The Journal of Systems and Software 209 (2024) 111916

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

DValidator: An approach for validating dependencies in build configurations☆

Christian Macho *, Fabian Oraze, Martin Pinzger

University of Klagenfurt, Klagenfurt, Austria

ARTICLE INFO

Keywords:
Dependency management
Apache Maven
Answer set programming
Build engineering

ABSTRACT

Reusing components is a well-established practice in modern software engineering and brings many advantages, such as a reduction of development costs and time. However, there are still several problems when reusing software components, such as the management of the dependencies of a project. Modern build systems provide dependency managers to support developers when dealing with dependencies. But even with this tool support, dependency management is an error-prone task which can lead to dependency hell if it gets out of control.

In this paper, we propose DValidator, an approach that considers dependencies on project level and method call level for validating dependencies in build configurations. First, DValidator encodes a project's dependency graph as specified in a build configuration and its call graph into a representation using Answer Set Programming (ASP). Then it uses Clingo as a solver to detect problems with the dependencies in that build configuration. In a preliminary evaluation with four open source Maven projects we show that our approach can detect selected dependency smells in less than eight seconds. Next steps concern the investigation of our approach for automatically improving dependency configurations, such as automatically repairing dependency smells and conflicts.

Editor's note: Open Science material was validated by the Journal of Systems and Software Open Science Board.

1. Introduction

Reusing existing software components for developing applications is best practice and has many benefits, such as reduced development time and the usage of well tested components (Frakes and Kang, 2005). Modern build systems, such as Apache Maven[1] or Gradle,[2] offer various ways to include existing components from software repositories (Valiev et al., 2018). One of the most frequently used ways in CI/CD workflows (Humble and Farley, 2010) is through the dependency management system that resolves, downloads, and includes software components that are declared as dependency, for instance in a build configuration file (Shahin et al., 2017). In turn, projects that are declared as dependency may also have dependencies to other projects which must be obtained by the dependency management system. Note that in this work, we use the Apache Maven terminology and refer to software components as projects.

While reusing software projects has many benefits, it also brings disadvantages and additional maintenance effort (Kerzazi et al., 2014). One of the disadvantages is the total number of projects that are requested and included, because this number can grow exponentially (Bavota et al., 2013), and managing those can become challenging (Capilla et al., 2019b,a; Mäkitalo et al., 2020). Another disadvantage concerns updating dependencies to recent versions which can also become a challenge, especially when many versions need to be kept up to date (Di Cosmo et al., 2008). If these maintenance activities are neglected (Kerzazi et al., 2014), a project can ultimately land in a state that is called the "dependency hell" (Fan et al., 2020; Tanabe et al., 2018; Abate et al., 2020; Chen et al., 2021), indicating that the dependencies require substantial rework and effort to return to a manageable state.

While recent research has contributed towards better support to manage dependencies, the core problem, *i.e.*, selecting the "right" dependencies, is still not solved (Abate et al., 2020; Kikas et al., 2017) and needs further attention. A trending direction to support developers in managing dependencies involves representing them as models or logic programs and using solvers. For example, Gebser et al. (2011a)

☆ Editor: Heiko Koziolek.

* Corresponding author.
E-mail addresses: christian.macho@aau.at (C. Macho), fabian.oraze@aau.at (F. Oraze), martin.pinzger@aau.at (M. Pinzger).
URLs: https://mitschi.github.io/ (C. Macho), https://pinzger.github.io/ (M. Pinzger).

[1] https://maven.apache.org/
[2] https://gradle.org/
[3] https://www.npmjs.com/

https://doi.org/10.1016/j.jss.2023.111916
Received 12 May 2023; Received in revised form 6 October 2023; Accepted 19 November 2023
Available online 28 November 2023
0164-1212/© 2023 The Authors. Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

provide *aspcud*, a tool that supports the dependency management for Linux dependencies. Moreover, Ignatiev et al. (2014) proposed a hybrid approach, combining an exact and an approximate solving strategy to optimize dependency selection. Recently, Pinckney et al. (2023) proposed MaxNPM, an approach that can replace the standard dependency management tool in npm[3] environments with an enhanced version. All three mentioned approaches leverage models and solvers to improve the dependency selection. However, these works only consider dependencies on a project level that are explicitly declared but neglect more detailed dependencies between projects, such as through method calls. Moreover, the produced dependency configurations of these approaches cannot ensure that all method calls are satisfied as they do not take method calls into account. Hence, the solving strategies solely rely on additional information about the (in-)compatibility of the project dependencies.

In this paper, we present DValidator, a novel approach that also considers method calls between projects as dependencies in addition to explicitly declared dependencies on project level. This results in a more comprehensive model that enables a more accurate analysis and validation of projects dependencies. The proposed model is an ASP-based (Gebser et al., 2014) model combined with Clingo (Gebser et al., 2011b) as its solver that is capable of modeling dependency configurations, while respecting more detailed dependencies, such as method calls.

In this work, we focus on investigating the potential of our model to validate existing configurations and to identify problems in dependency configurations, for example, duplicated dependencies. In future work, we plan to investigate the potential of our model to generate high-quality dependency configurations that adhere to the best practices dependency managment rules depicted in our model.

The remainder of the paper is organized as follows. Section 2 provides background information about dependency management and situates the paper with respect to related work. In Section 3, we propose our model and provide a preliminary evaluation in Section 4. Section 5 discusses the limitations of the work and provides an extensive outlook on future work. The paper is concluded in Section 6. Please also note that we provide supplementary material for this work online.[4]

2. Background and related work

In this section, we first provide a brief introduction to build systems and how they manage dependencies. Finally, this section summarizes related work about dependency management and software ecosystems.

2.1. Dependency management with build systems

Modern build systems offer techniques and mechanisms to specify and resolve dependencies to other projects. Projects that are needed are declared as a dependency in the language of the build system (*e.g.*, XML for Apache Maven) and the build system resolves, downloads, and provides the corresponding project to the compiler. Listing 1 shows such a dependency configuration in Apache Maven. In this example, the project declares two dependencies, namely to the `SomeDependency` and the `OtherDependency` projects. Moreover, the transitive dependency `ExcludedDependency` is excluded from being processed when processing the dependencies of the `SomeDependency` project in this case.

