Received December 30, 2021, accepted March 4, 2022, date of publication March 17, 2022, date of current version March 24, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3160485

Advancing Static Code Analysis With Language-Agnostic Component Identification

MICAH SCHIEWE, JACOB CURTIS, VINCENT BUSHONG, AND TOMAS CERNY
Department of Computer Science, Baylor University, Waco, TX 76798, USA

Corresponding author: Tomas Cerny (tomas_cerny@baylor.edu)

This work was supported in part by the National Science Foundation under Grant 1854049 and in part by the Red Hat Research (https://research.redhat.com).

ABSTRACT Static code analysis of software systems has proven beneficial for a broad range of domains, including security assessments, coding practice, error detection, and others. However, as modern systems have grown in complexity and heterogeneity over the past few decades, advances in development frameworks have dominated. Rather than involving low-level language constructs, these frameworks typically focus on software components, including data entities, controllers, and endpoints. As a result, current code analysis approaches have become unsuitable for analyzing these modern systems due to their focus on low-level constructs in a single language. Thus, code analysis has become a far more complicated endeavor thanks to the plethora of languages, frameworks, and design approaches in modern software development. This paper presents a novel approach to solving the problem of being tied to a single language and its low-level constructs. The system's source code is transformed into an intermediate representation called a language-agnostic abstract-syntax tree. This system representation is then assessed by generalized component parsers that extract relevant high-level information, such as components, from low-level structures. The design of the approach is presented here in detail, along with its evaluation in a case study involving two large, heterogeneous, cloud-native system benchmarks (Java and C++ microservices). The study demonstrates a unified identification approach to determine system data entities and endpoints. Utilizing higher-level constructs, such as components, can advance the current practice of system analysis to better face broader problems introduced by modern system development practices.

INDEX TERMS Code analysis, distributed systems, enterprise architecture, microservices.

I. INTRODUCTION

Static code analysis has been used to automate many tasks [1] in recent decades, including code reviews, formal verification, code clone detection, error detection, security assessments, code quality, reverse engineering, and so forth. As modern software systems grow ever larger and more complex, such automated analysis has become more important than ever. While modern frameworks have made creating systems far faster and easier than in the past, they have also introduced many high-level components that specialize in specific tasks (such as controllers, data entities, services, endpoints, transfer objects, and access objects). Such components are common across platforms and frameworks, and misusing them can introduce architectural smells like cyclic service dependency, sloppy delegation, promiscuous controller, and more [2]. Leveraging knowledge of these high-level components would allow easy detection of these problems. However, traditional static code analysis does not recognize them, instead operating only with low-level programming constructs, typically in a single language. This puts them at a disadvantage when analyzing modern systems due to platform dependency and an inability to recognize high-level system constructs.

In this paper, we argue that code analysis researchers must pay more attention to the needs of current system development methodologies, such as platform-independent analysis and high-level system constructs, to properly analyze modern and often distributed systems. For example, consider current cloud-native platforms implementing microservice

The associate editor coordinating the review of this manuscript and approving it for publication was Jemal H. Abawajy.

architecture [3]. Such architecture promotes distributed, self-contained modules with decentralized control, which in turn promotes distributed development in multiple languages. To statically analyze such a system, we would need to coordinate multiple tools, each with its own API, design philosophy, and data format; this which would be quite challenging.

To address these challenges (platform independence and high-level system constructs), we propose a two-step approach. The first step parses the system's code into an intermediate representation, which we call a Language-Agnostic Abstract-Syntax Tree (LAAST). This system representation is then assessed by a set of generalized parsers to detect specific component types. We call these parsers Relative Static Structure Analyzers (ReSSA). ReSSA can utilize generic parsers across platforms; however, to better cope with platform differences, custom system-specific parsers can be developed to improve detection precision.

To assess our approach, we implemented it, then used it to create parsers to identify select components in two commonly-used microservice testbeds developed by outside researchers, DeathStarBench [4] and TrainTicket [5]. DeathStarBench is a C++ system developed at Cornell University, while TrainTicket is a testbed comprised of 41 Java-based microservices developed at Fudan University. Both testbeds are good representatives of current systems using components, and both provide vastly different design methodologies for the study. Our promising case study results indicate high precision and recall on the component identification task, indicating ReSSA's suitability to implement robust parsers for vastly different structures.

