DOI: 10.15514/ISPRAS-2026-38(5)-8

On the Applicability of SOFA/SODA Antipattern Detection Approach to Multi-Module Solutions via a Pluggable Pre-Build Linter

A.S. Timonin, ORCID: 0009-0004-0730-3714 <ansetimonin@edu.hse.ru>
S.A. Shershakov, ORCID: 0000-0001-8173-5970 <sshershakov@hse.ru>
HSE University, 20, Myasnitskaya st., Moscow, 101000, Russia.

Abstract. This paper studies the applicability of the SOFA/SODA formalism, originally proposed for the detection of antipatterns in service-oriented systems, to the problem of structural analysis of multi-module software projects. We formulate antipattern detection as the evaluation of declarative rules over a static component dependency graph and propose a pluggable architecture in which any project that can expose its component graph through a small adapter interface becomes a valid input for the analyzer. The SOFA layer is implemented as a fixed set of graph metrics and the SODA layer as a set of declarative rule cards. A prototype command-line tool is implemented in Go with external rule plugins; two adapters cover the SwiftPM and Gradle ecosystems. The evaluation on two reference projects with labelled violations gives $F_1 = 1.00$ on the Swift project (13 violations, no false positives), zero false positives on the Kotlin control project, and 80% of the rules reusable between the two ecosystems. Our findings suggest that SOFA/SODA can be carried over from service-oriented systems to structural linting of multi-module projects in general, as long as a suitable project adapter is available.

Keywords: multi-module projects; structural analysis; software antipatterns; SOFA; SODA; static analysis; dependency graph.

For citation: Timonin A.S., Shershakov S.A. On the Applicability of SOFA/SODA Antipattern Detection Approach to Multi-Module Solutions via a Pluggable Pre-Build Linter. Trudy ISP RAN/Proc. ISP RAS, 2026, vol. 38, issue 5, pp. 119-132. DOI: 10.15514/ISPRAS-2026-38(5)-8.

Acknowledgments. The article was prepared within the framework of the HSE University Basic Research Program.

Применимость подхода «SOFA/SODA» для обнаружения проблем проектирования многомодульных решений на основе подключаемого предподготовленного анализатора исходного кода

А.С. Тимонин, ORCID: 0009-0004-0730-3714 <ansetimonin@edu.hse.ru>
С.А. Шершаков, ORCID: 0000-0001-8173-5970 <sshershakov@hse.ru>
Национальный исследовательский университет «Высшая школа экономики», Россия, 101000, Москва, ул. Мясницкая, 20.

Аннотация. В работе исследуется применимость формализма SOFA/SODA, изначально предложенного для обнаружения проблем проектирования в сервис-ориентированных системах, к задаче структурного анализа многомодульных программных проектов. Обнаружение проблем проектирования формулируется как вычисление декларативных правил над статическим графом компонентных зависимостей; предложена подключаемая архитектура, в которой любой проект, способный предоставить свой граф компонентов через небольшой интерфейс-адаптер, становится допустимым входом для анализатора. Слой SOFA реализован как фиксированный набор графовых метрик, а слой SODA – как набор декларативных карточек правил. Прототип консольной утилиты реализован на языке Go с внешними компонентами расширения правил; два адаптера покрывают инфраструктуру инструментов SwiftPM и Gradle. Оценка на двух эталонных проектах с размеченными нарушениями даёт F1=1.00 на Swift-проекте (13 нарушений, без ложных срабатываний), ноль ложных срабатываний на контрольном Kotlin-проекте и 80% правил, переиспользуемых между двумя инфраструктурами инструментов. Полученные результаты показывают, что SOFA/SODA можно переносить из сервис-ориентированных систем на структурный анализ многомодульных проектов в целом при условии наличия подходящего адаптера проекта.

Ключевые слова: многомодульные проекты; структурный анализ; проблемы проектирования программ; SOFA; SODA; статический анализ; граф зависимостей.

Для цитирования: Тимонин А.С., Шершаков С.А. Применимость подхода «SOFA/SODA» для обнаружения проблем проектирования многомодульных решений на основе подключаемого предподготовленного анализатора исходного кода. Труды ИСП РАН, 2026, том 38, вып. 5, стр. 119–132 (на английском языке). DOI: 10.15514/ISPRAS-2026-38(5)-8.

Благодарности. Исследование выполнено в рамках Программы фундаментальных исследований НИУ «Высшая школа экономики».