The problems with dependencies are manifold. First, dependencies that are not kept updated (Kerzazi et al., 2014) might cause problems when they need to be updated (Cox et al., 2015; Kula et al., 2018). Second, dependencies can also have dependencies to other projects, called transitive dependencies. Such transitive dependencies may cause problematic situations (Soto-Valero et al., 2021b,a). For example, they might lead to a dependency hierarchy that includes multiple versions of a single project. Third, dependencies might be incompatible with each other and consequently fail the build.

Listing 1: Example of a dependency configuration in Apache Maven. Note that only declared dependencies are in the dependency list. Transitive dependencies are derived from the dependency declarations in the build files of the respective projects.

```
<dependencies>
  <dependency>
    <groupId>at.aau</groupId>
    <artifactId>SomeDependency</artifactId>
    <version>2.3.1</version>
    <exclusions>
      <exclusion>
        <groupId>at.aau</groupId>
        <artifactId>ExcludedDependency</artifactId>
      </exclusion>
    </exclusions>
  </dependency>

  <dependency>
    <groupId>at.aau</groupId>
    <artifactId>OtherDependency</artifactId>
    <version>4.0.1</version>
  </dependency>
</dependencies>
```

State-of-the-art build systems offer various mechanisms to avoid such problems with dependencies. One of the provided solutions is to fail the build if conflicting declarations are found and report this to the developers (Abate et al., 2012). Another solution is to prefer dependencies that are declared closer to the root project. For example, if a project directly declares a dependency to a project A, and a transitive dependency B also has a dependency to project A (possibly in a different version), the build system omits the second dependency and includes only the first version of the project A. However, this has the disadvantage that if there are actual calls into the omitted version of library A, the build will still pass but runtime errors will occur. While research is attempting to provide solutions for these problems (Wang et al., 2018, 2019, 2021), state-of-the-art build systems still lack tool support to detect such problems.

2.2. Related work

Dependency management has been studied in various directions. Cox et al. (2015) studied dependency freshness and introduced a metric to measure it. They found that their metric was perceived as useful by developers and that systems with outdated dependencies are more likely to contain vulnerabilities. Kula et al. (2018) investigated a similar direction with an empirical study on library migration. They investigated 4,600 GitHub software projects and additionally found that 69% of the surveyed developers were not aware of their vulnerable dependencies. Moreover, they also stated that library updates are perceived as extra workload. Consequently, library updates are not as common as modern software development would suggest. Bavota et al. (2013, 2015) studied the evolution of dependency management. First, they found that the number of dependencies grows exponentially in a software ecosystem. Second, they identified that developers mostly update dependencies of their projects together with other substantial changes in the code. Finally, Larios Vargas et al. (2020) interviewed developers to identify 26 factors that are important when selecting third-party libraries and categorized them into technical, human, and economical factors. They found that stability, usability, documentation, and the type of license are among the most influential factors when selecting third-party libraries.

[4] https://zenodo.org/record/7928594

Table 1
Dependency smells used to showcase the model in this work.

Smell	Description
Dependency Cycles (DC)	The declared dependencies form a cycle in the dependency graph. This can happen if project A depends on project B and project B depends on project A, for example.
Double Included Dependency (DID)	A dependency that is included more than once. This can either happen if dependencies transitively depend on an already declared dependency or if the dependency declaration is flawed (*i.e.*, they are duplicated entries in the dependency list). Duplicated dependency declarations can also lead to dependency conflicts (Wang et al., 2018, 2019)
Shadowed Dependency (SD)	Many dependency resolution mechanisms, such as the one of Apache Maven, only allow loading one instance of a project. If another version of the project should also be loaded, for example, in a deeper level of the dependency hierarchy, it is shadowed by the higher declared dependency in the dependency hierarchy (Wang et al., 2018, 2019).
Jump Calls (JC)	IDEs often offer all possible calls in the code completion pop-ups to assist developers. This includes calls to methods in transitive dependencies. However, it is a bad practice to use transitive dependencies because the developer is not in control of them (Cao et al., 2022).
Unused Dependency (UD)	Similar to prior studies (Soto-Valero et al., 2021b,a), we detect declared dependencies that are not used, *i.e.*, they never appear in a call hierarchy that originates from the root project.

Besides studying the dependency management from different perspectives, research has also shed light on smells in dependency management. Jafari et al. (2021) studied 1146 Javascript projects to reveal smells in the dependency configuration. They found that dependency smells are still prevalent in Javascript projects seeing two or more smells in 80% of the studied projects. Also other works studied the occurrence of smells in dependency configurations. For example, Soto-Valero et al. (2021b,a) investigated unused dependencies. In their study, they found 2.7%, 15.4%, and 57% of the directly declared, inherited, and transitive dependencies as unused. Moreover, the study shows that developers are basically willing to reduce the amount of unused dependencies. Analogous to unused dependencies, Cao et al. (2022) investigated three main dependency smells and observed their occurrence in Python projects. Mostly, they are introduced because of synchronous updates and collaborative development. Finally, Wang et al. (2018, 2019, 2021) studied dependency conflicts, a more complex type of dependency misconfigurations. They proposed Decca (Wang et al., 2018), a tool that can detect dependency conflicts with a precision of 0.92 and a recall of 0.77. Extending this work, they provided Riddle (Wang et al., 2019) to automatically generate test cases that reveal dependency conflicts. Finally, they studied how such dependency conflicts impact the semantic of a program and proposed Sensor (Wang et al., 2021) to improve the test cases generated by their prior work.

While the related work reported so far studied the problems and evolution of dependencies and their management, there are other works that model dependencies to ensure valid dependency configurations. For instance, Gebser et al. (2011a) proposed an approach for Linux dependency management. They use ASP and the Common Upgradability Description Format (CUDF) (Treinen and Zacchiroli, 2009) to support dependency updates. However, for most build systems this format is not supported. Our approach extends this format by incorporating not only dependencies on a project level, but also on method call level. Ignatiev et al. (2014) aim at optimizing configurations in package managers. They propose a hybrid approach that stops the solving algorithm after a timeout is reached while still returning high-quality configurations. Lastly, Pinckney et al. (2023) proposed MaxNPM, an approach that can be used within the npm environment to use more recent versions of the dependencies. They report better configurations compared to the traditional NPM package manager, *i.e.*, a reduction of vulnerabilities, a higher number of used recent dependencies, and a lower number of dependencies at all.

