SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2017; **47**:273–309
Published online 13 July 2016 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2421

The accuracy of dependency analysis in static architecture compliance checking

Leo Pruijt[1,*,†], Christian Köppe[2], Jan Martijn van der Werf[3] and Sjaak Brinkkemper[3]

[1]*HU University of Applied Sciences, Utrecht, The Netherlands*
[2]*HAN University of Applied Sciences, Arnhem, The Netherlands*
[3]*University Utrecht, Utrecht, The Netherlands*

SUMMARY

Architecture compliance checking (ACC) is an approach to verify conformance of implemented program code to high-level models of architectural design. Static ACC focuses on the modular software architecture and on the existence of rule violating dependencies between modules. Accurate tool support is essential for effective and efficient ACC. This paper presents a study on the accuracy of ACC tools regarding dependency analysis and violation reporting. Ten tools were tested and compared by means of a custom-made benchmark. The Java code of the benchmark testware contains 34 different types of dependencies, which are based on an inventory of dependency types in object oriented program code. In a second test, the code of open source system FreeMind was used to compare the 10 tools on the number of reported rule violating dependencies and the exactness of the dependency and violation messages. On the average, 77% of the dependencies in our custom-made test software were reported, while 72% of the dependencies within a module of FreeMind were reported. The results show that all tools in the test could improve the accuracy of the reported dependencies and violations, though large differences between the 10 tools were observed. We have identified 10 hard-to-detect types of dependencies and four challenges in dependency detection. The relevance of our findings is substantiated by means of a frequency analysis of the hard-to-detect types of dependencies in five open source systems. © 2016 The Authors. *Software: Practice and Experience* Published by John Wiley & Sons, Ltd.

Received 20 May 2015; Revised 25 May 2016; Accepted 26 May 2016

KEY WORDS: Software architecture; architecture compliance; dependency analysis; benchmark test

1. INTRODUCTION

Software architecture is of major importance to achieve the business goals, functional requirements, and quality requirements of a system. In practice, a variety of architectural models is used to describe how systems are structured and how the components interact. However, the 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 [1]. 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 architectural erosion [2]. Knodel and Popescu defined architecture compliance as 'a measure to which degree the implemented architecture in the source code conforms to the planned software architecture' [3]. The terms architecture compliance and architecture conformance are both used in literature.

Many tools and techniques are available to analyze a software system and to reconstruct, visualize, check, or restructure its architecture [4]. In our study, we focus on tools supporting static ACC, which analyze software without executing the code. These tools, which we label as static

*Correspondence to: Leo Pruijt, HU University of Applied Sciences, Utrecht, The Netherlands.
†E-mail: leo.pruijt@hu.nl

ACC-tools, focus on the modular structure in the source code. The tools identify structural elements, such as packages and classes, and use-relations between these elements, such as an invocation of a method or access of an attribute. To support ACC, the tools provide facilities to: (i) define modular elements and rules restricting these elements and their relationships; (ii) check the compliance to these rules; and (iii) report violations to these rules. For example, a tool should report a violation if a method-call in the code from class A to B corresponds with a dependency from module X to module Y in the planned architecture, while a rule exists that forbids such a dependency.

Although static ACC-tools predominantly check for the same kind of inconsistencies between the implemented and intended modular architecture, only a few studies have compared these tools. Previous studies have identified large differences in terminology and approach [3, 5, 6]. For instance, the study of Passos et al. 5 identified and evaluated three techniques of static architecture checking. Furthermore, they explored the effectiveness and usability of three supporting tools by executing tests, based on a simple system with a basic architecture. Our research follows Passos et al. We aspire to contribute to the evolution of ACC, motivated by the notion that the adoption of ACC-tools is still limited [2, 7]. Further research is necessary to advance and improve current methods and tools 8. We focus on the effectiveness of ACC, because it is of primary interest to practitioners and researchers. The 'Quality in use model' of ISO 25010 9 defines effectiveness as 'accuracy and completeness with which users achieve specified goals'. In another study, we investigated the functional completeness of ACC support; more specifically, the support of semantically rich modular architectures in the context of ACC 10.

In this study, we focus on the accuracy of ACC support, which we scoped to the main question: *How accurate do ACC-tools report dependencies and violations against dependency rules?* Accuracy is relevant, because emerging trends are to use code analysis throughout the coding process [11], and to extract and update architectural views continuously [8]. Although static analysis is theoretically not difficult, the complexities of modern programming languages significantly impede source code analysis [11]. Nevertheless, unlike performance, accuracy of ACC does not receive much attention. The accuracy of dependency and violation reporting is omitted in many papers on ACC approaches, e.g. [1, 12–18], and when discussed, it is restricted to false positives only (the definitions of the terms false positive and false negative in this context is provided in section 4.1). To operationalize our main question, we decomposed it into the following research questions:

RQ1 Do ACC tools find all the dependencies between modules in the software (no false negatives)?
RQ2 Do ACC tools report all the violating dependencies in the software (no false negatives)?
RQ3 Do ACC tools report non-violating dependencies as violations (false positives)?
RQ4 Do ACC tools report the exact type and location of violations and dependencies?
RQ5 Are there types of dependencies, which proved hard-to-detect by several tools?

To answer these questions, we inventoried commonly used types of dependencies that can be established in object oriented program code. Next, we developed a custom-made test application in Java that included these types of dependencies and an accompanying test script (we will use the working title 'Benchmark test' to refer to this test software and test script). After completion, we used the Benchmark test to assess 10 ACC-tools. In addition, we selected the open source system FreeMind and used its code to examine the same tools on their ability to report dependencies and violations accurately.

The contribution of this study is threefold.

- We present two Java-based tests, the Benchmark test and the FreeMind test, to assess the ability of a tool to detect dependencies of 34 different types. The testware of these tests is freely available. Instructions on how to perform the tests are published as a technical report [19]. The required documents and code files are available at the following address: https://github.com/SaccToolTests/SacctAccuracyTest.
- We present the results of the tests on 10 commercial and non-commercial ACC-tools with respect to the accuracy of dependency detection and the exactness of the dependency and violation messages.
- We identify 10 types of dependencies that proved hard-to-detect by several tools in the tests. Furthermore, we identify challenges in dependency detection, and we substantiate the relevance of these challenges by means of analysis data of five open source systems.

DOI: 10.1002/spe

This paper extends earlier work [20] in which we have reported on the accuracy of dependency analysis and violation reporting of seven ACC-tools. *First*, we have improved the testware of the Benchmark test and FreeMind test, and made both sets of testware utilizable for other researchers. *Second*, we extended the Benchmark test at the point of the detection of local variables, and we retested all tools at this point. *Third*, we describe and illustrate the dependency types in the tests in more detail and provide an improved definition of indirect dependencies in the context of ACC. *Fourth*, we add the test results of three ACC-tools, of which two were presented at ICSE in recent years [16, 21]. *Fifth*, we present more test results and explain these results more extensively. Fourth, we identify 10 hard-to-detect types of dependency and four challenges in dependency detection. *Sixth*, we present the frequencies of the hard-to-detect dependency types in five open source systems. *Seventh*, we have improved the testware of the Benchmark test and FreeMind test, and made both sets of testware utilizable for other researchers. Seventh, we extended the Benchmark test at the point of the detection of local variables, and we retested all tools at this point.

In the remainder of this paper, the next section provides an introduction to dependency analysis. Section 3 introduces the tested tools, Section 4 describes the method and results of the Benchmark test, and Section 5 does the same for the FreeMind test. Section 6 describes method and results of a frequency analysis of hard-to-detect dependency types. Section 7 discusses the key findings and discusses the identified challenges in dependency detection. Section 8 discusses the threats to validity, and Section 9 relates our findings to other work. Section 10 concludes this paper; it answers the research questions, summarizes the results of this study, and casts a glance at future work.

2. DEPENDENCY ANALYSIS

Software architecture (SA) compliance checking covers a broad field, because 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' [22]. Static ACC does not cover the full width of SA, but covers the modular architecture. According to Perry and Wolf [22], this architecture should describe the modular elements, their form (properties and relationships) and rationale. In this study, we focus on the relationships between modules. *Relationships* are used to constrain how the different elements may interact or otherwise may be related. In static ACC's center of attention are *uses relations*: 'Module A *uses* module B if A *depends* on the presence of a correctly functioning B to satisfy its own requirements' [23].

Dependency analysis is 'the process of determining a program's dependences' [24]. Various types of dependencies are distinguished in literature. Callo Arias et al. [25] consider that all types fit into three main categories: structural dependencies, behavioral dependencies, and traceability dependencies. The category of structural dependencies, dependencies among parts of a system, is of interest to our study, because static analysis tools focus on dependencies that can be found by inspecting the source code. For instance, Lattix's LDM tool 'uses a standard notion of dependency, in which a module *A* depends on a module *B* if there are explicit references in *A* to syntactic elements of *B*' [12].

2.1. Example of a modular architecture

A small modular architecture in UML notation, which will be used to illustrate the different types of dependency included in our test, is shown in Figure 1. In this diagram, two modules, ModuleA and ModuleB, are shown, each with two submodules. The classes in the submodules are related via associations, showing for instance that an instance of Class1 may know upmost one instance of Class2. The dependency arrows (the dashed arrows) show that ModuleA1 is allowed to use ModuleB1 and that ModuleA2 is allowed to use ModuleB. However, not all rules are visible. The following list shows the full set of relationship rules, of which the first three rules are explicitly visible in the diagram, while the last two are implicit:

- ModuleA1 is allowed to use ModuleB1;
- ModuleA2 is allowed to use ModuleB, so also both sub modules, ModuleB1 and ModuleB2;
- ModuleA1 is not allowed to use ModuleB2;

DOI: 10.1002/spe

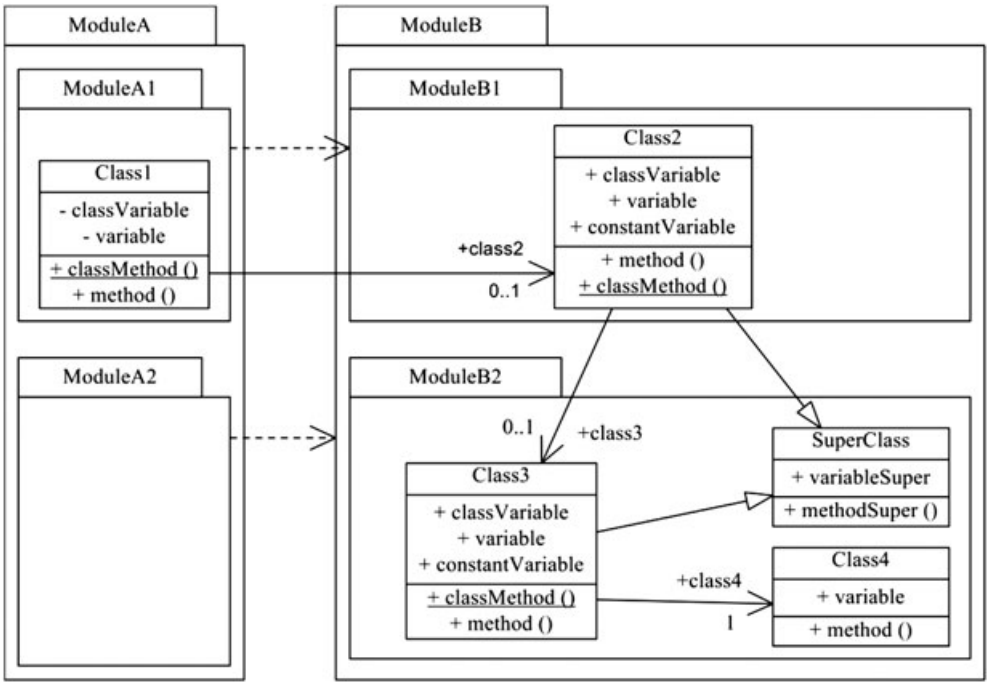

Figure 1. Explanatory model of a modular architecture in UML notation.

- The submodules of ModuleA are allowed to use each other.
- The submodules of ModuleB are allowed to use each other.

2.2. Structural dependency types in object oriented code

Many references of different types can be established in object oriented code. Because not all dependency types in all their variations could be included in our test, we have limited ourselves to quite common types, as described in more detail in the method subsection of the Benchmark test. We identified six main types of dependency: Import, Declaration, Call, Access, Inheritance, and Annotation. For each main type, sub types may be defined. For instance, declarations of instance variables may be distinguished from declarations of class variables.

An example of code per main dependency type is provided below. Each example contains a code construct that causes a rule violating dependency from Class2 to Class3 or to SuperClass, all in Figure 1.

- Import: import ModuleB.ModuleB2.Class3;
- Declaration: variable = class3.method();
- Call: variable = class3.method();
- Access: variable = class3.variable;
- Inheritance: public class Class2 extends SuperClass;
- Annotation: @Class3

2.3. Direct and indirect dependencies

In our study we have included another distinction, namely between direct and indirect dependency. In general, a dependency between two modules is direct, if the dependency relation does not involve an intermediate module. However, we use the term *direct dependency* more specifically, namely for a dependency of which the to-class (the depended-upon class) can be determined, completely based on the knowledge of the from-class (the class that contains the dependency). All six code examples above cause direct dependencies. For example, the dependency caused by the call statement in Class 2, may be traced to Class3, because variable class3 in Class 2 is declared to be of type Class3.