1. Introduction

The SOFA/SODA (Service Oriented Framework for Antipatterns / Service Oriented Detection for Antipatterns) approach by Moha et al. [1] and Palma et al. [2] provides a formal way to describe and detect antipatterns in service-oriented systems. Each antipattern is expressed as a card whose predicates are computed on top of a fixed set of metrics. The approach was designed for SOA and its practical validation was carried out on web service systems.

Modern software is often organized not as a set of services but as a set of source-level components: packages, modules or build targets that declare dependencies on each other. Such projects are subject to a class of defects that are structural rather than behavioural: dependency cycles, layer violations, forbidden cross-module references, uncontrolled transitive coupling. These defects are not detected by the compiler or by the package manager, and existing static analyzers are usually tied to a single language or build system.

The research question of this paper is therefore the following: *to what extent is the SOFA/SODA formalism applicable to the detection of structural antipatterns in multi-module projects, and can it be made independent of a particular ecosystem*? We approach this question by defining a uniform component graph model, mapping SOFA metrics and SODA rules onto it, and implementing a

pluggable command-line analyzer in which different project types are connected through a small adapter interface. Two adapters are provided as a proof of concept; adding a new ecosystem requires implementing the adapter only and does not affect the rule layer.

The paper makes the following contributions. First, we show how SOFA metrics and SODA rule cards, originally defined over behavioural service graphs, can be rewritten over a static component graph, and we describe a catalogue of rule cards that this rewriting enables. Second, we describe an analyzer in which project ecosystems and rule families are two independent extension points: a new ecosystem is added through a small adapter, and a new rule family is added through an external plugin without recompiling the core. Third, we report the results of evaluating the prototype on two reference projects in different ecosystems, which let us compare how much of the rule layer can be reused without change.

The rest of the paper is organized as follows. Section 2 formulates the problem. Section 3 adapts SOFA/SODA to a static component graph. Section 4 describes the analyzer. Section 5 reports the evaluation. Section 6 discusses related work and Section 7 concludes.

2. Problem Statement

We consider a multi-module project as a directed graph $G = (V, E)$, where V is the set of components declared by the project and $E \subseteq V \times V$ is the set of declared dependencies between them. For such a graph we write $P(G)$ for the set of properties that the SOFA layer computes on G: each property is a Boolean predicate or a numerical value read off the edges of G and its metric values. The pair (G, P(G)) is the object that the rule layer sees. A structural antipattern is a predicate on this pair that is forbidden by the team's architectural policy. The detection problem is, given $\big(G, P(G)\big)$ and a set of rule cards, to list all violations together with the evidence on which they are based.

Several conditions must be met for such a detection procedure to be practical. The component graph should be reconstructed from build manifests alone, so that the analysis can be performed before the project is compiled. The set of rules should bring detection value that is not already provided by the compiler, the package manager or file-level linters. The rule layer should operate only on the abstract graph model, so that a rule can be reused in different ecosystems without rewriting it; the share of rules that survive an ecosystem change without modification is therefore an important characteristic of the approach. Reports should be explainable, in the sense that every violation carries the evidence that triggered it, and idempotent, in the sense that two runs on the same repository state produce the same report. Finally, the analyzer should fit into a developer workflow, which constrains its run time and memory consumption. These conditions guide the design choices in Sections 3 and 4 and the metrics collected in the evaluation in Section 5.

3. Adapting SOFA/SODA to a Component Graph

3.1 Component Graph Model

The analyzer represents a project as $G = (V, E, adj, radj)$, where adj and $radj$ are precomputed forward and reverse adjacency indices that turn metric and rule evaluation into local lookups. Each vertex carries a set of tags (for example, a layer tag or a platform tag). Tags can be assigned manually or by a path convention, which makes the model usable on projects that do not declare layers explicitly.

The graph is the only object that the rule layer ever sees. All ecosystem-specific information is hidden inside the project adapter. This separation is what makes the formalism portable.

3.2 SOFA Layer: Graph Metrics

The SOFA layer collects structural properties of G. The prototype implements five metrics that are sufficient for the rule families we target: fan-in, fan-out, strongly connected components, the

transitive closure of adj, and per-component depth in the configured layer order. All metrics are language-independent and operate exclusively on G.

3.3 SODA Layer: Rule Cards

The SODA layer expresses each rule as a card that consists of an identifier, an intent borrowed from the original antipattern catalogue, a predicate over the SOFA metrics and an evidence template. The prototype defines cards for eight rule families.