2.3. Research gap

As reported, much research has been carried out towards improving build systems in general and dependency management in particular. Most studies focus on problems with build systems and improving poorly configured build systems. However, these approaches only consider dependencies on the level of project dependencies and neglect the more detailed dependencies, for example, on the level of method calls. In this new ideas paper, we aim to fill this gap with an approach that incorporates such detailed dependencies to further improve dependency configurations.

3. DValidator Approach

In this section, we propose DValidator, our approach for validating dependency configurations. First, we explain how we define and understand a well-formed dependency configuration based on literature and best practice approaches from industry. Next, we show how we transform the described rules for well-formed configurations into rules and constraints in ASP.

3.1. Well-formed dependency configurations

Many recent research works (Soto-Valero et al., 2021b,a; Claes et al., 2018; Huang et al., 2020; Abate et al., 2012; Decan et al., 2016, 2017) study build configurations and, in particular, configurations of the dependency management system with the focus of detecting smells and identifying misconfigurations. For example, Cao et al. (2022) and Jafari et al. (2021) provide approaches to detect dependency smells (*e.g.*, missing dependencies and unused dependencies) for Python and JavaScript projects, respectively. However, these approaches are bound to a specific programming language and build tool. Moreover, they are unsuitable for generating dependency configuration candidates that do not violate the smell detection rules.

In this work, we aim at a language and build system agnostic approach that is also capable of generating well-formed dependency configurations. As a first step, we identify types of dependency smells studied in related work. Table 1 presents the studied smells, related works that also proposed this smell, and a brief description of each smell. Please note that, for the purpose of this New Trends and Idea Papers track, we only selected a small subset of the overall collected smells and leave the extension of this list of smells to future work. We selected those smells that were frequently studied in recent works and perceived as problematic in industry.

3.2. A model for dependency configurations

With the obtained dependency smells, we set out to encode the rules and constraints for our model. Similar to the idea of Tourwé and Mens (2003), we encode our model as an answer set programming (ASP) problem (Gebser et al., 2014) and use Clingo (Gebser et al., 2011b) as the solver because both are well-established tools and suit our purpose. ASP problems consist of two main ingredients. The first ingredient are the predicates which represent facts that describe the problem instance (in our case the project under investigation consisting of the dependency graph and the call graph). For example, a dependency

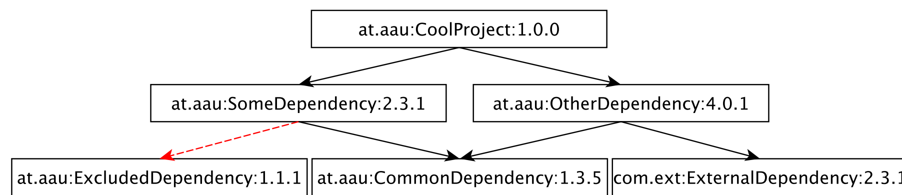

Fig. 1. An example dependency graph depicting the root project and its (transitive) dependencies.

between two projects is given as a fact. The second ingredient are rules and constraints which describe the way how further facts can be derived. For example, the rule to detect unused dependencies states that if a dependency is declared but no method of it is called, then the dependency is marked as an unused dependency.

We present the set of facts and constraints of our proposed model in two parts. The *first part* presents all the facts, *i.e.*, (1) facts that are given by a generator which provides the input for our model, and (2) facts that are deducted from other facts by our model. All the facts stem from data from two sources. Firstly, we leverage the dependency graph of a project that depicts all included projects and the (dependency) relations between them (see Fig. 1). Secondly, we leverage the call graph of a project that consists of all the (transitive) calls that are extracted throughout the included projects. The *second part* provides all the constraints that concern our subset of dependency smells depicted in Table 1. Note, while we describe the mechanisms using the Apache Maven build system and build specification, all the concepts are transferable to any build system and specification (for example Gradle).

Encoding the provided and deducted facts. In the first step, we encode the dependency graph and all its properties. A depende ncy graph is a graph that consists of all the projects that are declared as dependencies of the project and all transitive dependencies, recursively, where the nodes of the graph represent the projects and the edges the dependencies. Fig. 1 shows an example of a sim ple dependency graph in which the `CoolProject` declares two dependencies to `SomeDependency` and `OtherDependency`. The projects `SomeDependency` and `OtherDependency` again define three dependencies to `ExcludedDependency`, `CommonDependency`, and `ExternalDependency`. These are transitive dependencies of `Cool Project`. Moreover, Fig. 1 shows that a project is identified through a triplet *GAV* that consists of an identifier of the company (usually called `groupId` or G), an identifier of the project (usually called `artifactId` or A), and a version number (usually a valid version according to Semantic Versioning,[5] V). This triplet is often called the (Maven) coordinates of a project.[6]

We now explain the facts in detail in Listing 2. Consider the example in Fig. 1, there are six projects in total. Each of them is encoded as project as shown in line 2 of Listing 2. The depicted top project `at.aau:CoolProject:1.0.0` is the project that we want to investigate. We call this project the root project and it is encoded with the fact depicted in line 3. Next, we model the dependencies between two projects as line 4 shows. The first GAV triplet stands for the project that declares the dependency and the second GAV triplet is the declared dependency. In most build systems, the developer can decide to exclude certain transitive dependencies from being resolved,[7] for example, because they may contain a vulnerability, they may not be needed, or because the developer intentionally includes another version of the dependency. To account for this, we encode explicitly excluded projects as shown in line 5. Note that the exclusion fact does not take a version because usually developers want to exclude any version of that project and intentionally declare a dependency to a selected version of that project.