DOI: 10.1002/spe

In general, a dependency relation is indirect, when the dependency exists transitively through an intermediate module. According to this definition, many indirect dependencies are present in program code. For example, if Class1 in Figure 1 contains a method that calls Class2.method(), that somewhere in a scenario calls Class3.method(), then ModuleA1 depends indirectly on ModuleB2 via ModuleB1. In static ACC, this example of an indirect dependency will not be reported as a dependency or violation, because it should result in an overload of dependencies and violations. To prevent an excess of dependencies and violations, we narrow the definition of an indirect dependency in case of static ACC as follows. An *indirect dependency* is a dependency in the from-class of which the to-class cannot be determined without the analysis of the code of another class. Such a dependency should be reported, if a code construct in the from-class has as immediate consequence that the to-class is used; for example in case of access of an inherited attribute, or in case of a call of a method that causes a dependency on the return type of the method.

In these cases, another class needs to be analyzed, or even several other classes, including super classes. The following code examples from Class1 in Figure 1 include a rule violating indirect dependency to Class3 or to SuperClass.

- Call: variable = class2.class3.method();
- Access: variable = class2.variableSuper;
- Inheritance: public class Class1 extends Class2;

3. ACC-TOOLS INCLUDED IN THE TEST

Many tools are available with some of the facilities necessary to support ACC. However, our research focused on tools with explicit support of ACC. We selected publicly available tools, which were mentioned in academic work (e.g., [4, 5, 17, 21]), were able to analyze Java, and provided evaluation or research licenses (two vendors rejected and one did not respond). We excluded tools that focus mainly on architecture visualization, metrics, and/or architecture refactoring.

The 10 tools included in our study are shown in Table I, which also provides an overview of functionalities, code variants, and licensing per tool. The versions of the tools used in our tests together with an URL per tool, are described below Table I, in the footnotes.

The tools provide their support of ACC in various ways:

- Dependometer, Macker, and Sonar Architecture Rule Engine (Sonar ARE) are text-based tools, which support relation conformance rules. These tools provide HTML-based reports as output.
- dTangler and Lattix are based on the Dependency Structure Matrix (DSM) technique, complemented with text-based editors to define rules. The DSM is used to sort and select modules, to define rules, and to show dependencies and violations. Lattix is also able to visualize architectures graphically, and it provides extensive reporting facilities.
- ConQAT Architecture Analysis, JITTAC, and SAVE are strictly based on the Reflexion Model technique [1]. These tools provide a graphical editor to define the intended architecture and to show violations (in terms of divergence and absence) after the evaluation. In addition, ConQAT and SAVE generate textual reports at request, supportive to consistency checks subsequent to software development activities. JITTAC aims at real-time feedback during software development, and for that reason it is tightly integrated in the Eclipse IDE. JITTAC indicates divergences to the architectural model in a diagram and in the source code editor; not only afterwards, but also the moment an inconsistency is programmed.
- Sonargraph Architect and Structure101 are diagram-based too, but these tools are not strictly based on the RM-technique. To define modules and rules, these tools provide diagrams in which the horizontal and vertical position of a module implies rules. Violations are shown in these diagrams, but textual reports are provided in addition.

DOI: 10.1002/spe

Table I. Characteristics of the tools in the test (√ = supported).

Characteristic	ConQAT	Dependometer	dTangler	JITTAC	Lattix	Macker	SAVE	Sonar ARE	Sonargraph architect	Structure101
General functionalities										
Dependency browsing		√	√	√	√				√	√
Dependency visualization				√	√		√		√	√
Architecture compliance checking	√	√	√	√	√	√	√	√	√	√
Architecture refactoring/simulation		√			√				√	√
Team support									√	√
Code variants										
Java	√	√	√	√	√	√	√	√	√	√
Other languages	√	√			√		√			√
Source file analysis		√[a]		√			√	√[a]	√[a]	
Compiled file analysis	√	√	√		√	√		√	√	√
Licensing										
Paid: commercial use					√		√		√	√

[a]In addition to Compiled file analysis.
ConQAT Architecture Analysis—version 2011.9—conqat.org; Dependometer—version 1.2.5—source.valtech.com/display/dpm/Dependometer; dTangler—GUI version 2.0.0—web.sysart.fi/dtangler; JITTAC—version 0.2.0—lero.ie/project/arc; Lattix LDM—version 8.2.7—lattix.com; Macker—version 0.4.2—sourceforge.net/projects/macker; SAVE—version 1.7.1—iese.fraunhofer.de; Sonar ARE—version 3.2—docs.codehaus.org/display/SONAR/Architecture+Rule+Engine; Sonargraph Architect (fusion of Sotograph and SonarJ)—version 7.1.8—hello2morrow.com; Structure101—version 3.5—structure101.com.

DOI: 10.1002/spe

4. BENCHMARK TEST

Two separate tests were performed with the 10 tools: the Benchmark test, and the FreeMind test. This section describes the Benchmark test and the next section the FreeMind test. Both tests were developed and improved iteratively. The first iteration of preparing, testing, and reporting was conducted with 25 students Computer Science in the course of a specialization semester 'Advanced Software Engineering'. Afterwards, the authors have improved the tests and tested the tools completely again in several iterations. The final test results are presented in this paper.

4.1. Method

To prepare our test, we inventoried references in Java code and classified common structural dependencies. We use the term common in this context to indicate pretty basic code constructs supported by Java 1.5; no specialties or constructs only recently supported by Java. The inventory was partly based on research papers that distinguish different architectural dependency types, like [26–30] and partly based on professional literature on Java such as 31 and the official Java Documentation [32]. Furthermore, it was based on the knowledge of 25 students and two Java lecturers. We started with a literature study, which resulted in an initial set of dependency types. Next, six teams of students were tasked with the design of the test, and with the specification of variations of code constructs per dependency type. The results were compared and a best design was selected. The winning team improved its design and worked out the initial version of the test. Thereafter, each team used this initial version to test a tool. The results were interesting, but not reliable enough for publication. Therefore, the test was enhanced and executed iteratively by the authors in two iterations: one for the study published as a conference paper [20] and one for this extended study. Improvements were made in the inventory of dependency types, the test code, the score forms, and the instruction manual. We finished these studies with six main types of structural dependencies: Import, Declaration, Call, Access, Inheritance, and Annotation. In addition, sub types were defined for the main types Declaration, Call, Access, and Inheritance. For instance, we defined seven subtypes of type Call. In combination with the distinction between direct and indirect dependencies, we have distinguished 25 direct dependency types and 9 indirect dependency types. For each dependency type, at least one test case was created. In total, 64 test cases were implemented in Java with Eclipse Indigo SR2.

To measure the sensitivity (also called the true positive rate, or the recall) of the ACC tools, 64 test cases in the test set were aimed at the detection of true positives and false negatives regarding dependency detection and violation reporting. With respect to dependency detection, a true positive indicates that a tool reported a dependency existing in the code, while a false negative indicates that a tool failed to report a dependency existing in the code. With respect to violation reporting, a true positive indicates that a tool reports a violation of a defined rule, while a dependency in the code exists that justifies the violation message. A false negative indicates that a tool failed to report a dependency existing in the code that violates a defined rule. In this paper, we compute s*ensitivity* in percent, as: (number-of-true-positives / (number-of-true-positives + number-of-false-negatives)) × 100.

To measure the false positive rate of the ACC tools, 64 cases were aimed at the detection of false positives. The test code of these test cases was identical to the first 64 test cases, so dependencies to the same to-classes were contained. However, the from-classes, containing the code, were located in another package; as sibling at the same hierarchical level. This way, violations of classes in the second package, based on architectural constraints defined at the first package, should be qualified as false positives. With respect to dependency detection, a false positive indicates that a dependency is reported that is non-existing in the code, while a true negative indicates that a tool correctly did not report a dependency, because it is non-existing in the code. With respect to violation reporting, a false positive indicates that a tool reports a violation of a defined rule, while no dependency in the code exists that justifies the violation message. A true negative indicates that correctly no violation message was reported, because no dependency in the code exists that justifies the violation message. In this paper, we compute the false positive rate in percent, as: (number-of-false-positives / (number-of-false-positives + number-of-true-negatives)) × 100.

DOI: 10.1002/spe

Several tools report violations and dependencies only at the level of from-class, to-class, without further detail. To be able to obtain reliable test results, but also to facilitate and simplify the test process, we implemented a separate from-class per test case. Furthermore, we limited the number of the dependencies to the minimum and where possible to only one dependency on the target class.

After the test preparation, the 10 ACC-tools were tested. All the tools were subjected to the same test, described in the test script. During the first step of the test, the planned modular architecture was entered into the tool, including the mapping of modules to source code units, and the tool's output of the dependency analysis (if provided) was assessed. During the second step, the rules restricting the dependencies between modules were defined, and the output of the tool's conformance check was studied and compared with the expected result and with the output of the tool's dependency analysis. During the third step, the test results of the tools were aggregated and compared.

4.2. Included dependency types

Twenty-five direct dependency types were included in the test and nine indirect dependency types. For each direct and each indirect dependency type at least one separate test case was incorporated. For a part of the dependency types, additional test cases were created with variations of the type. This approach resulted in 34 direct and 30 indirect test cases.

4.2.1. Direct dependency types in the test. The 25 direct structural dependency types in the test are shown in Table II, together with a code example. Each code example shows a code construct that, if

Table II. Direct dependency types in the Benchmark test.

Dependency type	Example code
Import	
Class import	import ModuleB.ModuleB2.Class3;
Declaration	
Instance variable	private Class3 class3;
Class variable	private static Class3 class3;
Local variable	public void method() {Class3 class3; }
Parameter	public void method(Class3 class3) {}
Return type	public Class3 method() {}
Exception	public void method() throws Class4{throw new Class4 ("…"); }
Type cast	Object o = (Class3) new Object();
Call	
Instance method	variable = class3.method();
Instance method-inherited	variable = class3.methodSuper();
Class method	variable = class3.classMethod();
Constructor	new Class3();
Inner class method	variable = class3.InnerClass.method();
Interface method	interface1.interfaceMethod();
Library class method	libraryClass1.libraryMethod();
Access	
Instance variable	variable = class3.variable;
Instance variable-inherited	variable = class3.variableSuper;
Class variable	variable = Class3.classVariable;
Constant variable	variable = class3.constantVariable;
Enumeration	System.out.println(Enumeration.VAL1);
Object reference	method(class3);
Inheritance	
Extends class	public class Class1 extends Class3 { }
Extends abstract class	Idem, but in this case Class3 should be abstract.
Implements interface	public class Class1 implements Interface1 { }
Annotation	
Class annotation	@Class3

DOI: 10.1002/spe

programmed within Class1 in Figure 1, would violate the intended architecture in Figure 1. This is because the code construct includes a dependency to an element of ModuleB2, while the intended architecture does not allow ModuleA1to use ModuleB2. In these cases, we expect ACC-tools to report a violation with at least a specification of the from-class and the to-class. Most of the examples refer to elements in Figure 1, but to keep the figure clear, some specific elements are not included, like an enumeration, exception, and interface.

4.2.2. Indirect dependency types in the test. We included nine indirect structural dependency types in our test, which are shown in Table III, together with a code example per type. Each code example shows a code construct that, if programmed within Class1 in Figure 1, would violate the intended architecture in Figure 1. Because the code construct includes a dependency to an element of ModuleB2, while the intended architecture does not allow ModuleA1 to use ModuleB2. In these cases, we expect ACC-tools to report a violation with a specification of the from-class and the final to-class.

4.3. Findings: accuracy of dependency detection

The test results of our Benchmark tests are shown in detail in Tables IV and V, while the most interesting findings are described below. Table IV shows the results with regard to direct dependencies, and Table V shows the results with regard to indirect dependencies.

As a first observation, we noted that the false positive rate is null for all 10 tested tools; thus, no false positive dependencies were reported. For the observations regarding the sensitivity of the tools, more text is needed. These results are described in detail in the following sub sections.

4.3.1. Direct dependencies. Direct dependencies, caused by type declaration (except local variables), method call, variable access (except constants and object references), and inheritance, were detected by all tested tools, except ConQAT (which missed five type declaration dependency types) and SAVE (which missed two type declaration, one method call, and all six variable access dependency types). The following direct dependency types were often missing or were not reported accurately:

- Import dependencies were detected only by two tools: JITTAC and SAVE; the two tools that analyze source files only. Import statements are not included in compiled files.
- A type declaration of an initialized local variable was detected only by the following six tools: Dependometer, JITTAC, Lattix, SAVE, Sonargraph, and Structure101. However, a type declaration of a not-initialized local variable was detected only by JITTAC and SAVE; the two tools that analyze the source files only. Not-initialized local variables are removed in compiled files. This is interesting, because the tools that analyze compiled files were able to detect other declaration cases without initialization.
- A call of an instance method of an inner class was reported by all tools, except SAVE. However, the tools differ considerably in the accuracy of the reported to-class. JITTAC, Macker, Sonargraph,

Table III. Indirect dependency types in the Benchmark test.

Dependency type	Example code
Call	
Instance method	variable = class2.class3.method();
Instance method-inherited	variable = class2.methodSuper();
Class method	variable = class2.class3.classMethod();
Access	
Instance variable	variable = class2.class3.variable;
Instance variable-inherited	variable = class2.variableSuper();
Class variable	variable = class2.class3.classVariable;
Object reference-Reference var.	variable = class2.method(class2.class3.class4);
Object reference-Return value	Object o = (Object) class2.getClass4();
Inheritance	
Extends-implements variations	public class Class1 extends Class2 { } public class Class2 extends SuperClass { }

DOI: 10.1002/spe

Table IV. Benchmark test—detection of direct dependencies (0 = not detected; 1 = detected).