Two notions should be kept apart here. The Knot, Multi Service and the other names from the SOFA/SODA catalogue are antipatterns in the original sense: named structural defects with an intent. The cards listed below (No-cycle, Multi-module, Layered violation and the rest) are not the antipatterns themselves, they only describe these antipatterns in a form that the analyzer can evaluate on G. One antipattern can be described by several cards, and one card can catch only one of its variants.

The simplest card, No-cycle, requires that no strongly connected component has size greater than one; it describes the antipattern The Knot. Closely related is Layered violation, which forbids edges from a lower to a higher layer in the configured layer order. Forbidden dependency is more flexible: it bans individual edges that match a deny pattern expressed in terms of tags or component names, and in practice it is the card that projects configure most often. Transitive limit looks not at single edges but at the whole transitive closure of a component, which is useful when the team wants to cap how much of the project a single module can reach. Feature isolation applies to components marked as features and allows them to depend only on shared components and on each other through declared interfaces. Multi-module, an analogue of Multi Service, fires when a component exceeds configured fan-in and fan-out limits at the same time. Finally, two smaller cards round off the set: Orphan component checks that every component is reachable from at least one declared entry point, and Tag policy allows the team to require or forbid edges between configured pairs of tags.

Three of these cards (No-cycle, Multi-module, Layered violation) are direct counterparts of SOA antipatterns from the original SODA catalogue [1-2]. The rest grew out of concerns that do not have a clear equivalent on a behavioural service graph, such as transitive reach or feature boundaries.

Fig 1 shows the component graph of the Swift reference project used in the evaluation. Components are grouped by layer: UI (App, FeatureHome, FeatureAuth), Domain, Data (Data, NetworkKit) and Shared (Analytics, DesignSystem). Solid arrows represent declared dependencies. The dashed red arrow from FeatureHome to FeatureAuth is a violation of the Forbidden dependency rule, which bans edges between feature modules. The FeatureHome and FeatureAuth nodes are highlighted in red to mark them as participants in the violation. This is the kind of edge that the compiler does not flag but that the forbid card detects automatically.

Trimmed excerpts of three representative rule plugins (layer order, forbidden dependency and max fan-out) are given in Appendix A (Listings A.1–A.3). Each plugin extends BaseRule and receives the graph together with pre-computed SOFA metrics.

For every reported violation, the analyzer emits an evidence record that contains the involved vertices and edges, the values of the metrics that triggered the rule and a textual explanation derived from the intent of the card. Evidence is part of the rule definition, not of the report writer, which makes the explanations stable across runs.

3.4 Declarative Configuration

Policies are written in a YAML-based domain-specific language (DSL) [3] that supports profile inheritance, per-component overrides and a path convention for automatic tagging. The configuration is split in two files: an infrastructure file and an architectural policy file. The schema is validated against a JSON Schema before any rule is executed, which detects typos and missing fields early. Listing 1 shows a fragment of the rules file used in the Swift reference project.

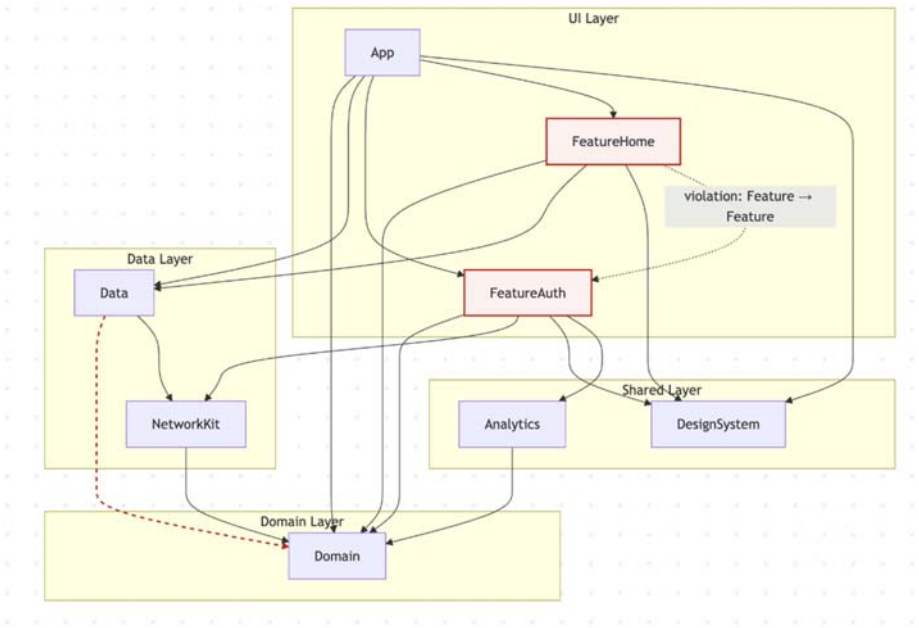

Fig 1. Component graph of the Swift reference project.
The dashed red edge marks a detected violation (Feature → Feature).

