Fast Static Analyses of Software Product Lines — An Example With More Than 42,000 Metrics

Sascha El-Sharkawy
University of Hildesheim,
Institute of Computer Science
Hildesheim, Germany
elscha@sse.uni-hildesheim.de

Adam Krafczyk
University of Hildesheim,
Institute of Computer Science
Hildesheim, Germany
adam@sse.uni-hildesheim.de

Klaus Schmid
University of Hildesheim,
Institute of Computer Science
Hildesheim, Germany
schmid@sse.uni-hildesheim.de

Abstract

Context: Software metrics, as one form of static analyses, is a commonly used approach in software engineering in order to understand the state of a software system, in particular to identify potential areas prone to defects. Family-based techniques extract variability information from code artifacts in Software Product Lines (SPLs) to perform static analysis for all available variants. Many different types of metrics with numerous variants have been defined in literature. When counting all metrics including such variants, easily thousands of metrics can be defined. Computing all of them for large product lines can be an extremely expensive process in terms of performance and resource consumption.

Objective: We address these performance and resource challenges while supporting customizable metric suites, which allow running both, single system and variability-aware code metrics.

Method: In this paper, we introduce a partial parsing approach used for the efficient measurement of more than 42,000 code metric variations. The approach covers variability information and restricts parsing to the relevant parts of the Abstract Syntax Tree (AST).

Conclusions: This partial parsing approach is designed to cover all relevant information to compute a broad variety of variability-aware code metrics on code artifacts containing annotation-based variability, e.g., realized with C-preprocessor statements. It allows for the flexible combination of single system and variability-aware metrics, which is not supported by existing tools. This is achieved by a novel representation of partially parsed product line code artifacts, which is tailored to the computation of the metrics. Our approach consumes considerably less resources, especially when computing many metric variants in parallel.

CCS Concepts

• General and reference → Metrics; • Software and its engineering → Software product lines; • Automated static analysis.

Keywords

Software Product Lines, SPL, Metrics, Implementation, Variability Models, Feature Models, Abstract Syntax Trees, AST

ACM Reference Format:
Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2020. Fast Static Analyses of Software Product Lines — An Example With More Than 42,000 Metrics. In Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS ’20), February 5–7, 2020, Magdeburg, Germany. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377024.3377031

1 Introduction

In software engineering, static analyses are commonly used in order to analyze a software system and to identify potential defects. A well-established form of static analyses are software metrics [12], which are used for the prediction of faults [33] or maintainability issues [34]. In Software Product Lines (SPLs), variability information is an important part, which is not covered by traditional software metrics. The SPL research community developed new variability-aware metrics to address this issue, which received increasing attention over the last decade [3, 10, 31]. In a previous study [10], we identified 147 variability-aware metrics to measure qualitative characteristics of variability models and code artifacts, which partly influence each other [2]. While traditional software metrics for single systems are well analyzed with respect to their ability to draw qualitative conclusions [33], there are only very few evaluations available regarding the application of variability-aware metrics for SPLs [10]. Further, there are no comparisons between well-established single system and variability-aware metrics available. The lack of available tools for measuring variability-aware metrics aggravates the situation.

In this paper, we present a concept for efficiently parsing code files of SPLs that stores sufficient information for the realization of single system metrics from traditional software engineering as well as variability-aware code metrics designed for the needs of SPLs. In addition, our concept allows the arbitrary combination of variability-aware code metrics with feature metrics, which was not investigated so far. Thus, the presented parsing concept provides the foundation for the realization and evaluation of new SPL metric suites like MetricHaven1. Here, we present the concepts behind the tool, which was presented in [9]. We pursue the following research questions:

RQ1 What are the requirements to support a flexible measurement of single system and variability-aware code metrics?

RQ2 How can existing variability-aware metrics for code and variability models be combined?

RQ3 What abstraction is required to support a scalable analysis of large-scale SPLs?

We implemented our concept in the publicly available tool MetricHaven [9], which provides practitioners and researchers with a foundation for the flexible definition and measurement
of code metrics for SPLs implemented in C. MetricHaven is also
designed as a highly configurable software product line and
provides re-implementations of traditional and variability-aware
code metrics from different research groups. Its design supports
the highly efficient measurement of more than 42,000 metric
combinations on large-scale product lines.