Dependency type	ConQAT	Dependometer	dTangler	JITTAC	Lattix	Macker	SAVE	Sonar ARE	Sonargraph	Structure101	
Import											
Class import	0	0	0	1	0	0	1	0	0	0	2
Declaration											
Instance variable	0	1	1	1	1	1	1	1	1	1	9
Class variable	0	1	1	1	1	1	1	1	1	1	9
Local variable, initialized	0	1	0	1	1	0	1	0	1	1	6
Parameter	0	1	1	1	1	1	1	1	1	1	9
Return type	0	1	1	1	1	1	0	1	1	1	8
Exception	1	1	1	1	1	1	1	1	1	1	10
Type cast	1	1	1	1	1	1	0	1	1	1	9
Call											
Instance method	1	1	1	1	1	1	1	1	1	1	10
Instance method, inherited	1	1	1	1	1	1	1	1	1	1	10
Class method	1	1	1	1	1	1	1	1	1	1	10
Constructor	1	1	1	1	1	1	1	1	1	1	10
Inner class method (instance)	1	1	1	1	1	1	0	1	1	1	9
Interface method	1	1	1	1	1	1	1	1	1	1	10
Library class method	1	1	1	1	1	1	1	1	1	1	10
Access											
Instance variable (read, write)	1	1	1	1	1	1	0	1	1	1	9
Instance variable, inherited	1	1	1	1	1	1	0	1	1	1	9
Class variable	1	1	1	1	1	1	0	1	1	1	9
Constant variable	0	1	0	1	0	0	0	0	1	0	3
Enumeration	1	1	1	1	1	1	0	1	1	1	9
Object reference, param. value	0	1	0	0	0	0	0	0	1	1	3
Inheritance											
Extends class	1	1	1	1	1	1	1	1	1	1	10
Extends abstract class	1	1	1	1	1	1	1	1	1	1	10
Implements interface	1	1	1	1	1	1	1	1	1	1	10
Annotation											
Class annotation	0	1	0	1	1	0	0	0	1	1	5
Detected (out of 25)	16	24	20	24	22	20	15	20	24	23	
Sensitivity (in %) (average = 83)	64	96	80	96	88	80	60	80	96	92	

DOI: 10.1002/spe

Table V. Benchmark test—detection of indirect dependencies (0 = not detected; 1 = detected).

Dependency type	ConQAT	Dependometer	dTangler	JITTAC	Lattix	Macker	SAVE	Sonar ARE	Sonargraph	Structure101	
Call											
Instance method	1	1	1	1	1	1	1	1	1	1	10
Instance method, inherited	0	0	0	1	0	0	1	0	0	1	3
Class method	1	1	1	1	1	1	1	1	1	1	10
Access											
Instance variable	1	1	1	1	1	1	0	1	1	1	9
Instance variable, inherited	0	0	0	1	0	0	0	0	0	1	2
Class variable	1	1	1	1	1	1	0	1	1	1	9
Object reference—reference var.	1	1	1	1	1	1	0	1	1	1	9
Object reference—return value	0	1	0	0	0	0	0	0	0	1	2
Inheritance											
Extends—implements variations	0	0	0	0	0	0	0	0	0	0	0
Detected (out of 9)	5	6	5	7	5	5	3	5	5	8	
Sensitivity (in %) (average = 60)	56	67	56	78	56	56	33	56	56	89	

DOI: 10.1002/spe

and Sonar ARE were specific and reported the outer and inner class. ConQAT, Dependometer, dTangler, Lattix, and Structure101 were less accurate and reported only the outer class.
- Access of a constant variable was detected only by three tools: Dependometer, JITTAC, and Sonargraph Architect. We included three test cases: one with a constant instance variable, one with a constant class variable, and one with an interface class variable. However, the results per tool were the same over these three test cases. Tools that analyze compiled code only, have problems with the recognition of constants, because their values are in-lined by the Java compiler. Dependometer and Sonargraph were detecting an access of a constant variable only with the option marked to include the source code in the analysis. Although SAVE analyzes source code, it did not report a dependency in one of the three test cases.
- Access of an object reference in the form of a parameter value (or argument), was detected only by three tools: Dependometer, Sonargraph, and Structure101. Another test case of this dependency type, with an object reference included in an if-clause, was detected by two tools only: Dependometer and Sonargraph.
- Dependencies of type annotation were detected only by five tools: Dependometer, JITTAC, Lattix, Sonargraph, and Structure101.

4.3.2. Indirect dependencies. Indirect dependencies caused by method call and variable access (except an object reference as return value), were detected by all tested tools, except SAVE, which did not report access dependencies. Even double indirect dependencies were detected (for instance, from Class 1 in Figure 1, via Class 2 and Class 3 to Class 4). However, the following indirect dependency types were often missing or were not reported accurately:

- A call of an inherited instance method was reported accurately only by three tools: JITTAC, SAVE, and Structure101. These tools reported an indirect dependency to the super class where the method was actually implemented, although this method was called via a subclass. The other tools reported a dependency to the intermediate subclass, but not to the super class where the method was implemented. Consequently, these tools did not report a violation in the test cases where the subclass is part of an allowed-to-use module, while its super class is part of a not-allowed-to-use module.
- Access of an inherited instance variable was reported accurately only by two tools: JITTAC, and Structure101. These tools reported an indirect dependency to the super class where the variable was actually implemented, although this variable was accessed via a subclass. The other tools reported a dependency to the subclass, but not to the super class where the variable was implemented (except SAVE, which did report no dependency at all). Consequently, these tools did not report a violation in the test cases where the subclass was part of an allowed-to-use module, while its super class was part of a not-allowed-to-use module.
- Access of an object reference, received as return value of a method call, was reported by only two tools: Dependometer and Structure101.
- An inherited dependency on a super-super class or interface of the from-class, solely based on extends/implements constructs, was not reported by any tool. We included three variations in our test (extends-extends, extends-implements, implements-extends), but none was reported.

5. FREEMIND TEST

In addition to the Benchmark test we have conducted tests with an open source system. Two different test were conducted, one aimed at the accuracy of dependency detection, and the other on the accuracy of violation and dependency reporting. The rules in these tests are defined by ourselves, in line with the objectives of the tests, and do not represent rules defined by the developers of Freemind.

5.1. Method

5.1.1. FreeMind. We used the mind-mapping tool FreeMind. Three main packages in FreeMind, as shown in Figure 2, were included in our tests: accessories, plugins, and freemind. The following packages were excluded from the test, because these packages were available in the source code, but not in

DOI: 10.1002/spe

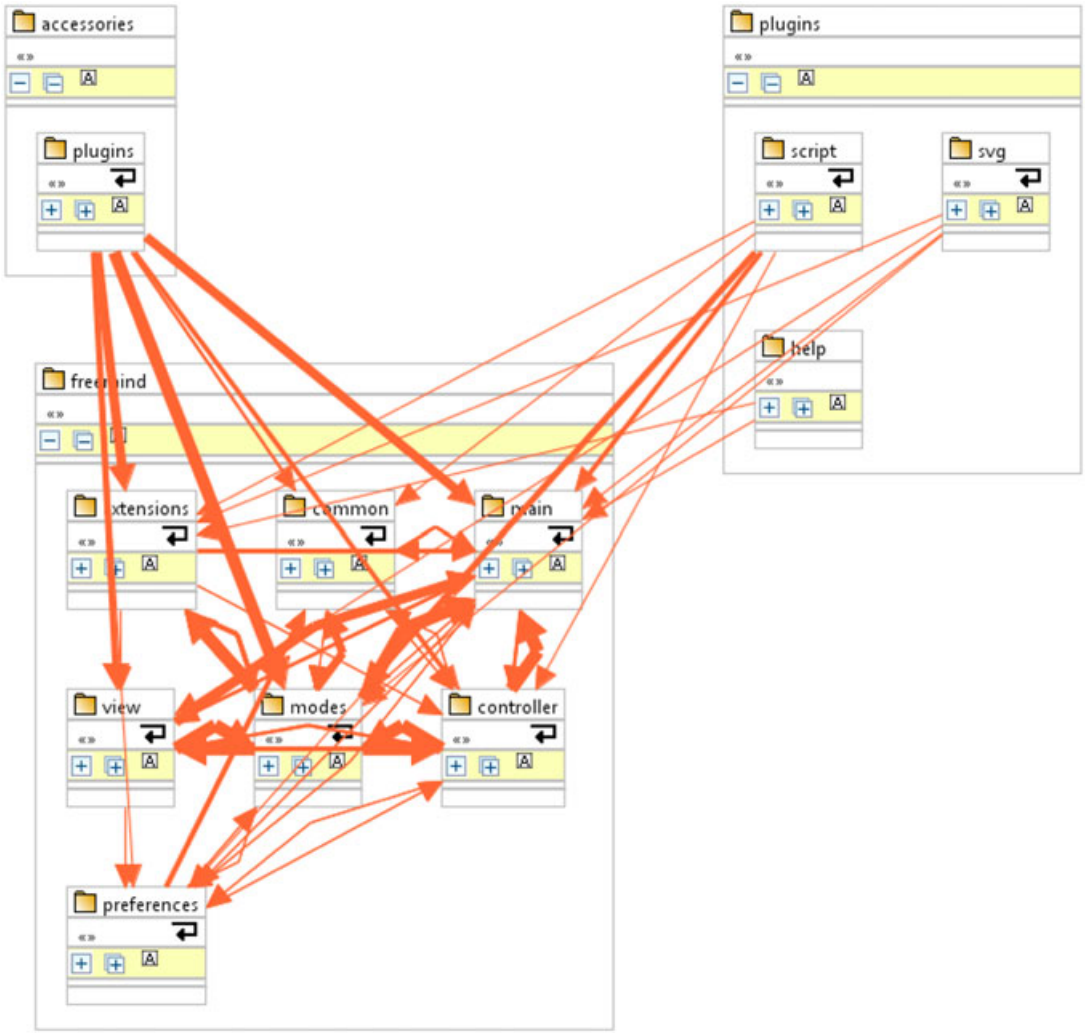

Figure 2. The package structure of Freemind, with dependency relations, as depicted by SAVE. Thick lines represent more dependency relations than thin lines.

the compiled code: plugins.latex.*, plugins.collaboration.*, and tests.*. We used version 0.9.0 for our tests; retrieved on 23-08-2012 from http://freemind.sourceforge.net/wiki/index.php/Download.

We selected FreeMind, because it suited to the following criteria. First, the system needed to be written in Java, just as the Benchmark test. Furthermore, source code files and compiled code files needed to be present, because some tools use source code, others compiled code, while some use both. Second, the system needed to have an uncomplicated implemented architecture, to enable a straightforward registration in the tested tools of the modules, their assigned code files, and the rules. Because the tools vary considerably in how modules and rules are defined, a simple one-to-one mapping between modules and code units is needed to prevent mapping errors and subsequent bias in the test. Third, the system needed to contain a lot of dependencies between its modules, and these dependencies should cover a wide range of possible dependency types. Fourth, the number of classes had to be lower than 1000, because of size constraints of some ACC-tool licenses.

5.1.2. Method: accuracy of dependency detection test. The objective of this test was to determine how well the ACC tools were able to report dependencies of different types, just as in the Benchmark test, but now in a real system. For this purpose, we selected the large class ScriptingEngine within sub package plugins.script, because it contained a considerable number of dependencies of

DOI: 10.1002/spe

different types. Furthermore, the class touched a diversity of object oriented specialties, including super class, inner class, and anonymous class.

5.1.2.1. Identification of dependency causing constructs. We identified all code constructs within class ScriptingEngine, which caused dependencies to package 'freemind', by manual inspection of the code, aided by the supporting facilities of the Java editor in the Eclipse IDE. To ensure the accuracy of our work during this step, one author made an inventory of the dependency-causing constructs, the depended-upon classes, and the related dependency types, while another author checked the inventory afterwards. Based on the inventory, 109 dependency-causing code constructs were included in a score form.

5.1.2.2. Tool selection and testing. We tested all 10 tools to determine which depended-upon classes were reported for class ScriptingEngine, by following the steps below.

1. Registration of rule: plugins.script.ScriptingEngine is not allowed to use package freemind.
2. Activation of the conformance check.
3. Study of the reported violations and dependencies.

Next, we selected the tools that were providing sufficient information to be able to trace reported dependencies to code constructs. The tools differ considerably in the exactness of dependency messages, as will be discussed in the result sub section. Only the following five tools provided detailed enough information to be included in this test: JITTAC, Lattix, SAVE, Sonargraph Architect, and Structure101. With these five tools we also went through the following steps:

4. Tracing of the reported dependencies to the manually identified dependency constructs.
5. Scoring of the detected dependency constructs in a scoring form per tool.

5.1.2.3. Scoring. We scored mildly, meaning that we marked a dependency as detected, if one of the reported dependency messages could be related to the dependency-causing code construct. With a strict accuracy level in mind, the number of missed dependencies would have been much higher.

- In case of inner class related dependencies, we scored a dependency as detected if it was reported as a dependency to the outer class instead of to the inner class.
- In case of inheritance related dependencies, we scored a dependency as detected if it was reported as a dependency to a sub class instead of the super class that actually implemented a depended-upon variable or method.
- In case of dependency messages with a non-optimal accuracy, we scored all dependencies as detected that could be related to the dependency message. For instance, if a tool reported one dependency to class X of type declaration or access at line Y, while in the source code a declaration construct and a type cast construct were present, both were scored to be detected. Similarly, if a tool reported one dependency to class X of type access in method Z, while in the source code of the method five of these access constructs were present, all five were scored as detected.