4. Pluggable Analyzer

The analyzer is a command-line tool with a four-stage pipeline, shown in Fig 2. At the ingestion stage, a platform-specific adapter reads the build manifests (Package.swift or settings.gradle.kts together with per-module build.gradle.kts files) and produces the component graph G. The model building stage computes SOFA metrics on G: fan-in, fan-out, strongly connected components via Tarjan's algorithm [4], transitive closure via BFS, and layer depth via Kahn's topological sort [5]. These algorithms are implemented once inside the Go core, as part of the SOFA layer, and are not duplicated in the rule plugins. The rule evaluation stage passes the graph and the metrics (see Section 4.2 and Listing A.4) to SODA rule plugins over a JSON protocol; each plugin returns a list of violations with evidence. The reporting stage collects the violations and writes them out as JSON and Markdown. Reports support baseline and suppressions, so that an existing project can be onboarded step by step without showing the developer old violations.

4.1 Project Adapters and Manifest Parsing

A small interface, ProjectAdapter, hides the ecosystem from the core. An adapter receives a project root and returns the abstract graph G together with vertex tags. Two adapters are provided: SwiftPM and Gradle. Both rely only on build manifests and do not invoke the compiler. The SOFA and SODA layers remain untouched when an adapter is added or replaced (Listing 1).

The two ecosystems are chosen because they are standard in their respective platforms. SwiftPM is the official tool for Swift package description, shipped together with the Swift compiler since version 3.0 [6]. Gradle is the de-facto build system for Android and Kotlin projects and supports multi-module builds through settings.gradle [7]. In both systems the component structure is described declaratively in manifest files, so the analyzer can read the graph before the project is

compiled. Multi-module architecture itself is a common practice in mobile development: large teams, for example Bumble [8], split their applications into many components to enable incremental builds, parallel work of several teams and code reuse between products. The way modules are composed also affects the application binary: in SwiftPM, the linkage type of a target (static or dynamic) influences the resulting binary size and the startup time [6], which users of mobile applications directly notice. For these reasons, rules that constrain the shape of the module graph are useful in this setting, and SwiftPM and Gradle cover the two platforms on which our reference projects are built.

```
profiles:
  - name: base
    rules:
      - id: no-cycles
        type: no_cycles
        severity: error
      - id: layer-order
        type: layers
        severity: error
        spec:
          layers:
            order: [UI, Domain, Data, Shared]
            allow_same_layer: false
      - id: no-feature-to-feature
        type: forbid
        severity: error
        spec:
          forbid:
            from: { name: "Feature*" }
            to:   { name: "Feature*" }
      - id: max-fan-out-5
        type: max_fan_out
        severity: warning
        spec:
          fan_out: { max: 5 }
  - name: strict
    extends: base
    rules:
      - id: max-fan-out-5
        severity: error
        spec:
          fan_out: { max: 4 }
use: base
```

Listing 1. Fragment of the rules file (.depslint.rules.yaml).

For each platform the parsing of manifests has to be done carefully, otherwise some edges or dependencies stay invisible to the rest of the analyzer and the report becomes unreliable. The two ecosystems we support use very different manifest formats, and this is visible already in small examples.

In Swift the project structure is defined in a single Package.swift file. Listing B.1 in Appendix B shows a trimmed version of the one used in the reference project. Each .target declares its dependencies on other targets as an array.

The Swift adapter extracts the graph by calling swift package dump-package, a standard SwiftPM command that returns the resolved manifest as JSON. The adapter reads target names and their dependency lists from this JSON and builds the edges of G.