Overall, we make the following contributions:

- We present the concept of Reduced Abstract Syntax Trees
  (RASTs) that contain sufficient information for the definition
  of most traditional and variability-aware code metrics, while
  minimizing resource overhead.
- A concept that allows a flexible combination of variability-aware
  feature and code metrics.
- A discussion of the limitations of the presented approach.

2 Related Work

The research community developed a huge variety of variability-
aware metrics, designed for the needs of SPLs [3, 10, 31]. Below, we
discuss the related work on variability-aware metrics based on four
characteristics: tool support, applicability, flexibility, and scalability.

Tool support. In 2012, Montagud et al. [31] investigated
to which extend authors of variability-aware metrics provide
tool-support. Their study included metrics for all life cycles of SPLs
and, thus, was not limited to implementation. They conclude that
only 52% of 35 identified papers provide (partial) tool support for
the computation of metrics. We address this issue by providing
a concept together with a publicly available tooling for the flexible
realization of variability-aware code metrics. The presented
approach supports a broad variety of single system as well as
variability-aware code metrics of different research groups [9].

Applicability. An important aspect is the applicability of the
available metrics. We categorized implementation-related metrics
according to four categories [10]: Metrics for variability models (this
was included, because variability models are used to manipulate
all artifacts of SPLs), annotation-based code, composition-based code,
and the combination of code and variability model metrics. We
discovered that available concepts and their realizations are limited
either to one of the aforementioned categories or are further
restricted to certain file types. For instance, S.P.L.O.T. [30] and
DyMmer [4] provide various metrics for variability models saved in
the S.P.L.O.T. file format (XML files). FEATUREVISU [1] was used for
the measurement of code artifacts from composition-based SPLs,
using different feature-oriented implementation techniques. In
the context of annotation-based code, many authors implemented
their metrics to operate directly on the XML output of srcml2
[17, 24]. Thus, their measurement is limited to a specific set of
implementation languages and require a re-implementation for the
measurement of SPLs using a different annotation technique. Passos
et al. [32] do not specify an implementation for the measurement of
scattering degree metrics, but their appendix3 contains a set of
Bash scripts explicitly designed for the analysis of Linux. This
approach requires a re-implementation of their metrics for the
measuring of other SPLs, even if they use a similar implementation
technique. We present a measurement concept for the analysis of

Footnotes:
1 https://www.srcml.org/
2 https://github.com/Mukelabai/featurescattering18/
3 https://www.mukelabai.com/
Table 1: Supported measures (code elements × variability dimensions; ✓ = represented in RAST, ● = supported only via String operations, × = no support).

| Code Element                  | No Variability | Variation Points (VPs) | Variation Point Expressions |
|-------------------------------|----------------|------------------------|-----------------------------|
| Only Variability              | ✓              | No. of VPs [13]; Cyclomatic Complexity on VPs [11, 27]; Nesting Depth of VPs [17, 18, 24, 38] | SDVP [17, 18, 24], SDVP [17, 39], TD [17, 18, 24] |
| Function Definitions          | ✓              | Fan-In / Fan-Out [16] | Conditional Fan-In / Fan-Out Degree Centrality [13] |
| Function Calls                |                |                        |                             |
| Control Structures            | ✓              | McCabe [28], Nesting Depth [6] |                             |
| Statements                    | ✓              | Statement Count [19] |                             |
| Line Numbers                  | ✓              | Lines of Code [19] | LoF [7, 11, 14, 17, 24, 38], PLoF [11, 17] |
| Comments                      | ✓              | (Non-)Commented LoC [19], Ratio of Comments per LoC [12] |                             |
| Operators & Operators         | ×              | Halstead [15] |                             |
| Variable Usage                | ×              | Liveness of Variables [6] |                             |

3.1 SPL Analysis Strategies

Thüm et al. [37] surveyed analysis approaches for SPLs and identified three categories of analysis strategies:

- **Product-based analysis techniques** operate on instantiated products of the SPL. This strategy allows the usage of standard analysis techniques from traditional software engineering, since the variability information is resolved [37]. For instance, professional metric tool suites like the Axivion Bauhaus Suite⁴, Teamscale from CQSE⁵, and SonarQube⁶ may be utilized for the measurement of instantiated code artifacts. However, for a high coverage of the original SPL, this strategy requires redundant computations as the products share code and, thus, is very time-consuming. Further, the analysis of all supported product variants of the SPL is often not feasible in practice as the number of products is typically exponential in the number of features.