In summary, the proposed approach provides a unified interface to extract a broad range of component types that can scale across various platforms. It shows strong potential to transform static code analysis practices by operating with component types rather than low-level programming constructs. Using our approach, many of the hardest challenges confronting static analysis on decentralized systems–such as understanding software evolution, detecting architectural degradation, understanding the system-centralized view, automated reasoning, and more–can be addressed more easily. This can fuel new advances through tools operating on high-level system constructs in a platform-independent manner.

This paper is organized as follows: background and related works are given in Section II, followed by an analysis of the problem domain in Section III. Section IV details our approach with LAAST and ReSSA. An evaluation case study and performance analysis, along with a discussion on limitations and threats to validity, can be found in Section V. The paper concludes with a few thoughts on future extensions of the project in Section VI.

II. BACKGROUND AND RELATED WORK

Modern software development has evolved quite far from the days of mono-language tech stacks, growing to encompass many languages spread across many platforms. In modern microservice development alone, language heterogeneity has shown itself to be a primary concern [6]. Many other key concerns–such as a lack of a centralized view of a microservice system and inter-service dependencies–are made more complicated by this first concern. These are a few select but major industry challenges in the evolvability of these systems [6], and make the development of frameworks to perform analysis on multiple languages all the more important.

An example of a modern analysis tool capable of utilizing multiple languages is SLACC [7], a tool focused on detecting code clones across languages. While this is an impressive feat, it does so via dynamic analysis (running software to determine what it does), limiting its capacity for performance when analyzing large-scale projects. This is not the only setback dynamic analysis faces. For multi-language analysis specifically, dynamic analysis is restricted further thanks to complexities in determining how running programs interact across a language barrier [8]. Because of these issues, we will focus on static analysis instead.

Static analysis, as described by the International Standard for Systems and software engineering–Vocabulary [9], is "a process of evaluating a system or component based on its form, structure, content, or documentation." This has been implemented in a number of ways over the years, from analyzing executables (e.g., bytecode [10]) to analyzing source code itself. The former is of somewhat limited utility in our pursuit of extracting higher-level components from programs, given that the translation from source code to machine code often erases symbols and obfuscates code intent and purpose; thus, it is the latter that we focus on.

Source code analysis has traditionally relied on a construct called an Abstract Syntax Tree (AST), a structure that defines a program as a series of nodes that represent the program in a form that is easily parsed by another program [11]. Being traditionally tied to a single programming language, this format can be somewhat difficult to apply to multi-language projects. This is not an insurmountable wall, however. As seen in the work by Zügner *et al.* [12], there has been a success in analyzing ASTs generated from multiple languages to perform complex tasks (in their case, computing a method's name from its body). However, Zügner's work relied on machine learning to make sense of the varied ASTs, making it more difficult to apply to new problems without being a machine learning specialist.

However, machine learning is not the only way to analyze multiple languages at once. Common interfaces for languages can also be achieved through intermediate languages for compilers such as Common Intermediate Language (CIL) or LLVM intermediate representation (LLVM IR) [13]. While these tools provide a common abstraction for static code analysis, they are still high-level assembly languages and may have had optimizations applied during compilation. Because they are lower level than source code, it is entirely possible for information about high-level constructs to be lost. This makes intermediate languages unsuitable for high-level component identification.

Many researchers have turned to UML as another language-agnostic intermediate representation. UML has been programmatically generated from source code with tools like Ptidej [14], [15] and Pilfer [16], converted into other representations for validation [17], and used as a base for advanced analysis like determining design patterns [18]. Despite the advances in this field, however, UML is not a perfect fit for detecting conceptual components in modern static code analysis. Without extending UML, it does not support proper "blocks of code," which can be crucial to detecting high-level concepts. For example, consider identifying endpoint calls between microservices via static code analysis. If the project under analysis uses Axios [19] or some similar HTTP request technology, the endpoint calls will exist as statements within blocks of code, which are lost when translated to UML. While the tools inferring UML elements from source code, like Ptidej, must have access to this specific information, they do not seem to offer an interface to access it. This limits their utility for developing brand-new tools. Thus, UML and its current ecosystem do not offer the detail necessary to infer many high-level components in current systems.

