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
Architectural conformance checking verifies whether a system conforms to its intended architecture, which is essential to safeguard the quality attributes of the system. Due to the size of many systems, performing conformance checking by means of manual code inspections is often practically infeasible. Code analysis tools can be used to automatically check architectural conformance.

In this paper, we investigate several code analysis tools that offer support for Java and compare them on their usefulness 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
Software architecture largely determines the quality attributes a system will exhibit. Architectural design decisions are typically shared with developers in an informal way, such as diagrams and textual descriptions. As a consequence, it often happens that over time the actual implementation of a system drifts away from the intended architecture.

Architectural conformance checking verifies 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 [2]. 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 automatic architectural 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 [1]. 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, architectural 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 benefits and disadvantages. Several approaches to check architectural conformance are discussed in [3, 4, 6].

In this paper, we investigate several code analysis tools that offer support for Java and compare them on their usefulness for architectural conformance checking: Architecture Rules, Macker, Lattix DSM, SonarJ, Structure101 and XDepend.

For an in-depth overview of the comparison, we refer the reader to [5].

2. COMPARISON
We evaluated the tools based on criteria defined in [5]. Table 1 summarizes the results.

3. DISCUSSION AND CONCLUSIONS
From our comparison we draw the following observations:
The analysis tools we evaluated can be used to check static conformance only: none of the tools are capable of checking indirectly-uses dependencies, which only manifest themselves at runtime. These dependencies can however occur quite often in object-oriented designs. As a consequence, these tools are not suited to verify dynamic aspects of the system.

A problem with using these tools to check for architectural conformance is that some architectural knowledge (namely the definition of modules and their allowed interdependencies) is duplicated: it is described once in the architectural description and also in the specifications which serve as input for these tools. This gives rise to a new problem of maintaining the specifications and keeping them consistent with the architectural description. Solutions to this problem would be to either use the full architectural description as input for these tools (avoiding duplication), or to store this architectural knowledge closer to the implementation itself.
Table 1: Overview of the comparison

<table>
<thead>
<tr>
<th>Criterium</th>
<th>Architecture</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>RM</td>
<td>DSM</td>
<td>RM</td>
<td>RM, DSM</td>
<td>CQL, DSM</td>
</tr>
<tr>
<td>Input files</td>
<td>class</td>
<td>class</td>
<td>class, java (workspace)</td>
<td>class, java</td>
<td>class, java</td>
<td>jar</td>
</tr>
<tr>
<td>Used concepts</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Implementation-Entities</td>
<td>package</td>
<td>class</td>
<td>package, class, interface</td>
<td>class</td>
<td>package, class, interface</td>
<td></td>
</tr>
<tr>
<td>-Architectural-Entities</td>
<td>package</td>
<td>implicitly (patterns)</td>
<td>subsystem</td>
<td>subsystem, layer, vertical slice, named interface</td>
<td>cell, layer</td>
<td>N/A</td>
</tr>
<tr>
<td>-Dependency-Constraints</td>
<td>rule</td>
<td>access rule</td>
<td>rule</td>
<td>dependency</td>
<td>implicit: diagram, explicit: override</td>
<td>N/A</td>
</tr>
<tr>
<td>Input of architectural description</td>
<td>rules defined in xml or unit test</td>
<td>patterns and access rules in xml</td>
<td>modification of implementation-based DSM</td>
<td>graphically</td>
<td>graphically (layered only)</td>
<td>N/A</td>
</tr>
<tr>
<td>Supported constraints</td>
<td>cannot-use</td>
<td>can-use, (can-use-via), cannot-use</td>
<td>cannot-use, cannot-use</td>
<td>cannot-use, (can-use-via), cannot-use</td>
<td>cannot-use, (can-use-via), cannot-use</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>List of violated rules</td>
<td>violated access rules</td>
<td>icons in DSM, list of violations</td>
<td>list of violations</td>
<td>dependency breakouts on diagram, DSM</td>
<td>arbitrary query results, DSM</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>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>
<td>N/A</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>development-, build-time, external</td>
<td>build-time, external</td>
</tr>
</tbody>
</table>

this way, whenever the architecture (and later the implementation) is updated, the maintainability problem is reduced because of the closer link between architectural knowledge and code.

Also, each of the tools uses its own terminology. Interfaces and other commonly used concepts have no direct counterpart in most of the tools. If an architect wishes to use these tools, he either has to define the architecture using the more limited set of tool-specific concepts. This unnecessarily restricts the architect while modelling his architecture and also hampers the transfer of architectural knowledge, since the precise semantic information of a concept (like an interface) gets lost if it is defined in terms of a more general tool-specific concept. Another option would be to define a translation between common concepts (like interfaces) and tool-specific concepts. In this way, the architect can still model his architecture using common concepts, but the translation step takes extra effort.

4. REFERENCES