5.1.3. Method: accuracy of reported violation and dependency messages test. The objective of this test was to identify differences in quantity and exactness of the reported violation and dependency messages. This test encompassed all dependencies between the three packages at top-level in the code: freemind, plugins, and accessories. Per tool, the test was executed as follows:

1. Registration of two rules: (i) package accessories is not allowed to use package freemind; and (ii) package plugins is not allowed to use package freemind.
2. Activation of the conformance check.
3. Study of the reported violation and dependency messages.
4. Scoring of the number and exactness of the messages.

DOI: 10.1002/spe

5.2. Findings: accuracy of dependency detection test

The first test with FreeMind concerns the accuracy of dependency detection. The test results are presented below. All 10 tools provided at least information in their violation messages on the depended-upon to-classes per from class. Therefore, the results of the reported depended-upon classes per tool are presented at first. Next, the results of the test at the level of the 109 identified dependency constructs are presented. Five tools are included in these results, because only these tools provided detailed enough information in their violation messages or dependency messages. Finally, examples are provided of code constructs that caused hard to detect dependencies.

5.2.1. Detected depended-upon classes. Class ScriptingEngine depends-upon 17 classes, of which most are shown in the freemind package in Figure 3. Two of these classes contain inner classes, which are also used by ScriptingEngine, namely OptionalDontShowMeAgainDialogue and Tools. In our test, we expected that usage of the seventeen depended-upon classes would be reported as violations. Please note that Figure 3 provides a simplified view. There are many more classes in package freemind, and the shown classes are in reality included in different subpackages of freemind. Furthermore, for reasons of readability, we have drawn no dependency arrows in the diagram, only UML inheritance relations.

Several inheritance structures are shown in the figure. For example, ScriptingEngine inherits from three superclasses in package freemind. In our test, we expected that usage of these classes would be reported as violations; especially in case of a call of method or in case of access of an attribute inherited from one of these classes. In these cases, actual usage takes place of the super class that implements the method or variable.

Table VI shows for each of the tested tool, which depended-upon classes in package freemind were reported in violation reports or other views. Furthermore, it shows per class the number of related dependency-causing constructs. The bottom row in the table shows that JITTAC was the only tool that reported usage of all seventeen classes. Dependometer, SAVE, and Sonargraph reported usage of fifteen classes, a sensitivity of 88%. Macker, Sonar ARE, and Structure101 reported usage of fourteen classes, a sensitivity of 82%. Finally, Conquat, dTangler, and Lattix reported usage of twelve classes, a sensitivity of 71%. On average, 82% of the classes was detected.

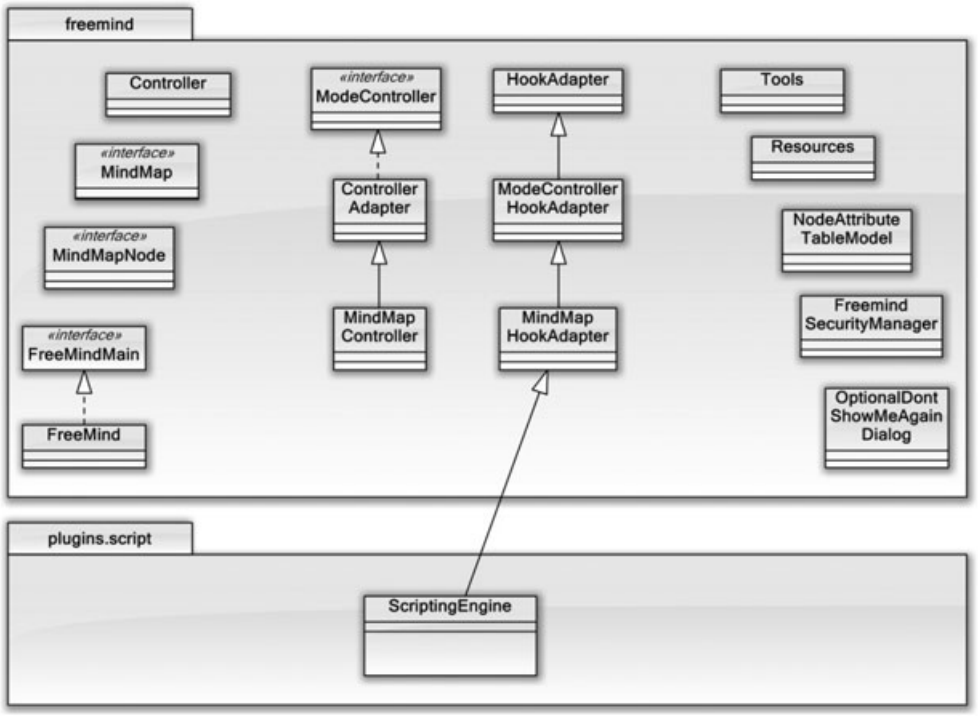

Figure 3. Class ScriptingEngine and its depended-upon classes in package freemind.

DOI: 10.1002/spe

Table VI. Freemind test—detected (1) and not detected (0) depended-upon classes.

Reported classes	Nr of dep. constructs	ConQAT	Dependometer	dTangler	JITTAC	Lattix	Macker	SAVE	Sonar ARE	Sonargraph	Structure 101	
Controller	1	1	1	1	1	1	1	1	1	1	1	10
ControllerAdapter	5	0	0	0	1	0	0	1	0	0	0	2
FreeMind	12	0	1	0	1	0	0	1	0	1	0	4
FreeMindMain	16	1	1	1	1	1	1	1	1	1	1	10
FreeMindSecurityManager	5	1	1	1	1	1	1	1	1	1	1	10
HookAdapter	6	0	0	0	1	0	0	1	0	0	0	2
MindMap	1	1	1	1	1	1	1	1	1	1	1	10
MindMapController	6	1	1	1	1	1	1	1	1	1	1	10
MindMapHookAdapter	5	1	1	1	1	1	1	1	1	1	1	10
MindMapNode	17	1	1	1	1	1	1	1	1	1	1	10
ModeController	2	1	1	1	1	1	1	1	1	1	1	10
NodeAttributeTableModel	6	1	1	1	1	1	1	1	1	1	1	10
OptionalDontShowMeAgainDialog	5	1	1	1	1	1	1	1	1	1	1	10
- StandardPropertyHandler (inner)	1	0	1	0	1	0	1	0	1	1	1	6
Resources	2	1	1	1	1	1	1	1	1	1	1	10
Tools	6	1	1	1	1	1	1	1	1	1	1	10
- BooleanHolder (inner)	13	0	1	0	1	0	1	0	1	1	1	6
Number of reported classes		12	15	12	17	12	14	15	14	15	14	
Sensitivity (in%) (average = 82)		71	88	71	100	71	82	88	82	88	82	

DOI: 10.1002/spe

All not reported classes (by all tools) were of one of the following types:

- Super class (ControllerAdapter, Hookadapter), of which methods are used via inheritance;
- Inner class (OptionalDontShowMeAgainDialogue.StandardPropertyHandler, Tools.BooleanHolder), which may be used in various ways: Import, Declaration, Access, Call;
- Normal class (FreeMind), of which only static constant variables are accessed.

5.2.2. Detected dependencies. Table VII shows for each of the five tools included in the detailed test, how many dependencies per dependency type were reported to classes in package freemind. All five tools detected all the dependencies of the following dependency types: (i) method call, class method; (ii) method call, interface method; and (iii) inheritance, extends class.

Dependencies of the other dependency types, which were not reported by one or more tools, are per type discussed below.

- Import, class import: Lattix, Structure101, and Sonargraph missed all 10 dependencies. SAVE missed one, because of a not-recognized inner class.
- Declaration, local variable: SAVE missed all six dependencies (in contrast to the Benchmark test), probably because off complex initialization statements at the same line.
- Declaration, parameter: SAVE missed three out of seven dependencies (because of a not detected inner class), while Sonargraph missed one.
- Declaration, type cast: SAVE missed all two dependencies (as in the Benchmark test).
- Call, instance method: JITTAC missed two dependencies, probably because these were located within an anonymous class.
- Call, instance, inherited: Lattix missed eight out of fourteen dependencies, Sonargraph also missed eight, and Structure101 missed all fourteen (in contrast to the Benchmark test), all in inheritance trees up to four levels.
- Call, constructor: SAVE missed two dependencies out of three: two constructor invocations of inner classes. It detected an invocation of the constructor of a normal class (as in the Benchmark test).

Table VII. Freemind test—reported dependencies per dependency type.

Dependency type (number of constructs)	JITTAC	Lattix	SAVE	Sonargraph	Structure 101
Import					
Class import (10)	10	0	9	0	0
Declaration					
Local variable (6)	6	6	0	6	6
Parameter (7)	7	7	4	6	7
Type cast (2)	2	2	0	2	2
Call					
Instance method (11)	9	11	11	11	11
Instance method-inherited (14)	14	6	14	6	0
Class method (6)	6	6	6	6	6
Constructor (3)	3	3	1	3	3
Inner class method (instance) (2)	2	2	0	2	2
Interface method (19)	19	19	19	19	19
Access					
Constant variable (12)	12	0	0	12	0
Object reference (16)	0	0	10	16	16
Inheritance					
Extends class (1)	1	1	1	1	1
Detected (109)	91	63	75	90	73
Sensitivity (in %) (average = 72)	83	58	69	83	67

DOI: 10.1002/spe

- Call, inner class method: SAVE missed all two instance method invocations (as in the Benchmark test).
- Access, constant: Lattix, Structure101 and SAVE missed all twelve dependencies (as in the Benchmark test).
- Access, object reference: JITTAC and Lattix missed all sixteen dependencies (as in the Benchmark test); 15 caused by variables passed as parameter value (or argument) and one caused by a variable used within an if statement. SAVE missed six, because of not detected inner classes.

Remind, we scored mildly, as explained in the method sub section of this test.

5.2.3. Examples of code constructs. Several examples of code constructs that caused dependencies that were hard-to-detect in the FreeMind test, are provided in Table VIII. The first column shows the type of the dependency with, if needed, some added details. The second column shows the example code. The text that causes a dependency is shown in italic.

5.3. Results: accuracy of reported violation and dependency messages

The second test with FreeMind concerns the accuracy of Accuracy of reported violation and dependency messages. This test encompassed all dependencies between the three packages at top-level in the code: freemind, plugins, and accessories.

The test focuses on two functional types of messages: violation messages and dependency messages. *Violation messages* report on inconsistencies between the implemented architecture and the defined architecture, with a class at the lowest level of granularity. The second type, *dependency messages* provide information about the dependencies, like the dependency type and the location of the dependency-causing code constructs in the program code. Findings regarding these two types are presented in the following subsections.

To illustrate the difference between the two types of messages, we have included practical examples from the FreeMind test with Structure101. Three examples show messages at three different levels of abstraction. Figure 4a shows a graphic with a violation message: a violating relation

Table VIII. Examples of code constructs within ScriptingEngine.

Dependency type	Example code
Import	
Class import (inner class)	import *freemind.main.Tools.BooleanHolder*;
Declaration	
Local variable	MindMapNode *node* = getMindMapController(). getMap().getRootNode();
Call	

Instance method
(within an anonymous class)
	final GroovyShell shell = new GroovyShell(binding) {
	public Object evaluate(…, …) throws … {
	try {
	securityManager.setFinalSecurityManager(…);
	}
	}
	};

Inherited method (of MindMapHookAdapter)	MindMapNode node = *getMindMapController()*. getMap().getRootNode();
Inherited method (of ControllerAdapter)	MindMapNode node = getMindMapController(). *getMap()*.getRootNode();
Constructor(of inner class)	BooleanHolder bh = *new BooleanHolder*(false);
Inner class method	*bh*.setValue(true);
Access	
Constant class variable (passed as parameter value)	String executeWithoutAsking = frame.getProperty(FreeMind. *RESOURCES_SIGNED_SCRIPT_ARE_TRUSTED)*
Object reference (passed as parameter value)	performScriptOperation(element, *bh*);

DOI: 10.1002/spe

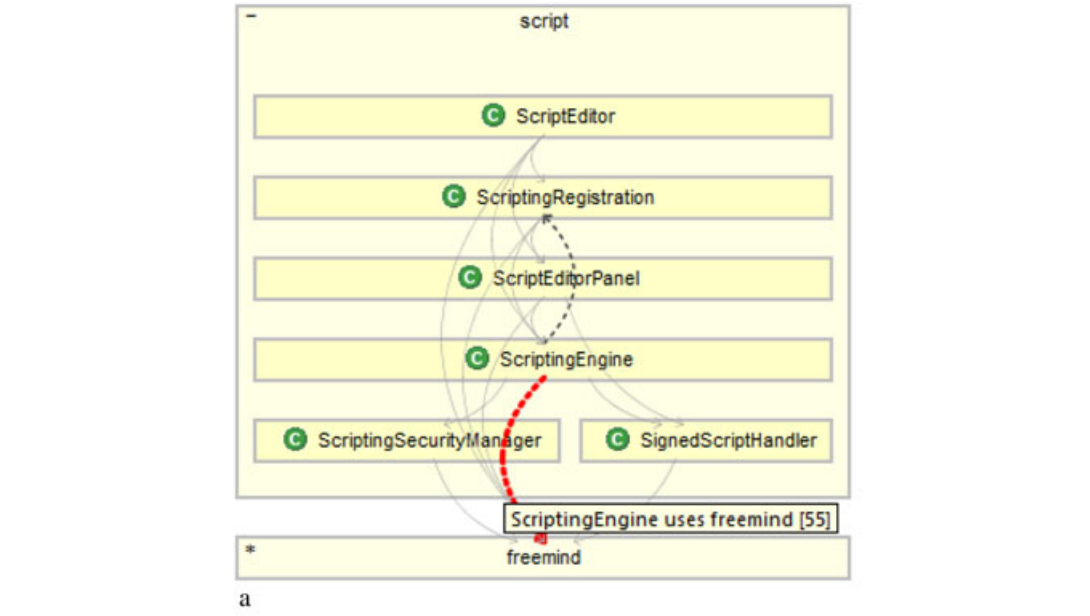

Figure 4. a) The package structure of Freemind plugins.script, as depicted by Structure101, with one violating dependency relation (red, bolt) from class ScriptingEngine to package freemind. b) Specification of the violating relation; 14 classes and interfaces are used. c) Detailed specification of dependencies from ScriptingEngine to freemind.