- **Family-based analysis techniques** operate on product line artifacts containing variability information and take advantage of a variability model to limit the analysis to valid configurations only. This strategy allows analysis of the code for all possible product configurations, without the need of generating any products. However, this strategy does not work with available tools developed for the analysis of single systems. Since family-based analysis techniques consider all product line artifacts as a whole, the size of the analysis problem can easily exceed physical boundaries such as the available memory [37].

- **Feature-based analysis techniques** analyze product line artifacts containing variability information, too. Contrary to family-based approaches, this strategy analyzes each feature in isolation and ignores all other features as well as the variability model. This reduces the potentially exponential number of analysis tasks. However, this kind of analyses cannot detect any problems caused by feature interactions [37].

Most of the surveyed variability-aware metrics operate on product line artifacts containing variability information and consider all features, but ignore the variability model [10]. Thus, they can be classified somewhere in between family-based and feature-based analysis approaches. We designed our analysis approach so that it can reproduce the current state-of-the-art in variability-aware metrics but may also incorporate information from the variability model.

3.2 AST Parsing Strategies for SPL Analyses

We observed two fundamentally different parsing strategies for family-based analysis approaches. Sincero et al. [35] focus on parsing only preprocessor blocks to extract variability information of product line artifacts. This approach takes advantage of the strong abstraction and allows the extraction of variability information in $O(n)$ with the number of variation points. According to [35], Undertaker⁷ requires about half an hour to parse all 25,844 source code files (*.c, *.h, *.S) of the Linux Kernel Version 2.6.33 with a quad core CPU and 8 GB RAM. While this strategy is very fast compared to more detailed data representations, the analysis capabilities of this approach are very limited. The authors designed this approach for the analysis of (un-)dead code with respect to the implemented variability [36]. This approach does not support any code analysis, since the parser does not parse any elements of the programming language.

Kästner et al. [20] use a more sophisticated parsing strategy consisting of a variability-aware lexer and a variability-aware parser, implemented as part of TypeChef⁸. The lexer annotates all tokens of the programming language with its presence conditions, i.e., the condition of the enclosing variation point used for the selection of the token. It also includes all header files and expands macros. The parser creates for each supported configuration of the parsed code an alternative subtree as part of the resulting variable AST. The authors use a SAT-solver during lexing and parsing to reason about code parts that belong together or may be skipped. The very detailed code representation in conjunction with annotated variability information allows a broad range of family-based analysis techniques, like variability-aware type checking, variable control-flow graphs, and variability-aware liveness analysis [26].

The creation of the very detailed variable AST requires much more effort than the previous approach. Parsing of the x86 architecture of the Linux Kernel version 2.6.33 with 7,665 C-files (*.c) are included through the variability-aware lexer) requires roughly 85 hours on dual/quad-core lap computers with 2 to 8 GB RAM (the authors do not precisely specify their measurement system) [20]. This parsing approach has an additional downside beside the massive time consumption. Through the macro expansion and the treatment of statements belonging to different configurations, the variable AST does not represent the developers view on the code anymore.

---

⁴https://www.axivion.com/en/products-60#produkte_bauhaussuite
⁵https://www.cqse.eu/en/products/teamscale/landing/
⁶https://www.sonarqube.org/
⁷https://vamos.informatik.uni-erlangen.de/trac/undertaker
⁸https://ckaestne.github.io/TypeChef/
We surveyed existing traditional and variability-aware code metrics in order to design a Reduced AST. On the one hand, our RAST contains more information than the approach by Sincero et al., which stores only information about the variation points used in code artifacts. On the other hand, our approach stores less information than the variable AST and, thus, does not facilitate the same code analyses as supported by the TypeChef infrastructure. However, our concept provides an efficient measurement of a large variety of traditional and variability-aware code metrics, which can not be done by any of the previously discussed parsing strategies.