These are not the only language-independent formats that exist. For example, by using LARA [20], [21], an aspect-oriented framework enabling source-to-source compilation, it is possible to stitch multiple language-specific ASTs together and apply a common interface to construct a "virtual AST" capable of being analyzed uniformly. While this offers a powerful representation that supports existing tools, it has numerous places for improvement. First, this limits the performance potential of the approach, as it requires a language-specific AST by definition. Given how large multi-language projects typically become, performance penalties of this sort must be taken seriously. Second, it increases complexity greatly because the language-specific AST must be handled at both the parsing and runtime analysis. This adds both opportunities for bugs and further performance penalties.

A similar approach to LARA is taken by Rakić *et al.* in their SSQSA static analysis framework [22]. Instead of attempting to translate an AST at runtime from one language to another, they instead inject additional information into the language-specific AST in the form of 'tags'–creating an "enriched Concrete Syntax Tree" (eCST). These tags convey information about the abstract function of a construct, such as indicating that both JavaScript's for-in loop and Java/C++'s do/while loop are looping constructs. These tags then enable metrics to be taken language-agnostically, with finer-grained analysis relying on the constructs from the original AST. This approach sees many benefits, such as relying on a constant API without needing too much language-specific analysis (as the tags are rather simply grafted in). However, it is still limited. Much information must still be handled in a manner particular to every language's AST. This is because many high-level components rely not on the presence of specific programming constructs, but on the way that these constructs interact to create a component. Thus, a higher level of abstraction is still needed.

A higher level of abstraction like this can be found in Klint *et al.*'s [23] metaprogramming language RASCAL. This language is specifically designed to easily navigate and transform source code in a language-dependent fashion. This offers great utility for source code analysis and manipulation, but differs from our approach in its core philosophy and applicability. As a more traditional programming language, RASCAL is an imperative language; the user must specify exact steps to accomplish a task. ReSSA, by contrast, is declarative, stating the structure to be found and letting the interpreter handle finding it. This makes RASCAL more flexible and generally applicable, but leaves it up to the individual scriptwriter to properly implement the project traversal and any needed optimizations. ReSSA, as a declarative language, allows the library maintainer to handle optimizations–which, given how complex ReSSA can become, is of utmost importance. This also means that future optimizations are grandfathered in by all existing ReSSA scripts, reducing maintenance overhead. The second difference, applicability, comes from the fact that RASCAL scripts do not operate on a language-agnostic interface; instead, one would need to translate between language-specific APIs (such as Java, C++, etc.) to be able to handle multiple languages. This is not at all impossible, but it is undoubtedly overhead and complexity that it would be best to avoid.

A semi-automated approach to detect microservice components (services, endpoints, etc.) for architectural reconstruction was proposed by Ntentos *et al.* [24]. This approach utilizes "Detectors" to extract information into "Matches." Once enough Matches are created to identify an architectural component, an "Evidence" is created for this component, to be verified by manual inspection of the linked source code. This is quite different from our approach, which seeks to create an automated, generic component detection strategy using hierarchical parsers. Similarly, their paper gives no indication that a language-agnostic representation is used as the underlying architecture for the Detectors.

Moving on from semi-automatic detection, automatic endpoint detection has been accomplished for various programming languages through a tool named Swagger [25]. This tool offers automatic generation of the API documentation for multiple programming languages; while this offers great utility, it is fundamentally different from our application. The API documentation generated by Swagger will contain all of the endpoints and their corresponding URLs, request and response type information, and header information, among others. However, Swagger performs static analysis for endpoint detection in a language-specific fashion, while

```
let y = 11;
let x = Foo::Bar(10);
let z = match x {
    Foo::Baz | Foo::Bar(_) if y < 10 => 5,
    Foo::Baz | Foo::Hello if y > 11 => 6,
    v @ Foo::Bar(_) if y == 100 => 7,
    Foo::Bar(n) => n * n,
    varname => return None,
};
```

LISTING 1. A complex match expression in Rust example.

```
var y = 11;
var x = new Foo.Bar(10);
int z;
if (
    (x instanceof Foo.Baz ||
    x instanceof Foo.Bar) && y < 10
) {
  z = 5;
} else if (
    (x instanceof Foo.Baz ||
    x instanceof Foo.Hello) && y > 11
) {
  z = 6;
} else if (x instanceof Foo.Bar && y == 100) {
  var v = x;
  z = 7;
} else if (x instanceof Foo.Bar) {
  var n = x.bar();
  z = n * n;
  } else {
  var varname = x;
  return null;
}
```

LISTING 2. A Java equivalent of the complex Rust match expression.