With these facts, we can model dependencies between projects in build systems and formulate rules and constraints that make up well-formed dependency configurations. Such rules include constraints of the listed dependency smells (*e.g.*, cyclic dependencies). However, for modeling particular dependency smells, the dependency graph is not sufficient. For example, to identify an unused dependency, we also need to consider the methods that are provided by a project that the root project depends on, and the corresponding method calls that are performed. We encode each method that a project provides as seen in line 7. G, A, and V stand for the coordinates of the project. `MNAME`, `NPARAM`, and `RET` represent the method name, the number of parameters of this method, and its return type, respectively. A call to a method is shown in line 8. Each end (caller and callee) of a call consists of the same seven properties. Hence, our call fact has 14 properties.

These six facts are extract from the project's artifacts (see Section 4 for information) and are provided to our model. There are also other facts that we derive from the provided facts which are useful to solve any dependency problem with our model. For example, the two dependencies `at.aau:SomeDependency:2.3.1` and `at.aau:OtherDependency: 4.0.1` represent the first level of dependencies which we call the declared dependencies, because they are directly declared by the root project. We encode the first level of dependencies as shown in line 11. This rule states that a dependency is a declared dependency if the declaring project is the root project. Additionally, we provide a fact that represents a transitive dependency relation. A project C is a transitive dependency of a project A if there is a (transitive) dependency from A to another project B and a (transitive) dependency from B to C. Lines 12 to 14 show the facts implementing this rule.

Finally, we need to express that all projects that the root project uses are marked as included projects (*e.g.*, they are on the classpath in Java projects). Line 15 and 16 provide the rules to mark included projects. Again, GAV stands for the project coordinates. The L variable accounts for the level in which the dependency is included. Level $L = 0$ indicates the root project which results in the fact `includedDependency(at.aau, CoolProject, 1.0.0, 0)` given the example in Fig. 1 and Listing 1. Moreover, as a dependency can be potentially included multiple times through the dependency mechanism, we also keep track of the lowest level (*i.e.*, highest declaration in the dependency graph) of each included project. The rule in line 17 defines this by storing the lowest LEVEL of each project with the coordinates G and A. For each included project, we determine the lowest LEVEL value, *i.e.*, the nearest declaration of this project with respect to the root project.

Encoding the constraints. Armed with the provided facts that define the projects' dependency graph and call graph, we now describe a preliminary set of generic rules for well-formed dependency

[5] https://semver.org/

[6] There are several other names for this triplet, such as ID of a project or simply the Maven Triplet.

[7] For example, see https://maven.apache.org/guides/introduction/introduction-to-optional-and-excludes-dependencies.html#dependency-exclusions

Listing 2: List of facts and rules to detect the five selected dependency smells.

```
1  % Given Facts (inputted from a tool)
2  project(G, A, V).
3  rootProject(G, A, V).
4  dependency(from_G, from_A, from_V, to_G, to_A, to_V).
5  excludeProject(G, A, V, ex_G, ex_A).
6
7  method(G, A, V, FQN_CLASS, MNAME, NPARAM, RET).
8  call(FROM_G, FROM_A, FROM_V, FROM_CLASS, FROM_MNAME, FROM_NPARAM, FROM_RET, TO_G,TO_A, TO_V, TO_CLASS, TO_MNAME,
       ↪ TO_NPARAM, TO_RET).
9
10 % Derived Facts:
11 declaredDependency(G, A, V) :- rootProject(ROOT_G, ROOT_A, ROOT_V), dependency(ROOT_G, ROOT_A, ROOT_V, G, A, V),
       ↪ project(G, A, V).
12 transitiveDependency(A_G, A_A, A_V, C_G, C_A, C_V) :- dependency(A_G, A_A, A_V, B_G, B_A, B_V), dependency(B_G, B_A, B_V
       ↪ , C_G, C_A, C_V).
13 transitiveDependency(A_G, A_A, A_V, C_G, C_A, C_V) :- dependency(A_G, A_A, A_V, B_G, B_A, B_V), transitiveDependency(
       ↪ B_G, B_A, B_V, C_G, C_A, C_V).
14 transitiveDependency(A_G, A_A, A_V, C_G, C_A, C_V) :- transitiveDependency(A_G, A_A, A_V, B_G, B_A, B_V),
       ↪ transitiveDependency(B_G, B_A, B_V, C_G, C_A, C_V).
15 includedProject(G, A, V, 0) :- rootProject(G, A, V).
16 includedProject(CHILD_G, CHILD_A, CHILD_V, L+1) :- includedProject(G, A, V, L), dependency(G, A, V, CHILD_G, CHILD_A,
       ↪ CHILD_V), not excludeProject(G, A, V, CHILD_G,CHILD_A), not dependencyCycle(CHILD_G, CHILD_A, CHILD_V,_,_,_).
17 minLevelOfDependency(GROUPID, ARTIFACTID, VERSION, LEVEL) :- includedProject(GROUPID, ARTIFACTID, VERSION, LEVEL), #
       ↪ min{LEVELX : includedProject(GROUPID, ARTIFACTID, _, LEVELX)} = LEVEL.
18
19 % Constraints
20 dependencyCycle(G, A, V) :- dependency(G, A, V, G, A,V).
21 dependencyCycle(G, A, V) :- transitiveDependency(G, A, V, G, A, V).
22
23 duplicatedDependency(GROUPID, ARTIFACTID, VERSION1, LEVEL1, VERSION2, LEVEL2) :- includedProject(GROUPID, ARTIFACTID,
       ↪ VERSION1, LEVEL1), includedProject(GROUPID, ARTIFACTID, VERSION2, LEVEL2), dependency(PARENT1_GROUPID,
       ↪ PARENT1_ARTIFACTID, PARENT1_V, GROUPID, ARTIFACTID, VERSION1), dependency(PARENT2_GROUPID, PARENT2_ARTIFACTID,
       ↪  PARENT2_V, GROUPID, ARTIFACTID, VERSION2), (PARENT1_GROUPID, PARENT1_ARTIFACTID, PARENT1_V) != (
       ↪ PARENT2_GROUPID, PARENT2_ARTIFACTID, PARENT2_V).
24
25 shadowedProject(GROUPID, ARTIFACTID, VERSION, LEVEL) :- minLevelOfDependency(GROUPID, ARTIFACTID, VERSIONMIN,
       ↪ LEVELMIN), includedProject(GROUPID, ARTIFACTID, VERSION, LEVEL), LEVEL>LEVELMIN, VERSION!=VERSIONMIN.
26
27 jumpCall(FROM_GROUPID, FROM_ARTIFACTID, FROM_VERSION, FROM_CLASS, FROM_METHOD_NAME, FROM_NUM_PARAM, FROM_RET_VAL,
       ↪ TO_GROUPID, TO_ARTIFACTID, TO_VERSION, TO_CLASS, TO_METHOD_NAME, TO_NUM_PARAM, TO_RET_VAL) :- call(
       ↪ FROM_GROUPID, FROM_ARTIFACTID, FROM_VERSION, FROM_CLASS, FROM_METHOD_NAME, FROM_NUM_PARAM, FROM_RET_VAL,
       ↪ TO_GROUPID, TO_ARTIFACTID, TO_VERSION, TO_CLASS, TO_METHOD_NAME, TO_NUM_PARAM, TO_RET_VAL), rootProject(
       ↪ FROM_GROUPID, FROM_ARTIFACTID, FROM_VERSION), project(TO_GROUPID, TO_ARTIFACTID, TO_VERSION), not rootProject(
       ↪ TO_GROUPID, TO_ARTIFACTID, TO_VERSION), not dependency(FROM_GROUPID, FROM_ARTIFACTID, FROM_VERSION, TO_GROUPID
       ↪ , TO_ARTIFACTID, TO_VERSION).
28
29 unusedProject(GROUPID, ARTIFACTID, VERSION, LEVEL) :- includedProject(GROUPID, ARTIFACTID, VERSION, LEVEL), not call(_
       ↪ ,_,_,_,_,_,_,GROUPID, ARTIFACTID, VERSION,_,_,_,_).
```