4 Concept

Here, we present the concept of parsing Reduced Abstract Syntax Trees (RASTs). This was motivated by designing a tailored parsing approach which is able to extract the information needed for the desired static analyses. In our case, we planned a flexible definition of single system and variability-aware code metrics to allow comparisons of them. Based on our survey on variability-aware code metrics and an informal literature study on metrics from traditional software engineering, we came up with the following requirements for parsing RASTs (cf. Req1):

**Req1 Parsing of un-preprocessed code.** While established metric analysis tools from commercial vendors usually resolve preprocessor statements before conducting metrics, variability-aware metrics analyze the preprocessor statements. Thus, we require a common data representation for code annotations (in our case C-preprocessor statements) and for elements of the programming language (in our case AST elements of the C-language). This is a challenging task, since the used preprocessor is not part of the programming language and can be used at arbitrary positions inside a code file, independently of any syntax definitions.

**Req2 No syntactically correct AST needed.** An important aspect is to extent the resulting AST-structure needs to support only syntactically correct programs. Contrary to compilation tasks and type checking analyses, we do not need a syntactical correct AST for the computation of code metrics. However, the AST structure should be as close as possible to the actual code structure to simplify the definition of code metrics. Thus, it is still a challenging task to enhance a traditional AST structure with variability annotations, since these annotations may be inserted at arbitrary positions intertwined with AST elements of the programming language.

**Req3 Granularity of RAST.** For optimization as well as for practical reasons it is important to assess the required granularity of parsed elements. A very fine grained AST, containing representations for all syntax elements of the annotation and programming language, conceptually supports every code metric. On the other hand, this requires much more effort to develop a very comprehensive parsing approach and leads to higher resource consumption. Due to limited development resources, we designed a Reduced Abstract Syntax Tree (RAST), which is sufficient for measuring all planed metrics and may be easily extended to support further metrics, if desired. The granularity of the RAST is driven by the measured elements of surveyed metrics, which we present in Table 1.

4.1 Reduced Abstract Syntax Tree (RAST)

Based on our SLR on variability-aware code metrics [10] and an informal literature study on metrics for single systems, we designed a Reduced Abstract Syntax Tree (RAST) for the efficient measurement of the most relevant traditional and variability-aware code metrics. Our goal is the measurement of C-based SPL implementations. Thus, we limited the scope of our analysis to the measurement of metrics on a per-function basis. Figure 2 presents the main elements of our RAST:

- **SourceFiles** represent the RAST representation of code files.
- **CodeElement** is the super class of all RAST elements. It stores the line numbers to trace parsed elements back to their location in code files and facilitates LoC-metrics. Further, we store for each element two representations of the condition of surrounding variation points: The **condition** stores the condition of the innermost variation block, considering conditions of siblings for #elif/#else-blocks. For instance, we store the condition \( A \) of the while statement in Line 3 of the listing in Figure 3. This allows the computation of feature-based metrics on all parsed elements, e.g., Scattering Degree metrics. Second, **presenceCondition** provides an alternative as it stores the full presence condition for the inclusion of the element, also considering all surrounding variation points. For code elements that are not surrounded by any variation points, we set **condition** and **presenceCondition** to TRUE.

---

The concepts we propose here could also be applied well beyond C.
The SingleStatement is the most fine-grained element of the RAST. We do not provide RAST representations for expressions of statements, but we store these elements as unstructured text (UnparsedCode). For instance, in Line 6 we store a SingleStatement with the text "stmt;", not knowing whether this is a function call, a variable declaration, or anything else.

The UnparsedCode element facilitates the storage of preprocessor elements at arbitrary positions inside the RAST. This is required since preprocessor directives are not syntactical elements of the C programming language and may be used at arbitrary positions inside a code file, independently of any syntax definitions. The UnparsedCode element is the parent of all (parsed) CppBlocks and UnparsedCode expressions of SingleStatements.

We use CppBlock to store conditional blocks, i.e., variation points. This means, we store #if, #ifdef, #ifndef, #elif, and #else-blocks in separate instances, referring to all siblings of the same block structure. The type attribute is used to distinguish between the different preprocessor elements and to allow a differentiation during the computation of metrics, if required. CppBlock inherits from Unparsed, which is used for elements of SingleStatements, and inherits from CodeElementWithNesting, which is used as a container inside our RAST. The multiple inheritance allows a nesting of preprocessor directives at arbitrary positions inside the RAST.