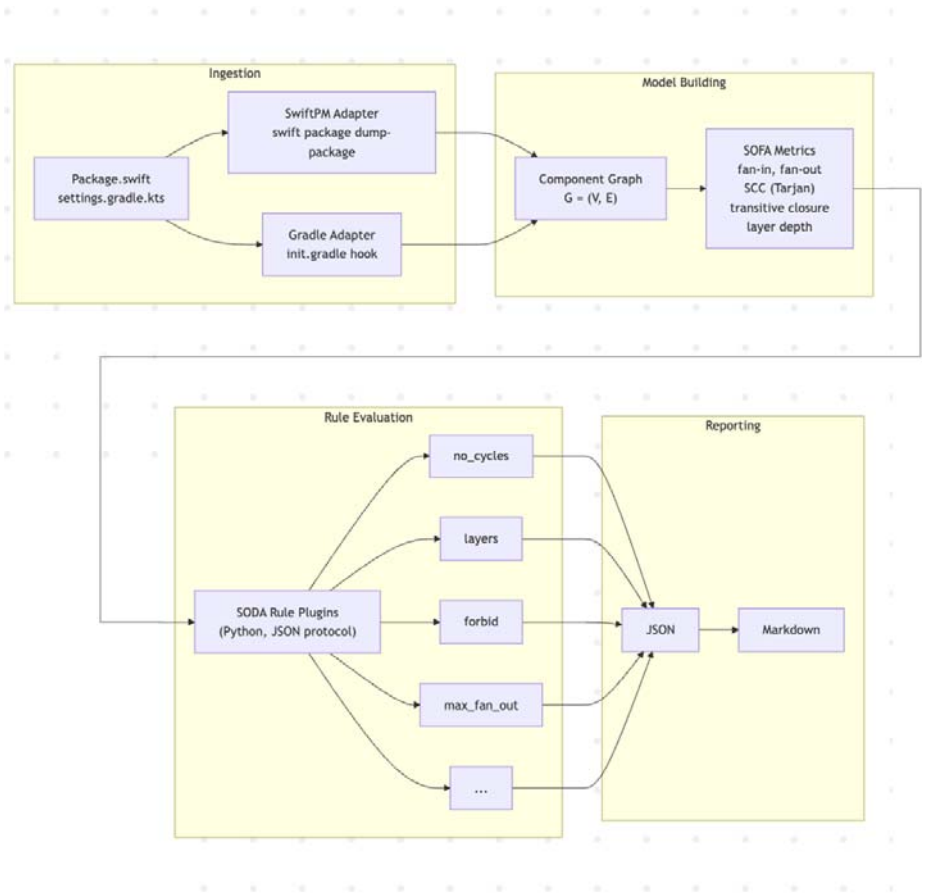

Fig 2. Four-stage pipeline of the analyzer.

In Kotlin/Gradle the project structure is split across many files. A settings.gradle.kts at the root lists all modules, and each module has its own build.gradle.kts with a dependencies block. Listings B.2 and B.3 in Appendix B show the two files for the reference project.

There is no single Gradle command that dumps the full inter-module graph the way swift package dump-package does for Swift. Having such a dump matters for the analyzer: it needs a single, predictable source of truth about the graph that does not depend on reading several manifest files by hand and does not run the full build. On the Swift side that source already exists in the form of a standard command; on the Gradle side the adapter has to produce it. The Kotlin adapter works around this by injecting a short init.gradle script [9] that hooks into projectsEvaluated and writes the result to a temporary JSON file. The trimmed script is shown in Listing B.4 in Appendix B.

Both approaches return the same graph type to the core, but they require separate work on the ecosystem side, and this is the kind of work that any future platform will have to repeat. For this reason, adapters are not exposed as external plugins and stay an internal part of the CLI, unlike the rule plugins written in Python: the correctness of the graph matters too much to delegate it to a loosely coupled extension.

4.2 Plugin Architecture for Rules

Rule evaluation is delegated to external rule plugins through a JSON protocol. The core sends to a plugin the relevant subgraph and the metric values, and the plugin returns a list of violations with evidence. The Go core implements the graph and metrics; rule plugins are written in Python. New rule families can be added without recompiling the core, which keeps the rule layer separate from the infrastructure. Listing A.4 in Appendix A shows the body of the reference no_cycles plugin. The plugin receives a pre-computed list of cycles from the SOFA layer and turns each cycle into a violation with an evidence chain.

4.3 Reporting

The Markdown report is intended for human readers in code review. The JSON report is the main form on which reproducibility is checked: running the analyzer twice on the same repository state produces a byte-identical JSON document.

5. Evaluation

Two reference projects of small scale were prepared, one in Swift (SwiftPM, $|E| = 22$) and one in Kotlin (Gradle, $|E| = 23$). Violations were labelled by hand for every rule family that the prototype supports. The Swift project is the main detection benchmark and contains 13 seeded violations covering all eight cards. The Kotlin project is a control project without seeded violations and is used to measure false positives on a clean repository. Measurements were repeated on a fixed test bench.