configurations. These rules represent the dependency smells listed in Table 1. In the context of our model for dependencies, we formulate them as hard constraints[8] to express that, regardless how and in which context our model is used (see Section 5 for pointers of future work), it is impossible to obtain instances of our model (*i.e.*, dependency configurations) that violate any of the given constraints depicted in Listing 2.

Line 20 and 21 depict the detection of *dependency cycles*. Line 20 states that if a dependency from one project to itself exists it is a cycle. Moreover, when there is a transitive dependency from a project to itself, it is also considered a dependency cycle.

[8] Note that, technically, each of the listed constraints has its own constraint formulation in addition to the shown facts. For example, there is an additional `:- dependencyCycle(G, A, V)` that disallows any model that would produce a dependency cycle fact. To avoid cluttering the paper, these constraints are omitted.

The rule in line 23 identifies projects that are *declared as dependency multiple times* in the dependency graph. This can happen due to the resolution of transitive dependencies, for example. The rule fires if there are two included projects that have the same groupId and artifactId. Moreover, the two included projects need to stem from two different parent projects to avoid false duplication reports (*e.g.*, without this constraint any included project would be a duplicated of itself). To retain the given information, the respective versions and level of the dependencies are also stored in this fact (VERSION1, VERSION2, and LEVEL1, LEVEL2).

Next, we detect projects that are shadowed by other versions of the project through the hierarchy in the dependency graph. Line 25 shows the rule to detect *shadowed projects*. A shadowed project is a project that would be included in the project but is declared on a lower level as the highest declaration of this project (LEVEL>LEVELMIN) and has a different version (VERSION!=VERSIONMIN). If both dependency declarations declare the identical project as dependency (*i.e.*, G, A, and

Table 2
Descriptive statistics and detection results of the studied projects. The abbreviations stand for: solving time (ST), declared dependencies (DD) number of facts that are inputted into the model (#InpFacts), the time to generate these facts (GT), and the corresponding smells Dependency Cycles (DC), Double Included Dependency (DID), Shadowed Dependency (SD), Jump Calls (JC), and Unused Dependency (UD).

Project	Repository			Clingo				Smells				
	LOC	#Classes	#Commits	ST	#DD	#InpFacts	GT	DC	DID	SD	JC	UD
dgc-lib	5807	59	230	7.7 s	9	426,382	90 s	0	18	8	293	10
jackson-databind	115,465	464	9125	0.6 s	2	29,565	6 s	0	0	0	0	0
jest-common	8535	158	1289	5.2 s	9	296,028	34 s	0	11	0	0	2
keycloak-admin-cli	6815	57	20,325	1.7 s	5	82,971	19 s	0	2	0	87	3

V are equal), no functionality is shadowed and hence, we do not count these as shadowed projects.

Combining the information from the dependency graph and the call graph, we can detect direct calls to projects that are not declared as direct dependencies of our project but are available via transitive dependency relations. We call such calls *jump calls* because they jump over at least one project dependency level. Line 27 shows the respective rule. This rule states that for any call in the root project that targets another project, we detect a jump call if the target project is not a declared dependency.

Similarly, we can detect *unused projects* that are included. Line 29 shows the detection rule. A project is marked as unused if it is included in the list of dependencies but no call to any of its methods is found. Note that a call from any project that is included to a project avoids marking the target project as unused.

We can leverage these rules in two ways, namely *validating* an existing dependency configuration and *generating* valid dependency configurations. For validating an existing dependency configuration, we input the facts of the project and its dependencies into DValidator. DValidator uses Clingo to solve the given ASP problem, *i.e.*, to detect the selected dependency smells. If Clingo returns `UNSAT`, then the model violates at least one of the given constraints. Reading the output of the solver, we can see which constraint was violated and initiate the repair of the configuration. For generating valid dependency configurations, we provide DValidator with the facts from additional versions of projects. For example, if we aim at generating a valid dependency configuration for the example given in Fig. 1, we additionally provide the encoded facts of other versions of the dependencies (for example version 2.4.0 of `ExternalDependency`).[9] Clingo then produces a set of possible dependency configurations that do not violate any of the given constraints.

4. Preliminary evaluation

In this section, we provide a preliminary evaluation of our proposed model to show its applicability. In particular, we show how our model can be used to *validate* existing dependency configurations of four open source Maven projects. We leave the evaluation of the generation of dependency configurations for future work. In the following, we first describe the setup of the evaluation and the used projects. Next, we present the obtained results.