We use BranchStatements similar to CppBlocks to store the if and else statements of the programming language. This class also stores the siblings of the same if/else-structure. Again, the type denotes which specific syntax element was used to allow a differentiation during the metrics computation, if necessary.

LoopStatements represent any loop of the programming language. Contrary to BranchStatements they do not have siblings. Again, we support different loop types.

Functions represent function definitions. The function’s signature is stored as UnparsedCode, while the function body is composed of previously described elements.

Reference elements are special as they neither represent syntax elements of the programming language nor of the annotation language. They are used in case that syntactical elements of the presented RAST, like loops or control structures, are split into multiple parts by C-preprocessor statements. The listing of Figure 3 shows an example in which the C-preprocessor is used for the conditional compilation of a loop statement, while the statements of the loop are always present. In this case, a LoopStatement with one Reference is stored inside a CppBlock. This can be seen on the right side of Figure 3. The actual statements are stored outside of the CppBlock. This way it is possible to simultaneously define metrics on the same parsed data structure, that consider the nested statements as variable as well as metrics that do not treat this statement as variable.

### 4.2 Application of RAST

The RAST is designed to preserve the code structure in order to facilitate the computation of variability-aware code metrics according to their original definitions, while reducing the performance overhead by omitting elements that are not required for the metrics. For this we neither resolve the variability as usually done by commercial metric tool suites nor do we duplicate parsed code elements of alternative variants as done by TypeChef [20], as this would lead to modified metric values. As a consequence, our RAST contains a 150% representation of the parsed code.

Parsing of C-code requires the ability to cope with undisciplined annotations [25, 29], which are conditional compilation directives that do not align with the underlying syntactic structure of the code. Our RAST provides two concepts to support these annotations as described above: In cases that elements of a statement are conditional, a CppBlock inside a SingleStatement may be used to store the conditional elements. Further, References may be used to represent conditional control structures.

Based on the RAST, new metrics may be implemented as a visitor, which may be parameterized to represent different variations of a metric family. For instance, MetricHaven uses one visitor to compute different variations of McCabe’s Cyclomatic Complexity measure [28], which counts the linear independent paths of the (variable) control graph. For the single system version, we completely ignore the variability of the code and add 1 to the number of visited control structures (while, for, if, case). According to [6] this counting approach is equivalent to the original definition of McCabe. However, this approach provides support for control structures of undisciplined annotations, since we do not need to compute a syntactically correct control graph. The left side of Figure 3 provides an example on how we compute the cyclomatic complexity of a conditional loop. By ignoring the annotations, we detect two linear independent paths of the resulting control graph. The variability-aware version of this metric considers only paths created by variation points [27], i.e., #ifdef-statements. Finally, we provide a superimposition of both variants by counting the number of control structures of the programming language and the annotation language. The right side of Figure 3 visualizes the resulting control graph, which contains three linear independent paths: The loop may be present but omitted completely at run-time ①, the loop may be executed ②, and the loop may be removed through conditional compilation but the statements are kept ③.

---

10 According to the visitor design pattern.
5 Realization

Our concept is implemented as a prototype for the analysis of C-based SPLs like the Linux Kernel. This is realized in several plug-ins for the KernelHaven infrastructure [22, 23]. This infrastructure supports three types of extractors to read information from the product line to be analyzed:

- Code extractors extract variability information from the source code implementing the software product line. For C source code, this typically involves parsing #ifdef-blocks of the C-preprocessor.
- Build model extractors extract variability information from the build process of the product line. This typically involves presence conditions that define in which configurations a given source code file is compiled into the product line.
- Variability model extractors extract the variability model of the software product line. This contains a list of all features and constraints between them.

In KernelHaven, the result from these extractors is represented in models that are agnostic to implementation details of specific product lines, while still being extensible to additional information. This allows the following analysis components, which access these models, to be implemented independently from implementation details of specific product lines. The following paragraphs will first introduce which extractors were used in our prototype implementation and then explain the analysis process that implements our approach.