ReSSA aims to perform static analysis of more generic software components in a language-agnostic fashion.

Finally, recognizing components in Java-based systems has been a practice used by many researchers to facilitate code-to-code and code-to-model transformations [26], [27]. However, this analysis on Java code is simplified by the existence and broad range of component standards involving annotations and the reflections library. This mechanism enables code introspection and thus the identification of classes with specific annotations. This means that specific component types can be easily located in the system byte-code. However, this approach cannot be generalized across systems, because other languages do not have bytecode. Another limitation is that the system has to be compiled in order to operate with the system metamodel. In many cases, this is impractical.

III. PROBLEM ANALYSIS

The challenge of agnostically extracting high-level concepts from source code must necessarily overcome two problems. The first problem is the language barrier. In modern software development, many languages are used extensively (such as Java, Java/Typescript, C++, etc.), which must all be analyzed to get useful information. While various solutions exist to handle the analysis of one language at a time, there are none that can "ignore" this language distinction. When we consider the abstraction of common structures (basic control structures, classes/interfaces, functions, and so forth), creating a stable abstraction for static-code analysis tools is possible. Early efforts enabling such a direction include Mozilla's rust-code-analysis crate [28], which brought many Concrete Syntax Tree (CST) parsers together in one language using one tree data structure. However, rust-code-analysis was limited by not uniformly formatting and categorizing their CSTs, leaving them still language-specific. Thus, taking this as an intermediate model, we can convert these language-specific forms into a Language-Agnostic AST (LAAST).

This leaves one more hurdle: different technologies, frameworks, and design approaches largely determine how similar high-level concepts are organized in the LAAST. This problem is far harder to solve than the previous one. While there are common abstractions that programming languages follow, no such abstractions exist for generic high-level concepts, like data entities or endpoints. To demonstrate, let us consider endpoint calls and data entities as simple concepts with radically different LAAST representations.

Endpoint accesses, (ex., when one microservice calls an endpoint on another), differ greatly across various frameworks. To demonstrate, let us consider the ways three different systems make remote calls:

Remote calls: C++/Apache Thrift

Apache Thrift [29] and gRPC [30] frameworks are used by the DeathStarBench system [4]. This system defines RPC endpoints in a configuration file, and accesses these endpoints using a `ClientPool<ThriftClient<XServiceClient>>`. Here, `XServiceClient` is the service in question. Calling `GetClient` on an item in this pool generates an object which can execute a remote call.

Remote calls: Javascript/Axios for Node.js

Javascript's Axios client for Node.js [19] uses a simple method call on the imported Axios instance to make all endpoint calls.

Remote calls: Java Spring

Java Spring [31] uses a WebClient to construct a request using chained or consecutive method calls, with `WebClient#uri(URI destinationUri)` defining the endpoint name.

Clearly, then, there are many ways to make a remote endpoint call.

The situation does not improve with entities, either. To extract all entity schema information possible, one must be able to parse any occurrence of an entity's attributes in the program, be it in inserts, queries, or updates.

Entities: C++/Apache Thrift

Within DeathStarBench [4], there are three distinct ways data attributes are used:

- A JSON object and a Data-Transfer Object (DTO), for inserts and object-returning queries.
- Specifying a MongoDB [32] query in parameters to methods.
- Tying an attribute directly to an object linked to an entity with a call to `BSON_APPEND_X` (where X is the attribute's type).

Entities: Java

In Java Hibernate [33], an ORM is used to specify entity attributes in a single, centrally-located format.

Entities: Python

With SQLAlchemy's non-ORM variant [34], methods are chained together to programmatically specify parts of an SQL query.

Clearly, parsing these different structures requires radically different solutions.