(dotted) from class ScriptingEngine to package freemind. After selection of the relation, the tool shows the message 'ScriptingEngine uses freemind [55]'. Figure 4b shows violation messages within the view that specifies the violating relation; it lists fourteen depended-upon classes and interfaces within package freemind. Figure 4c shows examples of dependency messages. Structure101 reported 55 instances of dependencies from ScriptingEngine to freemind. These dependencies were specified in a separate view, which listed for each dependency: the from-class

DOI: 10.1002/spe

(ScriptingEngine), from-method (e.g., evaluate), dependency type (e.g., extends), to-class (e.g., freemind.main.FreeMindSecurityManager), and to-method (e.g., setFinalSecurityManager).

5.3.1. Violation messages. Violation messages indicate where the implemented architecture deviates from the intended architecture. The tested tools differ considerably in the way violations are reported, for instance by means of colors in a Dependency Structure Matrix (DSM), additional symbols or line styles in diagrams, textual reports, or indicated code lines in a code viewer. Most tools offer more than one way to report violations, especially the commercial tools.

Observations regarding the accuracy of violation messages are described below.

- Reported violations versus reported dependencies

  No cases were noticed, in which a tool reported a dependency to a class, but failed to report a violation for this dependency. Because this is also true for the Benchmark test, Table IV and V do not only show the true positive violations, but also the false negative violations per tool and per dependency type. However, one exception applies: SAVE reported correct violations for classes containing violating *direct* dependencies, even when the specific dependency of the test case was not detected. The tool was able to do this, based on detected import statements. SAVE did not have this advantage in case of *indirect* dependencies, because no import statement was included in these cases, and in case of an inner class.

- Exactness of the violation messages

  To show where a violation is present in the modular architecture, seven of the ten tools (see Table IX) include violation messages in graphical overviews. Table IX shows also that all tools were able to report the from-class and to-class, generally in text-based violation messages. However, management information to indicate the severity of the violation of a rule, like the actual number and/or strength of the underlying dependencies, is less frequently included in violation reports, while this is meaningful information. As a positive example, Structure101 shows, in Figure 4a, the number of corresponding dependency messages when a relation is selected. Furthermore, it is available in JITTAC diagrams, where the number of dependencies is shown per dependency arrow, and in SAVE diagrams, though less accurate, where the thickness of a line indicates the number of dependencies.

- Number of reported messages

  Because of different capabilities of the tools and different choices made by the developers, the tools report varying numbers of violation messages at the level of from-class, to-class. This is illustrated in Table X, which holds the numbers of violation messages per tool during the FreeMind test (except for JITTAC and SAVE; see table footnote). The reported violations against the rule that the freemind package should not be used, are shown for package accessories, for package plugins, and for class plugins.script.ScriptingEngine. Several tools report more violations than depended-upon classes in their violation report. In these cases, a separate message is created for each combination of from-class, to-class, and dependency type.

5.3.2. Dependency messages. A dependency message enables developers to resolve a violation efficiently. To do so, detailed information is needed to trace the dependencies in the code. Six tools provide this information in separate reports or views: JITTAC, Lattix, SAVE, Sonar ARE, Sonargraph Architect, and Structure101. Our observations regarding the exactness of the dependency messages are described below.

5.3.3. Exactness of the reported location of a dependency. The tools differ in exactness of the reported location of a dependency, as shown in Table IX. At the highest level of accuracy, a tool

DOI: 10.1002/spe

Table IX. Exactness of violation and dependency messages (√ = included in message).

	ConQAT	Dependometer	dTangler	JITTAC	Lattix	Macker	SAVE	Sonar ARE	Sonargraph	Structure101
Violation message										
Graphical overview	√		√	√	√		√		√	√
Class from – Class to	√	√	√	√	√	√	√	√	√	√
Dependency message										
Class from – Class to	√	√	√	√	√	√	√	√	√	√
Dependency type					√		√		√	√
Method from – Method to							√			√
Line				√[a]	√[a]			√[a]	√	
Position within the line				√[a]	√[a]					

[a]An indication of the line (and position of the dependency construct within the line) is not provided in a report, but in a code viewer or IDE plug-in.

DOI: 10.1002/spe

Table X. Reported numbers of violation and dependency messages within the Freemind test.

	ConQAT	Dependometer	dTangler	JITTAC[a]	Lattix	Macker	SAVE[a]	Sonar ARE	Sonargraph	Structure101
accessories → freemind										
Violations	228	435	282	1739	288	386	1332	378	362	308
plugins → freemind										
Violations	54	100	63	348	65	87	229	79	75	71
ScriptingEngine → freemind										
Violations	12	18	12	97	25	16	54	15	15	14
Dependencies				97	25		54		121	55

[a]JITTAC and SAVE provide no aggregated violation report with messages at the level of from-class, to-class. Instead, the table shows the number of reported dependency messages.

indicates a dependency-causing construct within a line of code, even when several dependency-causing constructs are included in the same line. Only two tools were able to do this: JITTAC, and Lattix. Both tools highlight the violating code constructs in the source code within an IDE's code editor. Table IX shows that these two tools provide the following information in dependency messages: class-from, line, and position within line. However, Lattix did not always appoint the line and code construct correctly.

Two other tools with code viewers, Sonar ARE and Sonargraph Architect, indicated the line correctly, but not the position within the line. Sonar ARE's usability was restricted by the fact that per from-class, it indicated only the first instance of a violating usage of a depended-upon class. Following usages of the to-class were not indicated.

Several tools provided reports as well. Sonargraph Architect provided the most detailed report with from-class, to-class, dependency type, and a correct line number. Lattix provided an information view with dependency types and line numbers of the dependencies in the code, but here also, it did not always specify the correct line number in the source code. SAVE and Structure101 provided reports, which indicated the method including the violating dependency in the from-class and, in case of method calls, also the method of the to-class.

The practical implications of the different approaches became clear during the FreeMind test, in which reported violations needed to be traced to 109 constructs in the program code. At first, we tried to do this based on the dependency messages in the reports or in the dependency browsers of the tools. Sonargraph Architect provided a very useful report, which made it easy to trace the dependencies in the code. It contained all the detected dependencies with type and line number. The reports of SAVE and Structure101 required much more analysis and interpretation, with risk of misinterpretation in complex situations. In part, because one dependency message may abstract several dependency-causing code constructs. In concrete terms: SAVE and Structure101 reported respectively 54 and 55 messages, but these covered respectively 75 and 73 dependencies in the code. Because Lattix's reports proved to be too inaccurate for our use, and because JITTAC did not provide a report or dependency browser, we used the messages provided in code viewers of these tools. In the case of Lattix, we combined different reports with the code viewer to circumvent incorrect line numbers and positions. Finally, because Sonar ARE's support for this test was too restrictive, we did not include the tool in this part of the FreeMind test.

5.3.4. Exactness of the reported dependency type. Only four tools provide a dependency type (as shown in Table IX), which differentiates between different types of usage, like declare, access, or call. The tools differ in the exactness of the reported dependency type: the numbers of dependency types vary per tool, and the names of these types vary as well. Consequently, different tools label a dependency type in our classification in several ways. For example, a dependency of type 'Call constructor' in our classification was reported by Lattix as 'Construct with Arguments', by SAVE as 'ACCESS', by Sonargraph as 'Uses new', and by Structure101 as 'calls'. Some types used by the tools are very specific, while others cover many forms of code constructs. Even if two tools use the same type-name, like access, they may cover different dependency types within our classification.

6. FREQUENCY OF HARD-TO-DETECT DEPENDENCY TYPES

The results of the Benchmark test and FreeMind test have shown that certain types of depended-upon classes and 10 types of dependencies are not reported at all by some tools, or are reported inaccurately. To address the relevance of these findings, we have measured the number of dependencies per dependency type in five open source systems. The method and the results of this experiment are described below.

6.1. Method

To measure the numbers of dependencies per dependency type distinguished in the Benchmark test, we needed a tool that was able to detect and report all the types of dependencies in these tests. Because no tested tool was able to detect dependencies of all these types, we improved and extended a

DOI: 10.1002/spe

tool, HUSACCT, which we had developed in a line of research that focused on ACC support for rich sets of module and rule types [33]. We improved the dependency analysis process in HUSACCT version 4.0 to the level that all dependencies in the Benchmark test and FreeMind test were detected and reported, without false positives. Because a considerable part of the not-reported dependencies in the Benchmark test and FreeMind test were related to inheritance and inner class constructs, we extended the analysis process and data model to detect and store these characteristics per dependency. Furthermore, we extended the dependency report with a statistics sheet, which presents the numbers of dependencies in different ways: total, direct, indirect, total per type, total of inheritance related dependencies, total of inner class related dependencies, et cetera. To enable the reproduction of the experiment by other researchers, the improvements and extensions were included in version 4.1, which is downloadable via http://husacct.github.io/HUSACCT/.

Next, we selected five open source subject systems and downloaded their source code. We used the following selection guidelines. First, the systems had to be written in Java, because our Benchmark test and FreeMind test were also Java-based. Second, FreeMind was included, because it is interesting to compare its analysis results in the FreeMind test with those of other subject systems. Third, four other systems were selected, because they were used in published scientific experiments of other authors, but also because of their notoriety.

The five systems, their version, download address and size are shown in Table XI, sorted on size. The source of all the systems was downloaded on February 10, 2015 (except FreeMind, which was downloaded already in 2012; the current location of the source is included in the table). An impression of the size per system is provided in kilo lines of code (KLOC). The given numbers show the lines of code (including comments and blank lines) in all the files with extension '.java', as measured by HUSACCT.

Finally, for each system the source code was analyzed with HUSACCT and a dependency report was generated. The numbers of dependencies per dependency type per system were included in a spreadsheet and averages were calculated, per system, and over the systems. These final results are presented in the next sub section. The reported numbers of dependencies: (i) include internal dependencies and dependencies on external systems (library objects); (ii) exclude dependencies from a class to itself; and (iii) exclude dependencies on primitive types in case of declarations.

6.2. Results

Table XII shows the numbers of dependencies per dependency type and per system. The 10 dependency types that proved *hard-to-detect* in our tests are included in the table; they are shown in italics. The results are presented in three groups, visible in the first column, namely: (i) all dependencies; (ii) inheritance related dependencies; and (iii) INNER class related dependencies. Per group and per dependency type, the numbers of reported dependencies are shown per subject system, which are sorted on size, while the last column shows the average percentage of the dependency type over the four subject systems. The average percentage of a dependency type is calculated as the average for this type of the four subject system specific percentages (not shown in the table).

The first group shows the numbers of all reported dependencies. The first row within this group shows that the total number of dependencies increases with the size of the subject system, as can be expected. Thereafter, two subdivisions are shown; one for direct versus indirect dependencies, and another for the six main types (Import, Declaration, Call, Access, Inheritance, Annotation). The numbers show that on average 84% of the dependencies are direct, while 16% are indirect in these subject

Table XI. Subject systems used in the experiment.

System	Download address	Size (KLOC)
Hibernate 4.2.4	https://github.com/hibernate/hibernate-orm/releases/tag/4.2.4.Final	713
Findbugs 3.0.0	https://code.google.com/p/findbugs/source/browse/?name=3.0.0	327
Struts 2.3.20	http://struts.apache.org/download.cgi#struts2320	277
Ant 1.9.4	http://archive.apache.org/dist/ant/source/	267
Freemind 0.9.0	http://sourceforge.net/projects/freemind/files/freemind/0.9.0/	87

DOI: 10.1002/spe

Table XII. Number of dependencies per dependency type.

Dependencies	Type	Hibern.	Findbugs	Struts	Ant	%
All	All	401,356	128,876	122,877	88,943	100
	- Direct	319,530	115,070	102,332	76,350	84
	- Indirect	81,826	13,806	20,545	12,593	16
	Import	39,670	15,988	12,528	8,422	10
	Declaration	69,372	28,621	22,764	18,282	20
	- Local var.	22,976	10,167	8,316	5,076	7
	Call	155,051	43,271	51,874	37,208	39
	Access	115,224	35,276	29,804	20,388	26
	- Constant variable	4,589	1,695	920	1,923	1
	- Object ref. direct	54,797	21,541	16,223	12,233	14
	- Object ref. indirect	28,218	5,675	5,244	3,047	5
	Inheritance	8,610	3,066	4,665	2,368	3
	Annotation	13,429	2,654	1,242	2,275	2
Inheritance related	All	47,608	8,731	19,219	10,291	12
	Access	19,200	1,956	5,043	1,961	3
	- Inh. var. indirect	10,941	1,252	3,294	1,008	2
	Call	19,798	3,709	9,511	5,962	6
	- Inh. meth. indirect	17,186	2,861	5,431	4,657	4
	Inheritance	8,610	3,066	4,665	2,368	3
	- Indirect	3,839	1,641	2,779	1,205	2
Inner class rel.	*All*	16,650	14,283	12,343	10,739	9

systems. Furthermore, that Import statements cause 10% of the dependencies, while Declaration, Call, Access, Inheritance, and Annotation statements caused respectively 20, 39, 26, 3, and 2%.