5.1 Graph Reconstruction

Both adapters reconstruct the expected graphs ($|E| = 22$ for the Swift project, $|E| = 23$ for the Kotlin project) using only build manifests, without invoking the compiler. This confirms that manifests are sufficient for the rule families considered in this work and that pre-build analysis is possible for the two ecosystems.

5.2 Detection Quality

On the Swift project the analyzer detected all 13 labelled violations and produced no false positives, which gives $precision = recall = F_1 = 1.00$. On the Kotlin control project no violations were reported, that is, zero false positives. The detected violations belong to families that are not covered by the compiler or by file-level linters: cycles, layer violations, transitive limits and tag policies. The $F_1 = 1.00$ score is high, but it was obtained on a small project that was built for this evaluation and where the layers were clearly defined in advance. It shows that the tool works on a clean example, not that it will give the same numbers on every project.

5.3 Rule Portability

Across the eight rule families, 80% of the rule definitions are reused between Swift and Kotlin without modification when parametric adaptation, such as different layer names and tag values, is taken into account. The remaining rules refer to ecosystem-bound concepts and were left platform-specific. The result indicates that the abstract graph model carries enough information to express most structural rules in an ecosystem-independent way.

5.4 Explainability and Reproducibility

Every reported violation includes an evidence record with the involved edges and metric values, derived from the intent template of the corresponding card. To check reproducibility, the analyzer was launched more than ten times on different commits of the two reference projects. When a commit did not change the component structure of the project, the report stayed exactly the same as

on the previous run, byte by byte. When the structure did change, only the violations affected by the change appeared or disappeared in the report. This behaviour means that the tool can be safely used in code review without producing noisy diffs between runs.

5.5 Performance

On the two reference projects the analyzer finishes in roughly 0.7-2.5 seconds end to end, depending mostly on how long the adapter needs to read the manifests. Resident memory stays under about 100 MB in all runs. These numbers were measured on a single developer laptop and should be read as an order of magnitude rather than a benchmark: the projects are small, and we expect the run time to grow with the size of the component graph. What is important for the rest of the paper is that the tool is fast enough to be invoked on every commit without the developer waiting for it.

5.6 Threats to Validity

The evaluation has several limitations that we want to name explicitly. The two reference projects are small and were prepared by the author: it is a benchmark that the tool was built for, not an independent sample, and the perfect F_1 score on the Swift project should be read in this light rather than as a general quality claim. We also did not compare the analyzer against a baseline on the same projects, because most of the rules we check (transitive limits, tag policies, feature isolation) are not supported by the tools we would compare with. The 80% portability number depends on how we counted parametric adaptations; a stricter count would give a lower value. Finally, the run-time numbers come from a single machine and only describe the behaviour on small inputs. In addition, the applicability of the approach to large industrial projects with hundreds of components has not been investigated in this paper; scaling the evaluation to such projects is the first item of future work (Section 7).

6. Related Work

The work that this paper builds on is the SOFA/SODA family of papers [1-2], in which antipatterns of service-oriented systems are formalised as cards over a fixed set of metrics. Our contribution is to transfer this formalism from a behavioural service graph to a static component graph and to make the formalism independent of a particular ecosystem.

Several existing tools were considered. ArchUnit [10] expresses architectural rules as unit tests over JVM bytecode, so it requires a compiled project and stays within a single ecosystem. Dependency Cruiser [11] is the closest in spirit: it performs pre-build dependency analysis with declarative rules, but it is bound to the JavaScript/TypeScript ecosystem and its rules are not defined over graph-level metrics such as transitive reach. SwiftLint [12] and Detekt [13] check style and code quality at the level of individual files and do not build an inter-component dependency graph at all. Each of these tools therefore covers only a part of the problem: pre-build operation, declarative configuration or architectural checks. None of them builds a full component graph of the project and then lets the user define rules over graph-level metrics and apply the same rule set across different ecosystems. The analyzer described here sits in this gap. A direct quantitative comparison with these tools on our reference projects was not performed, because most of the rule families we check (transitive limits, tag policies, feature isolation) are not supported by them; this limitation is discussed in Section 5.6.

7. Conclusion and Future Work

Looking back at the experiment, the more interesting observation is not that SOFA/SODA can be reused on a static component graph – this was the working hypothesis – but how much work this transfer actually takes. Once the rule layer is forced to see nothing but a graph and a few metrics, most of the per-ecosystem detail ends up in the adapter, and the two adapters we wrote for SwiftPM and Gradle turned out to be thin. In this sense the project was less about inventing new detection

techniques and more about finding the right place to cut between the portable and the ecosystem-specific parts of a linter.