The parsing of the product line source code is implemented in the srcMLExtractor\(^{11}\) plug-in for KernelHaven. It is based on the srcML\(^{2}\) tool which parses source code to an XML format [5]. The srcMLExtractor parses this XML and converts it into a model compatible with KernelHaven. The extension mechanism of the general code model in KernelHaven is used to model the Reduced Abstract Syntax Tree (RAST), as introduced in Section 4.1. The parsing process of the srcMLExtractor does not parse the full AST output of srcML, but only descends to a granularity that provides sufficient information to build the RAST. See Figure 2 for a simplified class diagram of the resulting RAST structure.

For the build and variability models, we use the KbuildMinerExtractor\(^{12}\) and KconfigReaderExtractor\(^{13}\). The former extracts variability information from the build process of the Linux Kernel, the latter reads the Kconfig variability model present in the Linux Kernel source tree. These plug-ins and their underlying tools were already used in previous analyses of the Linux Kernel, and there were no changes done to these plug-ins when implementing the approach presented in this paper.

The analysis process in KernelHaven typically consists of multiple analysis components that are combined to an analysis pipeline. The output of the previous component(s) is used as the input for the following component(s). The initial input for the first analysis component(s) are the models supplied by the three extractors (see above). This structure allows for simple re-use of analysis components when creating new analysis pipelines.

The calculation of metrics is implemented as such an analysis pipeline in the MetricHaven\(^{1}\) plug-in for KernelHaven. Figure 4 shows an overview of this pipeline structure. The coloring of the lines indicates the flow of the three models extracted from the product line, as described above. The actual metric computation happens in the rightmost component at the end of the pipeline. The input for this component are the three extracted models (the code model went through the FunctionFilter first) and the output of three preprocessing components.

- The FunctionFilter component splits the code model into individual functions, and removes any elements that are outside of functions (such as global variables). This component does not compute any values for metrics; it is only used for convenient data organization. The result is a stream of code functions.
- The FunctionCallMap component analyses calls between functions. Since the Reduced Abstract Syntax Tree (RAST) is not fully parsed (cf. Section 4.1), function calls inside statements are identified heuristically: If the unparsed code string of a statement contains a function name followed by an opening parenthesis, we consider that statement to contain a call to this function. For each identified function call, the calling function (caller), the called function (callee), and the presence condition and location of the statement containing the function call are stored. This information is for example used in the Fan-In-/Out and Degree Centrality metrics.
- The ScatteringDegree component calculates the Scattering Degree metric for all features of the variability model. The result is a map of all features and their values for the different scattering degree types (SDVP [7, 18, 24, 32] and SDFile [17, 39]).
- The FeatureSize component calculates the Feature Size metric for all features in the variability model. The result is a map of all features and the number of statements controlled by the feature.

The preprocessing components are executed before the final metric computation component, because they require a full overview of the complete code model. In contrast, the metric calculation component calculates the metric values on a per-function basis.

---

1\(^{1}\)https://github.com/KernelHaven/srcMLExtractor
2\(^{2}\)https://github.com/KernelHaven/KbuildMinerExtractor
3\(^{3}\)https://github.com/KernelHaven/KconfigReaderExtractor

Figure 4: Analysis pipeline structure used in MetricHaven.
The Linux Kernel version 4.15\textsuperscript{14} to evaluate the implementation of our concept. Based on the RAST presented in Figure 2, we were able to realize various single system and variability-aware code metrics\textsuperscript{[9]}. Through the combination of variability-aware code metrics with feature metrics we support 42,796 metric variations (as of Summer 2019), which can be measured in a single pass (cf. RQ3). While most metrics can be implemented in a straightforward manner, some implementations require adaptations for our RAST. For instance, the detection of function calls for the Fan-In/-Out metric can only be implemented heuristically, since it would require full parsing of the expression syntax (cf. Section 5).