These differences may seem to necessitate completely separate programs for parsing each different high-level concept one may wish to analyze–a clearly impractical task. However, in software engineering, old problems often appear in new forms that can be easily solved by applying tested solutions. LAAST is simple to traverse and contains the data required in a well-defined format. Thus, providing a simple grammar describing the structures within the LAAST that contains the information we want and how to record this data allows us to solve the problem in a manner that, while not 100% language-agnostic, massively simplifies the task.

IV. HIGH-LEVEL CONCEPT PARSING APPROACH

In order to detect high-level concepts as described in Section III, our approach has two steps. First, we must convert the project being analyzed into its Language-Agnostic AST (LAAST) representation. Secondly, using this intermediate representation, we can uniformly define Relative Static Structure Analyzers (ReSSA) to extract information about the concepts within the project. With this high-level overview completed, we introduce each of these in more detail and describe how they connect to detect these high-level concepts.

A. LANGUAGE-AGNOSTIC AST (LAAST)

The LAAST needs to abstract away the differences between languages to allow them to be parsed uniformly. This will enable the development of a universal ReSSA interpreting architecture to use with any language, properly separating the concern of language parsing from AST node pattern matching.

Within the LAAST, it is essential to have representations for common control flow structures and expression types common across programming languages. By focusing on including these in the LAAST, ReSSA parsers need not be written with language specifics in mind.

The primary constructs to represent include classes and their fields, functions, parameters, control flow statements, variable declarations, and the various expressions and statements inside of function bodies. ReSSA will commonly be searching for variable declarations, call expressions and their arguments, specific usage of variables in functions that may be associated with function calls, etc. Due to this, function calls and variables are particularly crucial to include. Storing this information in the LAAST will allow ReSSA to target important nodes in the analyzed code.

There are cases where language-specific constructs need equivalent or distinct representations so that ReSSA can match them. An example of such a construct is the ternary operator. Not every programming language supports this operator, but it can be equivalently represented by a call to an inline lambda function containing an if-else statement, with a condition and a return statement in each branch.

A more complicated example than a simple ternary operator is a structure such as the Rust programming language's match expressions. Match expressions in Rust can be thought of as a highly expressive combination of a switch statement and if-else statement(s) with pattern matching capabilities. For example, pattern matching capabilities include executing a "match arm" or branch for a specific enum variant, destructuring values, binding variables, or matching string values. This structure is highly specific to Rust but can still be converted to a general representation, and even as Java code. Listing 1 and Listing 2 illustrate the Rust match expression example which demonstrates general flexibility in translating language-specific constructs.

Another example is Go's defer operator. Depending on how a specific LAAST implementation represents this construct, there could be information loss. For example, representing the defer operator as a try block with all of the function's statements in it, with a call to the defer operator's function argument in the finally block would be lossy since nodes are being reordered. However, this could be represented without loss as a function call to a function named defer that takes another function as an argument. The way in which these constructs are decided upon is an implementation decision, and it does not impact the approach itself. However, it does affect the constructs that will be matched in ReSSA.

1) LAAST IMPLEMENTATION

Our implementation is based on the AST format given by Prophet graph code representation [35]. This format has been used for software architecture reconstruction [36], code smell analysis [37] in distributed system design, and assessing security enforcement consistency [38]. The Prophet AST format was extended with function bodies, a wide range of control flow structures, variable declarations, statements, and expressions. This ensured flexibility for LAAST structures as well as compatibility with existing static code analysis tools and instruments. While the Prophet AST format was originally made to represent Java code, it proved easily extendable to capture other programming language constructs and method bodies in a language-agnostic fashion.

Our LAAST implementation was built off of Mozilla's rust-code-analysis crate [28]. This allowed us to leverage pre-existing language parsing for many programming languages into language-specific Concrete Syntax Trees (CSTs), thanks to the underlying usage of Tree-sitter. Tree-sitter is a parser generator and an incremental parsing library to build a CST for a source file. We then transform these language-specific CST representations into an equivalent LAAST structure for Java and C++, to be used for component matching through ReSSA.