The harder question is where the approach breaks. The analyzer is only as useful as the graph the adapter hands to it: if the adapter misses an edge, the rules simply will not see a violation, and if it adds a wrong edge, they will fire on something that is not there. This is not a bug that can be fixed on the rule side. Any future adapter – for a different build system or for a language we have not looked at yet – will need its own careful decisions about which kinds of dependencies to surface and which to hide, even when the rule cards above it stay the same.

There are three directions in which we would like to push the work. The first one is simple but necessary: running the analyzer on larger, real-world projects to see how the rules and the performance behave outside of the prepared benchmark. The second is writing new adapters and improving the existing ones. On the Apple side, many real projects mix Swift modules with Objective-C code wrapped in XCFrameworks: the current SwiftPM adapter sees only targets declared in Package.swift and does not pick up binary dependencies distributed as XCFrameworks or linked through bridging headers, so the graph it builds may be incomplete. Similarly, on the Android side, the Gradle adapter currently handles Kotlin modules but does not treat pure Java modules differently even though many older projects still have Java-only Gradle modules that live next to Kotlin ones during a migration. Supporting these mixed setups would make the tool useful on a wider set of real codebases.

The third direction is more open. So far, the metric layer consists only of graph properties, but a component typically carries a lot of non-code information – assets, configuration, localisation – that a team can have opinions about. Lifting some of this into the metric layer would let us write rules that today cannot be expressed on a pure dependency graph, and it would be a good test of whether the extension points of the analyzer hold up when they are pushed in a direction we did not originally plan for.

References

[1]. Moha N. et al., “Specification and detection of SOA antipatterns,” in Proc. ICSOC, 2012.
[2]. Palma F., Moha N., Tremblay G., Guéhéneuc Y.-G. Specification and detection of SOA antipatterns in web services. In Proc. ECSA, 2014.
[3]. Fowler M., Domain-Specific Languages. Addison-Wesley, 2010.
[4]. Tarjan R. E. Depth-first search and linear graph algorithms. SIAM J. Comput., vol. 1, no. 2, 1972.
[5]. Kahn A. B. Topological sorting of large networks. Commun. ACM, vol. 5, no. 11, 1962.
[6]. Swift Package Manager. Organize, manage and edit Swift packages. Available at: https://docs.swift.org/swiftpm/documentation/packagemanagerdocs/, accessed 20.07.2006.
[7]. Gradle User Guide. The Java Library Plugin. Available at: https://docs.gradle.org/current/userguide/java_library_plugin.html, accessed 20.07.2006.
[8]. Bumble Engineering: Scaling modular architecture. Available at: https://medium.com/bumble-tech, accessed 20.07.2006.
[9]. Gradle User Guide. Initialization Scripts and Init Plugins. Available at: https://docs.gradle.org/current/userguide/init_scripts.html, accessed 20.07.2006.
[10]. ArchUnit. Unit Test Your Java Architecture. Available at: https://www.archunit.org/, accessed 20.07.2006.
[11]. Dependency Cruiser. Available at: https://github.com/sverweij/dependency-cruiser, accessed 20.07.2006.
[12]. SwiftLint. Available at: https://github.com/realm/SwiftLint, accessed 20.07.2006.
[13]. Detekt. A static code analysis for Kotlin. Available: https://detekt.dev/, accessed 20.07.2006.

Информация об авторах / Information about authors

Антон Сергеевич ТИМОНИН – выпускник факультета компьютерных наук НИУ ВШЭ. Сфера научных интересов: разработка и профилирование инструментов построения программного обеспечения.

Anton Sergeevich TIMONIN graduated Computer Science faculty of the Higher School of Economics University. His research interest include design, development and profiling of the software tools and instruments.

Сергей Андреевич ШЕРШАКОВ – кандидат компьютерных наук, доцент, академический руководитель магистерской программы «Системная и программная инженерия» факультета компьютерных наук НИУ ВШЭ. Сфера научных интересов: извлечение и анализ процессов, встраиваемые системы, объектно-ориентированное программирование, Software Process Mining, сервис-ориентированная архитектура (СОА).

Sergey Andreevich SHERSHAKOV – Cand. Sci. (Comp. Sci.), Assoc. Prof., Academic Supervisor of the Master Programme "System and Software Engineering" on Faculty of Computer Science, HSE University. Research interests: process mining, data mining, embedded systems, object-oriented programming, Software Process Mining, SOA.

Appendix A. Rule Plugin Examples

This appendix contains trimmed excerpts of the Python rule plugins referenced in Sections 3.3 and 4.2.