The second group shows the numbers of inheritance related dependencies, which are caused by: (i) access of an inherited variable; (ii) call of an inherited method; and (iii) inheritance by means of an extends or implements statement. The numbers show that on average 12% of the dependencies is inheritance related, of which 3% of type access, 6% of type call, and 3% of type inheritance.

The third group shows the numbers of inner class related dependencies. A dependency was marked as such, if the from-class or to-class is an inner class (or if both classes are). The numbers show that on average 9% of the dependencies is inner class related.

6.2.1. Frequency of hard-to-detect types of dependency. Ten dependency types, shown in italics in Table XII, were hard-to-detect by several tools in our tests. The hard-to-detect types in the first group of Table XII represent 39% of all dependencies in the four systems: Import (10%), Declaration, local variable (7%), Access, constant variable (1%), Access, object reference, direct (14%), Access, object reference, indirect (5%), and Annotation (2%).

The hard-to-detect types in the second group, inheritance related dependencies, total to 8% on average of the dependencies in the four systems: Access of an inherited variable, indirect, (2%), Call of an inherited method, indirect (4%), and Inheritance, indirect (2%). The total of 8% may be added to the total of hard-to-detect dependencies in the first group, which makes 47% of potentially hard-to-detect dependencies, because there is no overlap between the types of hard-to-detect dependencies in the first and second group.

The third group concerns inner class related dependencies, of which all instances in our test were hard to detect by several tools. Nine percent of all dependencies fall within this group. However, this number may not be added to the sum of the hard-to-detect dependencies of the other groups, because there may be an overlap.

6.3. Comparison results of FreeMind test

Finally, we compared the average analysis result of the four subject systems in Table XII with the results of the FreeMind system as a whole, and with the class plugins.script.ScriptingEngine, on which the FreeMind test focused. Table XIII shows that the distribution of the 44,146 dependencies in the FreeMind system over the dependency types differs only a little from the average distribution in the reference systems, the other four subject systems. Main difference is that the FreeMind

DOI: 10.1002/spe

Table XIII. Dependency types of ScriptingEngine compared to freemind and other systems.

Dependencies	Type	Scripting engine	Scripting engine %	Freemind %	Reference systems %
All	All	126	100	100	100
	- Direct	92	73	86	84
	- Indirect	34	27	15	16
	Import	10	8	11	10
	Declaration	15	12	21	20
	- Local var.	6	5	7	7
	Call	61	48	40	39
	Access	35	28	25	26
	- Constant v.	12	9	1	1
	- Object ref. direct	16	13	15	14
	- Object ref. indirect	7	6	4	5
	Inheritance	5	4	3	3
	Annotation	0	0	0	2
Inheritance related	All	27	21	10	12
	Access	2	2	2	3
	- Inh. var. indirect	0	0	1	2
	Call	20	16	5	6
	- Inh. meth. indirect	15	12	4	4
	Inheritance	5	4	3	3
	- Indirect	4	3	1	2
Inner class rel.	*All*	14	11	12	9

system contains 12% inner class related dependencies, while the reference systems on average contain 9% inner class related dependencies.

The dependencies shown for class plugins.script.ScriptingEngine are the dependencies included in the FreeMind test, so limited to dependencies to classes and interfaces in the package freemind. The numbers in Table XIII show that ScriptingEngine contains relatively more indirect dependencies to package freemind, more call dependencies, and more inheritance related dependencies; especially more calls of inherited methods. On the other hand, the class contains relatively less declaration dependencies, and no annotation dependencies nor indirect dependencies of type access of an inherited variable. In total, ScriptingEngine contains relatively more hard-to-detect dependencies: 41% in group one, 15% in group two, and 11% in group 3, compared to respectively 39, 8, and 9%.

HUSACCT reported dependencies for all 109 dependency-causing constructs in class ScriptingEngine to package freemind. However, HUSACCT reported 126 dependencies, 17 more, because one construct may cause more than one dependency. The extra dependencies are of the following types: seven instances of Access, object reference, return value (indirect), six instances of Call, instance method, inherited (four direct, two indirect), and four instances of Inheritance, indirect. For example, construct 'extends MindMapHookAdapter' causes not only a direct inheritance dependency to class MindMapHookAdapter, but also four extra indirect inheritance dependencies to classes and interfaces higher up in the inheritance hierarchy.

7. DISCUSSION

In this section, we discuss the key findings, answer the research questions, and discuss identified challenges and their implications.

7.1. Key findings

In our opinion, all tested tools provide useful functionality to perform an architecture compliance check. However, our tests show that all 10 tools could improve the accuracy regarding dependency and violation reporting, though in varying degrees. A summarizing overview of the findings of our tests is provided in Table XIV, which shows a relative comparison of the tools with respect to the tested characteristics. The subsections below elaborate on these findings and answers the research questions.

DOI: 10.1002/spe

Table XIV. Relative comparison of the tools on the tested characteristics (the scales are explained in the related subsections).

	ConQAT	Dependometer	dTangler	JITTAC	Lattix	Macker	SAVE	Sonar ARE	Sonargraph architect	Structure101
Accuracy dependency detection										
Benchmark: Direct dependencies	*	***	**	***	**	**	*	**	***	***
Benchmark: Indirect dependencies	*	**	*	**	*	*	*	*	*	***
Freemind: Detected classes	*	**	*	***	*	**	**	**	**	**
Freemind: Detected dependencies				***	*		**		***	**
Accuracy dependency messages										
Graphical overview of violations	**		**	**	**		**		**	**
Exactness of dependency messages	*	*	*	***	**	*	**	**	***	**

Although this study includes a tool test regarding ACC support, we do not advise on a 'best' tool. To remain objective, we refrained from this. Accuracy is only one of several qualities that should be considered in the course of a selection process of an ACC-tool. For instance, some tools offer only a limited set of functionality, while others provide a rich set as shown in Table I, especially the commercial tools.

7.1.1. Accuracy of dependency detection and violation reporting. Our study shows that all 10 tools were able to detect dependencies established by basic constructs, like method calls and type declaration. However, the test results show also that significant numbers of dependencies were not reported, even by the best scoring tool. Consequently, the answer to research question RQ1 (Do ACC tools find all the dependencies between modules in the software?) is negative. Numerous false negatives were identified, so all tools may improve on the sensitivity regarding dependency detection. The answer to RQ2 is also negative, because we found no differences in the sets of reported dependencies and reported violations. If a tool was able to detect a dependency, then it was also able to report the dependency if it violated an architectural rule. With regard to false positives, the tested tools performed well; no tool reported false positives. Consequently, the answer to research question RQ3 is negative.

The Benchmark test showed that no tool in the test was able to detect all included dependency types, although several tools performed well. On the average, the 10 tools detected 77% of the dependency types in the test-software: 83% of the 25 direct types and 60% of the 9 indirect types. The 10 tools differ considerably in their ability to detect all types of dependencies included in our test. JITTAC and Structure101 detected the most direct and indirect dependency types; both 31 out of 34 types (91%). On the other side, ConQAT and SAVE detected a total of respectively 21 and 18 dependency types (62 and 53%). Table XIV summarizes the results, based on the following scales: Direct dependencies * = 0–79%, ** = 80–89%, *** = 90–100%; Indirect dependencies: * = 0–59%, ** = 60–79%, *** = 80–100%.

The FreeMind test delivered results regarding the accuracy of dependency detection at the level of depended-upon classes and at the more detailed level of dependency constructs within the code. First, all tools were able to report violations at the level of 'from-class makes use of to-class'. However, only one of the 10 tools (JITTAC) reported usage of all seventeen classes used by class ScriptingEngine, while the least well-performing tools reported 12 classes only. On average, the 10 tools were able to report 82% of the 17 depended-upon classes. Table XIV summarizes the results, based on the following scales: Detected classes * = 0–79%, ** = 80–89%, *** = 90–100%.

Second, none of the tools was able to detect dependencies for all 109 constructs within class ScriptingEngine to package 'freemind'. On the average, 78 of 109 dependencies (72%) were reported. However, the five tools within this test (the other five tools did not report detailed enough information at the level of dependencies) differed considerably in their performance. JITTAC and Sonargraph performed relatively well and reported respectively 91 dependencies (83.4%) and 90 dependencies (82.6%), while SAVE reported 75 dependencies (69%), Structure101 73 (67%), and Lattix 63 (58%). These numbers will not be higher for the other tools, as far as we were able to ascertain, based on the reported depended-upon classes in the violation reports of these tools. Table XIV summarizes the results, based on the following scales: Detected dependencies: * = 0–59%, ** = 60–79%, *** = 80–100%.

7.1.2. Exactness of violation and dependency messages. The FreeMind test revealed that the 10 tools differ considerably in the way violations and dependencies are reported, how many messages are reported, and how much information is reported in the messages. Consequently, the answer to research question RQ4 is diverse, a few ACC tools report the type and location of violations and dependencies quite exactly, but most tools not. The relevance of facilities to relate dependencies and violations to code is observed in a study [34] where professional architects of five systems have been using an ACC-tool.

All tools, except Dependometer, Macker, and Sonar ARE, provide diagrams in which violations are shown at the level of packages and classes. Some of these tools even provide an indication of the quantity of underlying dependencies, by means of a number or by the thickness of a line. In our

DOI: 10.1002/spe

opinion, such an indication of the severity of a violation is relevant information for architects and management; information that also should be included in violation reports. Table XIV summarizes the results. In case graphical support is provided, two asterisks are shown.

We regard the exactness of a dependency message to be high, if the message helps to locate the dependency causing code construct accurately in the source code. Four tools (ConQAT, Dependometer, dTangler and Macker) only provide information on the from-class and the used to-class. The other tools provide more information: SAVE and Structure101 provide also the names of the invoking method at the from-side and the invoked method at the to-side, while JITTAC, Lattix, Sonar ARE, and Sonargraph Architect indicate the line number. JITTAC and Latix even indicate the position of the dependency-causing construct in the line. Table XIV summarizes the results. One asterisk is shown, in cases where only from-class, to-class information is provided. Two asterisks are shown, in cases where also method from, method to information is provided. Three asterisks are shown, in cases where the line number is provided, and/or the position of the dependency causing construct in a line. For reason of usability issues, described in Section 5, we have valued Lattix and Sonar ARE with two instead of three asterisks.

7.1.3. Hard-to-detect dependency types and their frequency in open source systems. The answer to research question RQ5 is positive: yes, there are hard-to-detect types of dependencies. We identified 10 dependency types of which several tools failed to report instances of dependencies. Analysis of the number of dependencies per type in open source systems has yielded interesting data. The average distribution of dependencies in four reference systems over the six main types (import, declaration, call, access, inheritance, annotation) is respectively 10, 20, 39, 26, 3, and 2%. Below is a summary of the findings related to hard-to-detect types of dependencies.

- The 10 hard-to-detect types of dependencies in our test account for at least 47% of the dependencies in the reference systems. Inner class related dependencies, also hard-to-detect, are not included in this percentage, because there may be an overlap with the already included dependencies.
- A considerable fraction of the dependencies within the reference systems is inheritance related, namely 12% on average, while 3% of the dependencies is inheritance related and indirect.
- A considerable fraction of the dependencies within the reference systems is inner class related, namely 9% on average.
- A considerable fraction of the dependencies within the reference systems is indirect, namely 16% on average. In the Benchmark test, only 60% of the indirect dependencies were detected, on average.

7.2. Challenges in dependency detection

Based on the results of the Benchmark test and FreeMind test, we have identified challenges in dependency detection. Analysis of the most common shortcomings in dependency detection revealed the challenges, which are discussed below.

C1: *Report dependencies accurately in case of inheritance structures*

The test results show that inheritance structures frequently hamper the accuracy of dependency detection. This finding is relevant, because our analysis of four reference systems showed that 12% of all reported dependencies were inheritance related. Furthermore, three hard-to-detect types of dependency are inheritance related. Together, these three types account for 8% of all the reported dependencies.

The results of the Benchmark test show that only three of the 10 tools were reporting a dependency on the super class in case of a call of an inherited method, or in case of access of an inherited variable. These three tools reported a dependency of these types as a dependency to the super class where the method is actually implemented, while the other seven tools reported a dependency to the addressed subclass, but not to the super class where the method was implemented. Moreover, the results of the FreeMind test show that many method calls of inherited instance methods were not detected at all (43%, on average);

DOI: 10.1002/spe

neither as dependency on the addressed subclass, nor as dependency on the super class where the method is actually implemented.

The relevance for ACC is considerable. In case of a compliance check, the seven tools will fail to report a violation if the used subclass is part of an allowed-to-use module, while the super class that has implemented the called method or accessed variable, is part of a not-allowed-to-use module. In such cases, a strong dependency stays unnoticed.