4.1. Evaluation setup

In this preliminary investigation, we selected four Maven projects from different vendors, having different size, complexity, and purpose. The first four columns of Table 2 list the studied projects and report preliminary statistics about each studied project.

As described above, our model requires a dependency graph and a call graph of a project. We used a modified version of the tool by Gousious[10] to generate the call graph. In particular, we used the static call graph generator and modified it, that it not only extracts the call graph of a single project, but recursively creates a call graph including dependencies. Using the static call graph of Gousious et al., we share the following limitations: (1) our approach also misses reflective calls and (2) calls to abstract methods might end at the abstract class. Moreover, we used the tool of Macho et al. (2018) to extract and create the dependency graph. Starting from the root project, we transform the two graphs into the corresponding Clingo facts as described in Section 3. We repeat the extraction for each project that is declared as a dependency, recursively until no more dependencies are found. After this step, we obtain a fact base for each of the four projects that contains all the Clingo facts of the projects that the investigated project (transitively) depends on.

We then evaluated the precision of our approach by manually analyzing the smells that DValidator reported. In particular, the first two authors manually verified each smell separately and discussed unclear smell reports together. We report the precision in the following section. Please not that we did not analyze the recall of our tool because currently there is no benchmark data set to compare with and establishing a ground truth is not feasible in this context.

4.2. Results

Besides the descriptive statistics, Table 2 also shows the results of our preliminary study. Columns 5-8 (third block, “Clingo”) show the time to generate the model, the time that Clingo needed to execute all rules in a model, the number of dependencies that the root project declared, and the corresponding total number of facts that were encoded during the transformation process. Furthermore, columns 9-13 (last two blocks, “Smells”) report the number of instances per dependency smell and project that Clingo found based on our model.

The number of facts varies between the projects. The *dgc-lib* project generated the highest number of facts directly followed by *jest-common*. These two also have the highest number of declared dependencies, declaring nine direct dependencies each. Processing the large number of facts and executing the rules of a model only took 7.7 s at most which is still within a reasonable amount of time.

Regarding the smell detection, we see that dependency cycle (DC) was not detected in any of the four projects. However, three projects had duplicated dependencies (DID) defined throughout their dependency graph. Only the `jackson- databind` project did not have any duplicated dependencies. This may be a result of the low number of declared dependencies (2) in this project. Inspecting the reported duplicates, we observed that they are all correct. Although not all of them might necessarily be prone to cause an error, our model still identifies and reports them to give future generative approaches all available information to output high quality dependency configurations, which might include avoiding duplicates as much as possible.

[9] The selection of which dependency versions to provide is subject to future work. We envision various strategies that for example select versions close to the selected version numbers or the next minor/major version.

[10] https://github.com/gousiosg/java-callgraph

Shadowed dependencies (SD) were only found in the `dgc-lib` project. Analyzing them, we found that they mostly concern declarations of selected Spring framework components and the well-known logging framework slf4j.

In addition to the purely dependency-based smells, we also report the results for the dependency smells that leverage the call graph, *i.e.*, jump calls (JC) and unused dependencies (UD). The projects *dgc-lib* and *keycloak-admin-cli* contained jump calls. One of the jump calls in the *dgc-lib* project[11] concerns a call to a `getMessage()` method of a `JsonProcessingException` which is provided by the `jackson-core` project. This project is included transitively via the `jackson-databind` project. Directly declaring the `jackson-core` project as dependency from the root project would resolve this issue. Overall, we observed that in 98 (26%) cases the tool identified a jump call although a corresponding library is directly declared. However, all 98 cases refer to the declaration of the `bouncycastle` libraries. Our tool wrongly classifies these calls as jump calls, because the call graph wrongly assigns the call to the transitive dependency declaration. We also investigated unused dependencies, *i.e.*, dependencies that are included but no calls are made to them. We analyzed the reported smells and found that all identified 15 unused libraries were actual smells.

The observed shortcomings can be attributed to limitations of the tool to generate the call graph. For example, the *dgc-lib* project uses Lombok[12] which is a library to reduce the amount of boilerplate code. However, most of the usage of this library concerns adding annotations to the project's classes. The call graph tool currently skips annotations as they are no calls and hence, we miss the relationships from the investigated project to the library. In this case, the Lombok project is marked as unused because there are no method calls to this library. Similarly, we currently do not cover any calls that are unresolvable by the call graph tool. For example, this concerns calls to methods in abstract classes or interfaces, or calls that are preformed using reflection.

5. Future research directions

The results of our preliminary evaluation show the potential of our approach to detect dependency smells. While DValidator did not detect any dependency cycle in the four projects, it detected duplicated dependencies, shadowed dependencies, jump calls, and unused dependencies. The time to detect did not exceed 7.7 s and the time to generate facts did not exceed 90 s. Most of the detected dependency smells were confirmed by a manual analysis. Based on these promising results, we first describe possible technical improvements and then present several envisioned research directions that are enabled by our approach.

Technical Improvements. First, we plan to integrate our approach into IDE's, such as IntelliJ IDEA. This will help to solve the problem of generating the dependency graph and the call graph from scratch each time a project needs to be analyzed, because we can obtain the performed changes by the IDE and directly update the model facts without generating the whole model from scratch. This will save time in the data generation process.

As reported in Section 4, many false positives are due to the limitations of the static call graph generator. In future work, we plan to enhance our graph generation tool to better handle inheritance, reflection, and annotations.

Furthermore, we also reported in Section 4 that our tools currently only extract project level dependencies and method calls. This leads to false positives when detecting unused dependencies. In the next steps, we plan to improve our tools to also extract dependencies due to the usage of annotations and data types.

Improving the Search Space. Finding optimal solutions of dependency problems is an NP-hard problem (Abate et al., 2020). One reason for the complexity of this problem is the large search space in which the solution is located. Depending on the size of the investigated project and the number of (transitive) dependencies, the number of generated facts can grow large and hence, the performance of our approach can suffer. While this is usually no problem when validating existing dependency configurations, the problem can occur when our model is used to generate dependency configurations. We plan to mitigate this issue by additional experiments to streamline our approach with heuristics that help to narrow the search space.