Listing A.1. Layer order plugin (excerpt).

```
class Layers(BaseRule):
  def evaluate(self, rule, graph, metrics):
    order = rule["spec"]["layers"]["order"]
    allow_same = rule["spec"]["layers"]
                     .get("allow_same_layer", False)
    idx = {name: i for i, name in enumerate(order)}
    violations = []
    for edge in graph.edges:
      fr = graph.components[edge["from"]].tag("layer")
      to = graph.components[edge["to"]].tag("layer")
      fi, ti = idx.get(fr), idx.get(to)
      if fi is None or ti is None:
        continue
      if fi == ti and not allow_same:
        violations.append(Violation(...))
      elif fi > ti:
        violations.append(Violation(...))
    return violations
```

Listing A.2. Forbidden dependency plugin.

```
class Forbid(BaseRule):
  def evaluate(self, rule, graph, metrics):
    spec = rule["spec"]["forbid"]
    violations = []
    for edge in graph.edges:
      fr = graph.components[edge["from"]]
      to = graph.components[edge["to"]]
      if matches(fr, spec["from"]) \
         and matches(to, spec["to"]):
        violations.append(Violation(
          rule_id=rule["id"],
          source=edge["from"],
```



```
          evidence=[EvidenceHop(
            from_id=edge["from"],
            to_id=edge["to"])],
          message=f"{edge['from']} -> {edge['to']}"
                  f" is forbidden"))
    return violations
```

Listing A.3. Max fan-out plugin.

```
class MaxFanOut(BaseRule):
  def evaluate(self, rule, graph, metrics):
    max_val = rule["spec"]["fan_out"]["max"]
    violations = []
    for cid, comp in graph.components.items():
      fo = metrics.fan_out.get(cid, 0)
      if fo > max_val:
        violations.append(Violation(
          rule_id=rule["id"],
          source=cid,
          message=f"{cid}: fan-out {fo} > {max_val}"))
    return violations
```

Listing A.4. No-cycle plugin.

```
class NoCycles(BaseRule):
  def evaluate(self, rule, graph, metrics):
    violations = []
    for cycle in metrics.cycles:
      if len(cycle) < 2:
        continue
      evidence = []
      for i, cid in enumerate(cycle):
        nxt = cycle[(i + 1) % len(cycle)]
        evidence.append(
          EvidenceHop(from_id=cid, to_id=nxt))
      chain = " -> ".join(cycle + [cycle[0]])
      violations.append(Violation(
        rule_id=rule["id"],
        source=cycle[0],
        evidence=evidence,
        message=f"cycle: {chain}"))
    return violations
```

Appendix B. Build Manifests and Gradle Init Script

This appendix contains the manifest fragments and the Gradle init script referenced in Section 4.1.

Listing B.1. Fragment of Package.swift (four targets shown).

```
let package = Package(
  name: "DemoApp",
  targets: [
    .target(name: "App",
      dependencies: ["Domain", "FeatureAuth",
                     "FeatureHome"]),
    .target(name: "FeatureHome",
      dependencies: ["Domain", "FeatureAuth"]),
```

```
    .target(name: "Domain", dependencies: []),
    .target(name: "FeatureAuth",
      dependencies: ["Domain", "NetworkKit"]),
    // ...
  ])
```

Listing B.2. settings.gradle.kts from the reference project.

```
rootProject.name = "demo-app"
include(":app")
include(":domain")
include(":data")
include(":network")
include(":analytics")
include(":feature-auth")
include(":feature-home")
include(":design-system")
```

Listing B.3. build.gradle.kts of the feature-home module.

```
plugins { kotlin("jvm") }
dependencies {
  implementation(project(":domain"))
  implementation(project(":data"))
  implementation(project(":design-system"))
  implementation(project(":feature-auth"))
}
```

Listing B.4. Gradle init script injected by the Kotlin adapter.

```
gradle.projectsEvaluated {
  def model = [:]
  allprojects { proj ->
    def deps = []
    ["api", "implementation",
     "testImplementation"].each { cfgName ->
      def cfg = proj.configurations
                    .findByName(cfgName)
      cfg?.dependencies
         ?.withType(ProjectDependency)
         ?.each { dep ->
           deps << [config: cfgName,
                    path: dep.dependencyProject.path]
         }
    }
    model[proj.path] = [
      name: proj.name, dependencies: deps]
  }
  new File(rootProject.projectDir,
           ".linter-model.json")
    .text = groovy.json.JsonOutput.toJson(model)
}
```