Finally, no tool reported indirect inheritance dependencies. Again, the relevance for ACC is considerable, because no tool reported a violation in case the super class of the from-class was part of an allowed-to-use module, while the super class of this parent super class in the inheritance hierarchy was part of a not-allowed-to-use module. This appears as inconsistent behavior of the tools, because all tools reported a violation in case the first super class of the from-class was part of a not-allowed-to-use module. In our opinion, a violating indirect inheritance relation should be reported, because it marks a strong dependency, and changes may have substantial consequences.

C2: *Report dependencies accurately in case of inner classes*

The test results show that inner classes also may hamper the accuracy of dependency detection. This finding is relevant, because our analysis of four reference systems showed that 9% of all reported dependencies were inner class related.

The results of the FreeMind test show that four tools reported no dependency at all on inner classes, while fourteen violating dependencies on inner classes were present in the code of class ScriptingEngine. In the majority of these cases, a dependency to the outer class instead of the inner class is reported, with as consequence a diminished traceability to the related code constructs in the source code. Furthermore, dependencies between inner classes of the same outer class will not be reported.

C3: *Report relevant object references*

The results of the Benchmark test show that seven tools had problems with the detection of dependencies of type 'Variable access, object reference'. Dependencies of this type are frequently included in the code as parameter values (arguments), or reference variables within if clauses.

Our analysis of four reference systems showed that 14% of all reported dependencies were of this type.

The results of the FreeMind test show the practical implications: two of the five tools in this test missed all sixteen dependencies of this type. These sixteen missed object references represented 15% of the 109 dependency causing constructs in class ScriptingEngine.

In our opinion, it is a good practice to filter out object and type references, which precede method calls and variable access. Most tools (including HUSACCT) do, because a dependency message for such a reference doubles with the dependency on the same type for the call or access. However, a reference needs to be reported in case of standalone references, e.g. when an object is passed as a parameter value.

C4: *Report information that is missing in compiled files*

We encountered several situations where tools failed to report dependencies accurately, just because information in source files is removed from the compiled files. Tools that analyze compiled files only, were not able to report dependencies of three dependency types: (i) dependencies caused by import statements; (ii) dependencies caused by declaration statements of not initialized local variables; and (iii) dependencies caused by access of constant variables (instance and class).

Import dependencies were reported by only two tools in the Benchmark test. In our opinion, import statements should be reported, because they cause coupling, although weak. Reporting import dependencies enhances the accuracy of violation reporting in these cases where a tool fails to report dependencies of specific types. In a number of situations in the Benchmark test, SAVE missed a dependency of a specific type, but reported a correct violation message at class level, merely based on the import statement.

Finally, the exactness of the messages with respect to the line number is diminished if a tool does not make use of source code. For example, in the FreeMind test, the line numbers of

DOI: 10.1002/spe

dependencies reported by Lattix (which makes use of compiled files only) were by far not as accurate as the line numbers reported by Sonargraph (which makes use of source files, in addition to compiled files).

8. THREATS TO VALIDITY

To discuss the validity of the results of our laboratory experiments, we make use of the four validity threats, as described and defined by Wohlin et al. [35].

8.1. Construct validity

Construct validity is concerned with the relation between theory and observation. The experiment should be suitable to answer the research question, which in our case means that the experiments should be suitable to answer the main question 'How accurate do ACC-tools report dependencies and violations against dependency rules?' We have ensured the construct validity in several ways. First, by starting with an inventory of common dependency types in object oriented code, on which we have based the test cases of the Benchmark test. The set of dependency types included in our Benchmark test is no random set, but a carefully chosen set of 34 types. It is large enough to assess the sensitivity of the tools, but we do not claim that our classification of dependency types is complete, or that our test cases cover all types of dependency causing code constructs. We have covered a wide range of common code constructs, but dependencies may be established in other ways. For instance, we have no variables with generic types in our test cases, and we have included only one type of annotation.

Second, we have taken care that the code constructs in each test case of the Benchmark test are specific for the related dependency type. Furthermore, we have taken care that false negatives could not be caused by code constructs that were not specific to the related dependency type.

Third, to answer research questions RQ1 – RQ3, we have scored for each test case not only the ability of a tool to report a dependency, but also the ability to report a violation.

Fourth, we have complemented the Benchmark test with the FreeMind test, and we have cross-checked the results of both tests, as described under the results of the FreeMind test.

8.2. Internal validity

Internal validity is threatened, if certain influences have affected the results, without the researcher's knowledge. In the context of our experiments, the internal validity may be threatened by inclusion of problematic or very uncommon code constructs in the Benchmark test or FreeMind test.

In our opinion, the strict design of our custom-made Benchmark test strengthens the internal validity. In this test, each test case is specific for one dependency type, and each test case has one separate from-class in the test code. This approach proved to be valuable, especially to test tools that provide only messages with a low level of exactness: with no more information than the from-class and to-class. Another aspect in favour of the internal validity of the Benchmark test is that all test cases were detected by at least one tool, except in case of indirect 'Inheritance, extends-implements variations' (although two of the three cases represented quite common situations).

The FreeMind test adds to the internal validity, because it contains several code variations not included in the Benchmark test. We used it to validate and extend the Benchmark test. The FreeMind test showed in several cases that a tool might fail to detect a dependency in a complex real life application, while it is able to detect a dependency of the same type in a simpler situation within the same application, or in the relatively simple code of the Benchmark test application.

8.3. Conclusion validity

Conclusion validity reflects on the relationship between the treatment and the outcome of the experiment. In favour of conclusion validity, no statistical operations were needed to interpret the results of our experiments. The research questions could be answered straightforwardly, based on the results of the Benchmark test and FreeMind test. Please, remind that we scored mildly, as described

DOI: 10.1002/spe

before, with as consequence that the presented numbers of detected dependencies could be too high in case of tools that report message with a non-optimal accuracy.

To secure the validity of the identified challenges in dependency detection, we have substantiated each challenge by means of data and arguments.

8.4. External validity

External validity reflects the extent to which the experiment results may be generalized. Because we did not work with a randomized selection of tools, our study can be characterized as a quasi-experiment, according to Wohlin et al. [35]. Consequently, our findings may not be generalized to other tools, even though we tested 10 tools in a small market. Also, be aware that our findings may not be generalized to newer versions of the tested tools; the performance of the tools may improve. Furthermore, our findings are limited to Java code analysis and should not be generalized to tools that analyze code of other programming languages.

8.5. Comparison of the frequency of dependencies per type

In favor of the internal, external and conclusion validity, we have compared the frequency of dependencies per type in the FreeMind system and its class ScriptingEngine to the average of four reference systems. In our opinion, this comparison confirms that the FreeMind system and its class ScriptingEngine are suitable for research on the accuracy of dependency detection. The system as a whole contains dependencies of many different types and in proportion to the average percentages of the reference systems. The same applies for class ScriptingEngine, although one should keep in mind that this class contains 10% more hard-to-detect dependencies than the reference systems on average. Even so, the wide variety of dependency-causing constructs, including a set of complex constructs, makes it an appropriate subject class for the test.

However, limitations apply to the validity of the analysis results of the four reference systems and FreeMind. First, the external validity is limited, because the reference systems are all open source, and the sample size is small. Second, with respect to conclusion validity, we cannot guarantee that all dependencies in the reference systems are reported and that all dependencies of a type are reported. We have ensured the validity of the analysis results by upgrading HUSACCT to the level (and beyond) that all test cases in the Benchmark test and all dependencies in ScriptingEngine were reported. However, because many code variations are possible, some variations may not be reported. Furthermore, deficiencies may be present in HUSACCT itself or in the included open source (ANTLR based) lexer and parser functionality. For instance, a small percentage of the classes is skipped by the parser, because of unexpected (and often erroneous) code in these classes.

9. RELATED WORK

Callo Arias et al. [25] state that dependency analysis approaches that identify structural dependencies have a high degree of accuracy. Our research outcome shows that it is appropriate to be aware of the limitations of the tools used. Practitioners and academics rely on tools for their work. It is not hard to get impressed by the output of these tools, but it is hard to get an impression of what is missing in the output of a tool. Our study demonstrates that the tested tools will not always provide a 100% accurate output. Other comparative tool studies also show that static analysis tools and techniques are not always accurate. For instance, Sutton and Maletic compared four tools that reverse engineer C++ source code into UML models [36]. The numbers of recovered classes and relationships differed by about 20% and much more for attributes, operations, and generalizations.

Rutar et al. [37] compared five bug finding tools for Java, and they reported false positives, false negatives, redundant warnings and only 15–33% overlap between the tools. Compared to the set of bug finding tools, the ACC-tools in our test perform better, with no false positives and no redundancy, but with differences in output and quite a number of false negatives.

According to Binkley [11], source code analysis is impeded by the complexities of modern programming languages. Barowski and Cross [38] pay special attention to dependencies on virtual members and on synthetic methods in their paper on the extraction and use of class dependency information for Java.

DOI: 10.1002/spe

Our study confirms that their special attention is justified, because these types of dependencies (on super classes and inner classes) are involved in many unreported dependencies and violations.

Another topic in their paper is source file versus class file based dependency extraction, and they describe some differences between both forms. For their own tool, they choose for class file based extraction. We do not object to this choice, but we advise, based on our study, to include source code too in the analysis of ACC-tools. In order to optimize the accuracy of the tool with respect to import statements, constant variables, local variables, and the exact position of a dependency-causing construct in the source code.

The FreeMind system has been used in several other scientific studies. We compared our analysis results with these studies, but we did not find overlap, except for comparison on the number of packages and classes of FreeMind version 0.9.0. Emanuel and Surjawan analyzed all versions of FreeMind to illustrate the use of their revised Modularity Index [39]. Their counts of version 0.9.0 match our counts quite closely. We counted 58 packages and 853 classes (including inner classes, but excluding anonymous classes), which is around 10% more than their counts. The difference may be explained by the fact that Emanuel and Surjawan analyzed compiled code, while we analyzed source code with inclusion of test files. Zoller and Schmolitzky used Freemind 0.9.0 also in a study [40], however they counted only 445 classes. On the other side, Arlt et al. counted 1,362 classes 41, much more than reported in the other studies. Summarizing, we noted large difference in class counts, while the counted numbers of NCLOC differed not more than 25% between the three studies.

With respect to our analysis data of the four reference systems, we found some interesting studies. Tempero et al. focused on the use of inheritance in Java systems in two empirical studies [42, 43] of more than 90 open-source systems. They found high levels of use of inheritance, with about three out of four types being defined using inheritance. Furthermore, they found that the inheritance structures are used actively, also for what we typified as access of an inherited variable or call of an inherited method. In line with their research, our study has revealed a high percentage of inheritance related dependencies in the four reference systems. On average, 9% of the dependencies are caused by access of inherited variables and calls of inherited methods.

Tempero conducted another large study [44] to investigate whether the advice is followed to avoid non-private fields. It is good practice to prevent usage of attributes of other classes, because it compromises encapsulation [45]. The results of his study indicate that it is not uncommon (albeit not that terribly common) to declare non-private fields. In line with his findings, our study shows that access of an attribute of another class accounts for about 5% of the dependencies in the four reference systems.

Dyer et al. conducted a large-scale empirical study on the adoption of Java language features [46]. With respect to annotations, these results showed that annotations were among the most used new features of the last three Java versions. However, they noted a relative lack of custom annotations. In line with their work, our study showed that annotations with a reference to another type (internal or external) accounted for 2% of the dependencies in the four reference systems.

10. CONCLUSION

ACC relies on the support of tools to define modules and rules, analyze the code, check the compliance, and report violations to the rules. In this study, we have investigated to which extent static ACC-tools report dependencies and violations accurately. We classified 34 common dependency types, prepared a Benchmark test, and tested 10 tools based on this Benchmark test. In addition, we have tested these tools based on the program code of open source system FreeMind, which we used to test the ability of the tools to report all depended-upon classes, all dependency-causing constructs, and all the information needed by the tool-user to locate dependency-causing constructs in the source code.

10.1. Answers to the research questions

We started our study with the following question in mind: How accurate do ACC-tools report dependencies and violations against dependency rules? This main question was decomposed into four research questions, which are answered as follows:

DOI: 10.1002/spe

RQ1: *Do ACC tools find all the dependencies between modules in the software (no false negatives)?*
No, the 10 tools detected on average 77% of the dependency types in the Benchmark test; 83% of the 25 direct types, and 60% of the 9 indirect types. Furthermore, the tools detected on average 72% of the 109 constructs with dependencies in a class of FreeMind. All 10 tools were able to detect dependencies established by basic constructs, like method calls and type declaration. However, our study showed also that relevant numbers of violating dependency constructs were not reported. For example, in the FreeMind test, the tool with the lowest scores missed 46 out of 109 constructs, while even the best scoring tool missed 18 constructs. Consequently, all tools may improve the accuracy of dependency detection.The tools differ considerably in their ability to detect all types of dependencies. For instance, in the Benchmark test, JITTAC and Structure101 detected 91% of all *dependency types*, while ConQAT and SAVE detected respectively 62 and 53%. In the FreeMind test, JITTAC, and Sonargraph Architect reported dependency messages for respectively 83% of the 109 violating code constructs, while Structure101 and Lattix detected 67 and 58% respectively.

RQ2: *Do ACC tools report all the violating dependencies in the software (no false negatives)?*
No, during the tests no cases were noticed, where a tool detected a dependency, but failed to report it in case it violated an architectural rule. Consequently, the answer to the previous research question details the answer to this one too.

RQ3: *Do ACC tools report non-violating dependencies as violations (false positives)?*
No, during the Benchmark test no false positives were detected. No tool interpreted allowed dependencies in the program code as violating dependencies. In addition, nearly no errors in the violation messages were identified during the tests; only a few messages contained incorrect information.