Moreover, our model currently treats each constraint for a valid dependency configuration as a hard constraint, which disallows to violate any of these constraints. However, this can prevent the model from finding a valid dependency configuration. In some cases, the developers would accept violating selected dependency constraints as a trade-off for finding a solution. We plan to conduct experiments in that direction and survey developers to learn which constraints can be lowered to soft constraints.

Dependency Conflict Detection and Repair. Similar to the recent trend of auto repair of broken builds (Macho et al., 2018; Hassan and Wang, 2018), the first envisioned research direction is the detection (Wang et al., 2018, 2019) and automated repair of dependency conflicts. Concerning conflict detection, our model can be extended by adding special constraints to forbid (transitive) calls of the root project into two different versions of a project (*i.e.*, a dependency conflict). In a situation in which such two (transitive) calls exist, the model would, depending on its configuration, either report that this is an unsatisfiable model (*i.e.*, no fix within the given constraints could be found), or report the conflicting calls and dependencies.

Extending this idea further, our model can also generate solution candidates for such conflicting dependencies. It can be used to generate dependency configurations that do not violate the given rules and that do not violate the extended conflict detection rules of the preceding paragraph. In such a situation, a valid dependency configuration can be found and incorporated by the developers, avoiding dependency conflicts.

Updating Versions of Dependencies. Developers usually want to keep their dependencies up-to-date (Cox et al., 2015). Moreover, if a critical vulnerability is found in one of the used dependencies, developers must update a subset of their used dependencies to newer, or in some cases, older versions of a dependency. This is often challenging because changing dependency versions can have a huge impact on the build success of the project (Kerzazi et al., 2014; Seo et al., 2014; Rausch et al., 2017). The most prominent tool that incorporates such a mechanism is Dependabot.[13] However, Dependabot (Alfadel et al., 2021; He et al., 2022; Cogo and Hassan, 2022) cannot check whether its suggestions semantically break the project. Neither it can suggest other dependencies that would suit better. By extending our proposed model, such critical dependency updates or downgrades can be performed whilst checking the validity of the new dependency configuration. The provided rules of this extension can then indicate possible problems in the new configuration and, as depicted above, suggest changes to other parts of the dependency configuration to obtain a valid and updated dependency configuration.

[11] https://github.com/eu-digital-green-certificates/dgc-lib/blob/046831ac5555df11f2459ef2ccdbc9be1a36a37e/src/main/java/eu/europa/ec/dgc/gateway/connector/DgcGatewayValidationRuleDownloadConnector.java#L172

[12] https://projectlombok.org/

[13] https://github.com/dependabot

6. Conclusions

In this paper, we propose DValidator, a novel approach to model dependencies. It extends state-of-the-art methods by considering dependencies not only on a project level but also on a method call level. With this improvement, we can show that DValidator can detect well-known dependency smells. Moreover, our approach enables future work to automatically repair dependency smells and conflicts.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data and code is available under the Zenodo link.

Acknowledgments

The research reported in this paper has been partly funded by BMK, BMAW, and the State of Upper Austria in the frame of the SCCH competence center INTEGRATE (FFG 892418) part of the FFG COMET Competence Centers for Excellent Technologies Programme.

References

Abate, P., Di Cosmo, R., Gousios, G., Zacchiroli, S., 2020. Dependency solving is still hard, but we are getting better at it. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 547–551.

Abate, P., Di Cosmo, R., Treinen, R., Zacchiroli, S., 2012. Dependency solving: A separate concern in component evolution management. J. Syst. Softw. 85 (10), 2228–2240.

Alfadel, M., Costa, D.E., Shihab, E., Mkhallalati, M., 2021. On the use of dependabot security pull requests. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories. MSR, IEEE, pp. 254–265.

Bavota, G., Canfora, G., Di Penta, M., Oliveto, R., Panichella, S., 2013. The evolution of project inter-dependencies in a software ecosystem: The case of Apache. In: 2013 IEEE International Conference on Software Maintenance. IEEE, pp. 280–289.

Bavota, G., Canfora, G., Di Penta, M., Oliveto, R., Panichella, S., 2015. How the apache community upgrades dependencies: An evolutionary study. Empir. Softw. Eng. 20 (5), 1275–1317.

Cao, Y., Chen, L., Ma, W., Li, Y., Zhou, Y., Wang, L., 2022. Towards better dependency management: A first look at dependency smells in Python projects. IEEE Trans. Softw. Eng..

Capilla, R., Gallina, B., Cetina, C., Favaro, J., 2019a. Opportunities for software reuse in an uncertain world: From past to emerging trends. J. Softw.: Evol. Process 31 (8), e2217.

Capilla, R., Gallina, B., Cetina Englada, C., 2019b. The new era of software reuse. J. Softw.: Evol. Process 31 (8), e2221.

Chen, X., Abdalkareem, R., Mujahid, S., Shihab, E., Xia, X., 2021. Helping or not helping? Why and how trivial packages impact the npm ecosystem. Empir. Softw. Eng. 26 (2), 1–24.

Claes, M., Decan, A., Mens, T., 2018. Inter-component dependency issues in software ecosystems. In: Software Technology: 10 Years of Innovation in IEEE Computer. John Wiley & Sons.

Cogo, F.R., Hassan, A.E., 2022. Understanding the customization of dependency bots: The case of dependabot. IEEE Softw. 39 (5), 44–49.

Cox, J., Bouwers, E., Van Eekelen, M., Visser, J., 2015. Measuring dependency freshness in software systems. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Vol. 2. IEEE, pp. 109–118.

Decan, A., Mens, T., Claes, M., 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 2–12.

Decan, A., Mens, T., Claes, M., Grosjean, P., 2016. When GitHub meets CRAN: An analysis of inter-repository package dependency problems. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering. Vol. 1. SANER, IEEE, pp. 493–504.

Di Cosmo, R., Zacchiroli, S., Trezentos, P., 2008. Package upgrades in FOSS distributions: Details and challenges. In: Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades. pp. 1–5.

Fan, G., Wang, C., Wu, R., Xiao, X., Shi, Q., Zhang, C., 2020. Escaping dependency hell: Finding build dependency errors with the unified dependency graph. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 463–474.

