecturalEntities</td>
<td>implicitly (patterns)</td>
<td>subsystem, layer, cell, layer</td>
<td>subsystem, layer</td>
<td>subsystem, layer, cell, layer</td>
<td>N/A</td>
</tr>
<tr>
<td>- DependencyConstraints</td>
<td>rule</td>
<td>rule</td>
<td>rule</td>
<td>rule</td>
<td>N/A</td>
</tr>
<tr>
<td>Input of architectural description</td>
<td>rules defined</td>
<td>patterns</td>
<td>modification of</td>
<td>graphically</td>
<td>graphically (layed only)</td>
</tr>
<tr>
<td>Supported constraints</td>
<td>C4</td>
<td>C1, (C2), C4</td>
<td>C1, (C2), C4</td>
<td>C1, (C2), C4</td>
<td>C1, (C2), C4</td>
</tr>
<tr>
<td>Output</td>
<td>List of violated rules</td>
<td>violated access rules</td>
<td>list of violations</td>
<td>dependency breakouts on diagram</td>
<td>arbitrary query results, list of violations</td>
</tr>
<tr>
<td>Mapping</td>
<td>N/A</td>
<td>regular expressions (patterns)</td>
<td>manual assignment</td>
<td>regular expressions, manual assignment</td>
<td>N/A</td>
</tr>
<tr>
<td>Consistency Checking</td>
<td>basic, but little room for inconsistencies</td>
<td>very little</td>
<td>not necessary</td>
<td>good</td>
<td>good, but room for inconsistencies</td>
</tr>
<tr>
<td>Integration in development-cycle</td>
<td>development-, build-time</td>
<td>build-time</td>
<td>external</td>
<td>development-, build-time, external</td>
<td>external</td>
</tr>
</tbody>
</table>