RQ4: *Do ACC tools report the exact type and location of violations and dependencies?*
The answer to this research question is diverse. Only four tools provide a dependency type that differentiates between types of usage, like declare, access, or call. The number and names of dependency types vary per tool. Furthermore, only a few tools report the location of dependencies exactly. All tools report violations to dependency rules at the level of from-class, to-class, but at this level of abstraction, one message may represent several dependencies. Six tools also provide dependency details in reports or IDE plug-ins, but not always precisely enough to localize dependencies discretely.

RQ5: *Are there types of dependencies, which proved hard-to-detect by several tools?*
Yes, based on our tests, we identified 10 hard-to-detect types of dependencies, which were each missed by several tools. To substantiate the relevance of our findings, we performed an analysis of the number of dependencies per dependency type in five open source systems. The analysis results revealed that the hard-to-detect types of dependencies account for at least 47% of the dependencies in the reference systems.

10.2. Challenges

Because significant percentages of false negatives were revealed per tool during the Benchmark test and FreeMind test, we have analyzed the test results in detail. As an outcome, we have identified and described four challenges in dependency detection. In summary, the test results revealed that the most common shortcomings in dependency detection, encountered in our study, have to do with: (i) inheritance structures; (ii) inner classes; (iii) object references; and (iv) missing information in compiled files.

Our tests have shown that inheritance structures and inner classes hamper the accuracy of violation reporting in many cases. A dependency caused by usage of inherited methods or variables is often not reported, and if reported, than mostly as a dependency on the accessed subclass only and not on the super class that implements the method or variable. Furthermore, usage of an inner class is frequently not reported at all, and if reported, it is most often reported as a dependency on the outer class instead of the inner class, which diminishes the traceability in the source code.

DOI: 10.1002/spe

10.3. Future work

Benchmark test are relevant to advance the state of the arts of tools. We have developed and applied initial tests to benchmark tools on the accuracy of dependency detection and reporting. The testware of our Benchmark test and FreeMind test is available at the following address: https://github.com/SaccToolTests/SacctAccuracyTest. Future work can be based on these tests and be aimed at the development and application of more comprehensive benchmark tests for a larger set of dependency types, a wide variety of tools, or a wide variety of programming languages.

Research on the performance and improvement of dependency analysis is relevant for practitioners and academics, because dependency analysis supplies the data not only for ACC, but also for architecture reconstruction, metrics, and architecture restructuring advice.

ACKNOWLEDGEMENTS

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.

REFERENCES

1. Murphy GC, Notkin D, Sullivan K. Software reflexion models. *ACM SIGSOFT Software Engineering Notes* 1995; **20**(4): 18–28. DOI: 10.1145/222132.222136.
2. de Silva L, Balasubramaniam D. Controlling software architecture erosion: a survey. *Journal of Systems and Software* 2012; **85**(1): 132–151. DOI: 10.1016/j.jss.2011.07.036.
3. Knodel J, Popescu D. *A Comparison of Static Architecture Compliance Checking Approaches*. In: Working IEEE/IFIP Conference on Software Architecture. IEEE Computer Society, Washington, DC, USA, 2007; 12–21. DOI:10.1109/WICSA.2007.1.
4. Ducasse S, Pollet D. Software architecture reconstruction: a process-oriented taxonomy. *IEEE Transactions on Software Engineering* 2009; **35**(4): 573–591. DOI: 10.1109/TSE.2009.19.
5. Passos L, Terra R, Valente MT, Diniz R, Das Chagas Mendonca N. Static architecture-conformance checking: an illustrative overview. *IEEE Software* 2010; **27**(5): 82–89. DOI: 10.1109/MS.2009.117.
6. Van Eyck J, Boucké N, Helleboogh A, Holvoet T. Using code analysis tools for architectural conformance checking. *In: Proceeding of the 6th International Workshop on SHAring and Reusing Architectural Knowledge—SHARK '11*. ACM, New York, NY, USA, 2011; 53–54. DOI:10.1145/1988676.1988687.
7. Gleirscher M, Golubitskiy D, Irlbeck M, Wagner S. Introduction of static quality analysis in small- and medium-sized software enterprises: experiences from technology transfer. *Software Quality Journal* 2014; **22**(3): 499–542. DOI: 10.1007/s11219-013-9217-z.
8. Canfora G, Di Penta M, Cerulo L. Achievements and challenges in software reverse engineering. *Communications of the ACM* 2011; **54**(4): 142–151. DOI: 10.1145/1924421.1924451.
9. ISO/IEC. 25010 Systems and software engineering—system and software product quality requirements and evaluation (square)—system and software quality models. 2011.
10. Pruijt L, Köppe C, Brinkkemper S. Architecture compliance checking of semantically rich modular architectures: a comparison of tool support. In: *2013 IEEE International Conference on Software Maintenance*. IEEE Computer Society, Washington, DC, USA, 2013; 220–229. DOI: 10.1109/ICSM.2013.33.
11. Binkley D. Source code analysis: a road map. In: *Future of Software Engineering (FOSE '07)*. IEEE Computer Society, Washington, DC, USA, 2007; 104–119. DOI: 10.1109/FOSE.2007.27.
12. Sangal N, Jordan E, Sinha V, Jackson D. Using dependency models to manage complex software architecture. In: *Conference on Object Oriented Programming Systems Languages and Applications*. ACM, New York, NY, USA, 2005; 167–176. DOI:10.1145/1103845.1094824.
13. Bischofberger WR, Kühl J, Löffler S. Sotograph—a pragmatic approach to source code architecture conformance checking. In: Oquendo F, Warboys B, Morrison R, eds. *European Workshop on Software Architecture*. Vol 3047. *Lecture Notes in Computer Science*. Springer: Berlin/Heidelberg, 2004; 1–9. DOI:10.1007/b97879.
14. Huynh S, Cai Y, Song Y, Sullivan K. Automatic modularity conformance checking. In: *Proceedings of the 13th International Conference on Software Engineering—ICSE '08*. ACM, New York, NY, USA, 2008; 411–420. DOI: 10.1145/1368088.1368144.
15. Koschke R, Frenzel P, Breu APJ, Angstmann K. Extending the reflexion method for consolidating software variants into product lines. *Software Quality Journal* 2009; **17**(4): 331–366. DOI: 10.1007/s11219-009-9077-8.
16. Deissenboeck F, Heinemann L, Hummel B, Juergens E. Flexible architecture conformance assessment with CONQAT. In: *2010 ACMIEEE 32nd International Conference on Software Engineering*. Vol 2. IEEE Computer Society, Washington, DC, USA, 2010; 247–250. DOI:10.1145/1810295.1810343.
17. Adersberger J, Philippsen M, Reflex ML. UML-based architecture-to-code traceability and consistency checking. In: *5th European Conference on Software Architecture*. Springer, Berlin/Heidelberg, 2011; 344–359. DOI: 10.1007/978-3-642-23798-0_37.

18. Haitzer T, Zdun U. DSL-based support for semi-automated architectural component model abstraction throughout the software lifecycle categories and subject descriptors. In: *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures*. ACM, New York, NY, USA, 2012; 61–70. DOI: 10.1145/2304696.2304709.
19. Pruijt L, Köppe C, van der Werf JMEM, Brinkkemper S. Accuracy test of software architecture compliance checking tools—test instruction. http://www.cs.uu.nl/research/techreps/UU-CS-2015-020. Published 2015. [accessed December 17, 2015].
20. Pruijt L, Köppe C, Brinkkemper S. On the accuracy of architecture compliance checking: accuracy of dependency analysis and violation reporting. In: *21st International Conference on Program Comprehension*. San Francisco, CA, USA: IEEE Computer Society, Washington, DC, USA, 2013; 172–181. DOI:10.1109/ICPC.2013.6613845.
21. Buckley J, Mooney S, Rosik J, Ali N. JITTAC: a just-in-time tool for architectural consistency. In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Washington, DC, USA, 2013; 1291–1294. DOI:10.1109/ICSE.2013.6606700.
22. Perry DE, Wolf AL. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes* 1992; **17**: 40–52. DOI: 10.1145/141874.141884.
23. Clements P, Bachmann F, Bass L, et al. Documenting Software Architectures: Views and Beyond. Pearson Education, 2010.
24. Podgurski A, Clarke LA. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering* 1990; **16**(9): 965–979. DOI: 10.1109/32.58784.
25. Callo Arias TB, Spek P, Avgeriou P. A practice-driven systematic review of dependency analysis solutions. *Empirical Software Engineering* 2011; **16**(5): 544–586. DOI: 10.1007/s10664-011-9158-8.
26. Feilkas M, Ratiu D, Jurgens E. The loss of architectural knowledge during system evolution: an industrial case study. In: *2009 IEEE 17th International Conference on Program Comprehension*. IEEE Computer Society, Washington, DC, USA, 2009; 188–197. DOI:10.1109/ICPC.2009.5090042.
27. Ko AJ, Myers BA, Member S, Coblenz MJ, Aung HH. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering* 2006; **32**(12): 971–987. DOI: 10.1109/TSE.2006.116.
28. Terra R, Valente M. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience* 2009; **39**(12): 1073–1094. DOI: 10.1002/spe.
29. Saraiva J, Soares S, Castor F. Assessing the impact of AOSD on layered software architectures. In: *European Conference on Software Architecture*. Springer: Berlin/Heidelberg, 2010; 344–351. DOI:10.1007/978-3-642-15114-9_27.
30. Stafford JA, Wolf AL. Architecture-level dependence analysis for software systems. *International Jounal of Software Engineering and Knowledge Engineering* 2001; **11**(4): 431–451. DOI: 10.1142/S021819400100061X.
31. Skansholm J. *Java from the Beginning*, 2nd edn. Addison-Wesley, 2004.
32. Oracle. Java documentation—language basics. http://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html. [accessed December 17, 2015].
33. Pruijt L, Köppe C, van der Werf JM, Brinkkemper S. HUSACCT: architecture compliance checking with rich sets of module and rule types. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering—ASE '14*. ACM, New York, NY, USA, 2014; 851–854. DOI:10.1145/2642937.2648624.
34. Buckley J, Ali N, English M, Rosik J, Herold S. Real-time reflexion modelling in architecture reconciliation: a multi case study. *Information and Software Technology* 2015; **61**: 107–123. DOI: 10.1016/j.infsof.2015.01.011.
35. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. *Experimentation in Software Engineering*. Springer: Berlin/Heidelberg, 2012.
36. Sutton A, Maletic JI, Ohio K. Mappings for accurately reverse engineering uml class models from C++. *Information and Software Technology* 2007; **49**(3): 212–229. DOI: 10.1109/WCRE.2005.21.
37. Rutar N, Almazan CB, Foster JS. A comparison of bug finding tools for Java. In: *15th International Symposium on Software Reliability Engineering*. IEEE Computer Society, Washington, DC, USA, 2004; 245–256. DOI:10.1109/ISSRE.2004.1.
38. Barowski L, Cross J. Extraction and use of class dependency information for Java. In: *Reverse Engineering, 2002*. Ninth Working Conference on. IEEE Computer Society, Washington, DC, USA, 2002; 309–315. DOI:10.1109/WCRE.2002.1173088.
39. Emanuel AWR, Surjawan DJ. Revised modularity index to measure modularity of OSS projects with case study of Freemind. *International Journal of Computer Applications* 2012; **59**(12): 28–33. DOI: 10.5120/9602-4227.
40. Zoller C, Schmolitzky A. Measuring inappropriate generosity with access modifiers in Java systems. *In: Joint Conf. of Int. Workshop on Software Measurement and Conf. on Software Process and Product Measurement*. IEEE Computer Society, Washington, DC, USA, 2012; 43–52. DOI:10.1109/IWSM-MENSURA.2012.15.
41. Arlt S, Podelski A, Bertolini C, Schäf M, Banerjee I, Memon AM. Lightweight static analysis for GUI testing. In: *Software Reliability Engineering, ISSRE*. IEEE Computer Society, Washington, DC, USA, 2012; 301–310. DOI:10.1109/ISSRE.2012.25.
42. Tempero E, Noble J, Melton H. How do Java programs use inheritance? An empirical study of inheritance in Java software. In: *European Conference on Object-Oriented Programming (ECOOP)*. Springer: Berlin/Heidelberg, 2008; 667–691. DOI:10.1007/978-3-540-70592-5_28.
43. Tempero E, Yang HY, Noble J. What programmers do with inheritance in Java. In: *27th European Conference on Object Oriented Programming*. Springer: Berlin/Heidelberg, 2013; 577–601. DOI:10.1007/978-3-642-39038-8-24.

44. Tempero E. How fields are used in Java: an empirical study. In: *Australian Software Engineering Conference (ASWEC 2009)*. IEEE Computer Society, Washington, DC, USA, 2009; 91–100. DOI:10.1109/ASWEC.2009.19.
45. Wirfs-Brock R, Wilkerson B. Object-oriented design: a responsibility-driven approach. In: *Object-Oriented Programming Systems, Languages and Applications (OOPSLA '89)*. ACM, New York, NY, USA, 1989; 71–75. DOI:10.1145/74878.74885.
46. Dyer R, Rajan H, Nguyen HA, Nguyen TN. Mining billions of AST nodes to study actual and potential usage of Java language features. In: *ICSE 2014 Proceedings of the 36th International Conference on Software Engineering*. ACM, New York, NY, USA, 2014; 779–790. DOI:10.1145/2568225.2568295.