Frakes, W.B., Kang, K., 2005. Software reuse research: Status and future. IEEE Trans. Softw. Eng. 31 (7), 529–536.

Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., 2014. Clingo=ASP+ control: Preliminary report. arXiv preprint arXiv:1405.3694.

Gebser, M., Kaminski, R., Schaub, T., 2011a. Aspcud: A linux package configuration tool based on answer set programming. arXiv preprint arXiv:1109.0113.

Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M., 2011b. Potassco: The Potsdam answer set solving collection. Ai Commun. 24 (2), 107–124.

Hassan, F., Wang, X., 2018. Hirebuild: An automatic approach to history-driven repair of build scripts. In: Proceedings of the 40th International Conference on Software Engineering. pp. 1078–1089.

He, R., He, H., Zhang, Y., Zhou, M., 2022. Automating dependency updates in practice: An exploratory study on GitHub dependabot. arXiv preprint arXiv:2206.07230.

Huang, K., Chen, B., Shi, B., Wang, Y., Xu, C., Peng, X., 2020. Interactive, effort-aware library version harmonization. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 518–529.

Humble, J., Farley, D., 2010. Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Pearson Education.

Ignatiev, A., Janota, M., Marques-Silva, J., 2014. Towards efficient optimization in package management systems. In: Proceedings of the 36th International Conference on Software Engineering. pp. 745–755.

Jafari, A.J., Costa, D.E., Abdalkareem, R., Shihab, E., Tsantalis, N., 2021. Dependency smells in Javascript projects. IEEE Trans. Softw. Eng. 48 (10), 3790–3807.

Kerzazi, N., Khomh, F., Adams, B., 2014. Why do automated builds break? an empirical study. In: 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, pp. 41–50.

Kikas, R., Gousios, G., Dumas, M., Pfahl, D., 2017. Structure and evolution of package dependency networks. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories. MSR, IEEE, pp. 102–112.

Kula, R.G., German, D.M., Ouni, A., Ishio, T., Inoue, K., 2018. Do developers update their library dependencies? Empir. Softw. Eng. 23 (1), 384–417.

Larios Vargas, E., Aniche, M., Treude, C., Bruntink, M., Gousios, G., 2020. Selecting third-party libraries: The practitioners' perspective. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 245–256.

Macho, C., McIntosh, S., Pinzger, M., 2018. Automatically repairing dependency-related build breakage. In: Proc. of the International Conference on Software Analysis, Evolution, and Reengineering. SANER, pp. 106–117.

Mäkitalo, N., Taivalsaari, A., Kiviluoto, A., Mikkonen, T., Capilla, R., 2020. On opportunistic software reuse. Computing 102 (11), 2385–2408.

Pinckney, D., Cassano, F., Guha, A., Bell, J., Culpo, M., Gamblin, T., 2023. Flexible and optimal dependency management via max-smt. In: Proceedings of the 2023 International Conference on Software Engineering. Ser. ICSE.

Rausch, T., Hummer, W., Leitner, P., Schulte, S., 2017. An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories. MSR, IEEE, pp. 345–355.

Seo, H., Sadowski, C., Elbaum, S., Aftandilian, E., Bowdidge, R., 2014. Programmers' build errors: A case study (at Google). In: Proceedings of the 36th International Conference on Software Engineering. pp. 724–734.

Shahin, M., Babar, M.A., Zhu, L., 2017. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. IEEE Access 5, 3909–3943.

Soto-Valero, C., Durieux, T., Baudry, B., 2021a. A longitudinal analysis of bloated Java dependencies. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1021–1031.

Soto-Valero, C., Harrand, N., Monperrus, M., Baudry, B., 2021b. A comprehensive study of bloated dependencies in the Maven ecosystem. Empir. Softw. Eng. 26 (3), 1–44.

Tanabe, Y., Aotani, T., Masuhara, H., 2018. A context-oriented programming approach to dependency hell. In: Proceedings of the 10th International Workshop on Context-Oriented Programming: Advanced Modularity for Run-Time Composition. pp. 8–14.

Tourwé, T., Mens, T., 2003. Identifying refactoring opportunities using logic meta programming. In: Seventh European Conference OnSoftware Maintenance and Reengineering, 2003. Proceedings. IEEE, pp. 91–100.

Treinen, R., Zacchiroli, S., 2009. Common upgradeability description format (CUDF) 2.0. Mancoosi Project (FP7) 3.

Valiev, M., Vasilescu, B., Herbsleb, J., 2018. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 644–655.

Wang, Y., Wen, M., Liu, Z., Wu, R., Wang, R., Yang, B., Yu, H., Zhu, Z., Cheung, S.-C., 2018. Do the dependency conflicts in my project matter? In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 319–330.
Wang, Y., Wen, M., Wu, R., Liu, Z., Tan, S.H., Zhu, Z., Yu, H., Cheung, S.-C., 2019. Could i have a stack trace to examine the dependency conflict issue? In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 572–583.
Wang, Y., Wu, R., Wang, C., Wen, M., Liu, Y., Cheung, S.-C., Yu, H., Xu, C., Zhu, Z.-l., 2021. Will dependency conflicts affect my program's semantics. IEEE Trans. Softw. Eng..

Christian Macho is a PostDoc assistant (tenure track) in the Software Engineering Group (SERG) at the University of Klagenfurt. He received his M.Sc. from the Technical University Vienna in March 2015 and his Ph.D. from the University of Klagenfurt in 2019 both with distinction. His research interests include software evolution, mining software repositories, program analysis, build systems, continuous integration, automated repair, and empirical studies in software engineering.

Fabian Oraze is a student assistant in the Software Engineering Group (SERG) at the University of Klagenfurt. He received his B.Sc. from the University of Klagenfurt in 2022. His research interests include build systems, continuous integration, and empirical studies in software engineering.

Martin Pinzger is a full professor at the University of Klagenfurt, Austria where he is heading the Software Engineering Research Group. His research interests are in software evolution, mining software repositories, program analysis, software visualization, and automating software engineering tasks. He is a member of ACM and a senior member of IEEE.