The x86 architecture in the Linux Kernel has 20,356 C-source files that are evaluated by us. 106 ($\approx 0.5\%$) of these files, cannot be handled by the current implementation of the srcMLExtractor, which translates the XML output of srcML to KernelHaven’s code model. This is mostly related to very special corner cases of undisciplined C-preprocessor usage\textsuperscript{[29]}, i.e., conditional annotations that do not align with the syntactic structure of the code. srcML marks up the C-syntax independently of C-preprocessor directives. In conjunction with undisciplined C-preprocessor directives, this can lead to incorrect markups provided by srcML. Proper handling of those structures requires adaptations of the srcML parser. Some special cases are detected and fixed by our srcMLExtractor. However, this approach cannot repair all of these cases and requires significant development effort. Further, a minority of these corner cases cannot be mapped to our RAST at all as they violate the few structural assumptions of the RAST. An even more lenient RAST that can model these cases, however, would significantly complicate the definition and computation of metrics.

We ran two sets of metrics on the Linux Kernel, to evaluate scalability of our approach (cf. RQ3): First, a selected subset of metric combinations that we also used in practice for our own work with the Linux Kernel. We call this set atomic metrics. It contains all basic code metrics and all possible combinations of code metrics combined with a single feature metric. Code metrics combined with multiple feature metrics are not included. This results in a set of 648 metrics. Second, we allowed all metric combinations. However, the implementation of one metric family (approximation of Eigenvector Centrality) requires significantly more memory than the other implementations and is not optimized for the provided parallelization capabilities of MetricHaven. For this reason, we executed only the 148 metric variations of this metric, which were already executed as part of the atomic metrics, while we executed all variations of the remaining metrics. This results in a set of 29,976 metrics.

For the performance measurements, we ran our implementation in a virtual machine running Ubuntu 16.04 with 40 logical CPU cores of an Intel Xeon E5-2650v3 @ 2.3 GHz and 314 GiB RAM. For the analysis, we limited the JVM once to 50 GiB memory\textsuperscript{15} and once to 24 GiB. However, it must be noted that the extractors run in separate processes as they execute external tools and, thus, allocate additional memory. Accurate timings for specific phases are hard to measure since KernelHaven makes heavy use of parallelization. For example, the preprocessing components already start to run while code parsing is still running. However, the actual metric calculation component can only start when the complete code model has been passed through the preprocessing components. That means that we identify two distinct execution phases: code parsing and metric calculation. The preprocessing phase, which partially happens in parallel to the code parsing only takes a few seconds, which is insignificant compared to the total runtime. Thus, we do not supply measurements for this phase. The code parsing and metric calculation components can also be independently configured to use a specified amount of threads. We measured the runtime of these components for different numbers of configured threads. For the experiments, we used a range of 1 to 10 threads to cover a spectrum which is supported by most workstation computers.

Running the 648 atomic metrics on all 409,253 functions that we parse from the x86 Linux Kernel architecture produces 265,195,944 measures. In CSV format, this is about 1.2 GiB. The metric calculation step takes about 27.5 minutes (54 minutes with 25 GiB memory for the JVM) to run on a single thread and can be decreased to about 13.75 minutes (38.5 minutes) on 10 computation threads.

Running the large set of 29,976 metrics on all 409,253 functions that we parse from the x86 Linux Kernel architecture produces 12.2 billion measures. In CSV format, this is about 53 GiB. The metric calculation step takes from 11 hours and 26 minutes on one thread to 6 hours 15 minutes on 10 threads (6 hours 20 minutes total runtime). The parsing performance stays the same as described in the previous paragraph as it is independent from the number of computed metrics. The parallelization benefit of metric computations is slightly less pronounced compared to the atomic metrics. However, our prototype is not fully optimized with regard to parallelization.

Figure 5 visualizes the runtime of running the 648 atomic metrics on a different number of threads with 50 GiB memory. The number of threads for the parsing and the metric computation phase were both modified together on a range of 1 to 10. For each run, we present the parsing time (lower, blue bars), the metrics

\textsuperscript{14}https://mirrors.edge.kernel.org/pub/linux/kernel/v4.x/linux-4.15.tar.xz

\textsuperscript{15}via the command line switches: -Xmx50g -Xms50g
In future work, we plan to further extend this framework in terms of the range of supported metrics, improve its performance and apply it to study numerous properties of product line implementations. We are particularly interested in the prediction of defects based on product line metrics.

**Acknowledgments**

This work is partially supported by the ITEA3 project REVaMP², funded by the BMBF (German Ministry of Research and Education) under grant 01IS16042H. Any opinions expressed herein are solely by the authors and not of the BMBF.