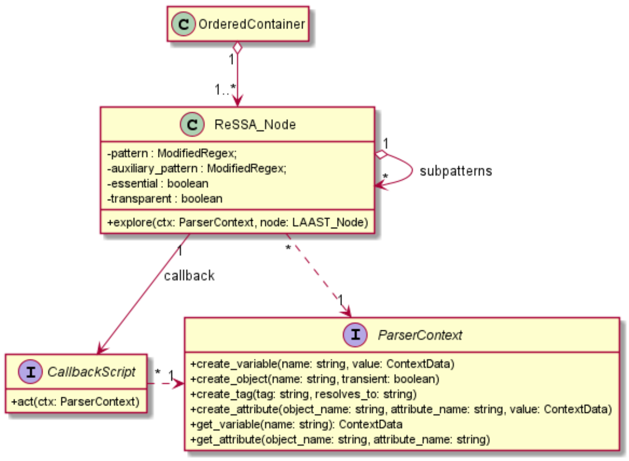

OrderedContainer
1
1..*
ReSSA_Node
-pattern : ModifiedRegex;
-auxiliary_pattern : ModifiedRegex;
-essential : boolean
-transparent : boolean
+explore(ctx: ParserContext, node: LAAST_Node)
1
*
subpatterns
1
callback
*
1
ParserContext
+create_variable(name: string, value: ContextData)
+create_object(name: string, transient: boolean)
+create_tag(tag: string, resolves_to: string)
+create_attribute(object_name: string, attribute_name: string, value: ContextData)
+get_variable(name: string): ContextData
+get_attribute(object_name: string, attribute_name: string)
CallbackScript
+act(ctx: ParserContext)
*
1

FIGURE 1. **UML of ReSSA parser structure.**

B. RELATIVE STATIC STRUCTURE ANALYZERS (ReSSA)

The purpose of ReSSA is to provide sufficient tools to extract information out of the LAAST. It is aimed at allowing users to specify a set of grammar rules describing what some component of interest would look like within the LAAST (say, a database entity), and get a report providing the names of entities, their attributes, types, and other details.[1]

At a high level, the grammar relies on trees of *parsers*, each parser describing a node type we expect to find in the LAAST. If the *patterns* within the parser match information within the node, then this parser's child parsers are executed as well, recursively searching the descendants of the LAAST node. Once all child parsers have been run, and all *essential* child parsers have matched, the current parser feeds collected information back into the *parser context* via *callbacks*. Once all parsers are run, the parser context is processed and reported back to the user as the final output. This structure is illustrated in Fig. 1. Moreover, Fig. 2 highlights an example of LAAST and ReSSA on a simple class.

With this abstract baseline, let us investigate each part and its realization in more detail. The below subsections provide more details on *parsers, patterns, the parser context, and callbacks*.

1) PARSERS AND PARSER TREES

The parser trees are built up from *root parsers* stored in an ordered list. These root parsers scan the full program LAAST, with each applied against every node in the LAAST once. Root parsers run in order of appearance within the list. Such a setup enables the ReSSA's designer to extract information in a specific order, building up a knowledge of the project as needed. For example, one can start by finding all calls to microservices in the system, then use this information to identify what microservices exist and which classes are services. This bypasses situations where the information needed to assemble a component is spread widely across the codebase. Continuing the above example, this covers situations where the evidence needed to prove a class is a microservice is in a different source code file.

If a root parser finds a match, it delegates it to its child parsers, which recurse downwards in children of the LAAST node to test if they find a match anywhere in the subtree. This process continues until all child parsers have matched or failed. If a child parser is marked *essential*, then its failure to match anywhere short-circuits the search and returns to the node the root parser matched, and the root parser resumes its search. This allows sequences of expected nodes to be defined loosely for information extraction, such as defining the set of nodes corresponding to method calls on an object already determined to be a Remote-Procedure Call (RPC) proxy to an endpoint.

Parsers match if three conditions hold:

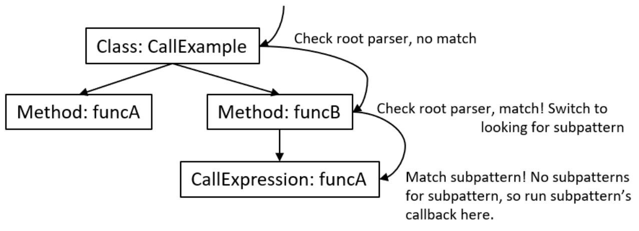
1) The type of the LAAST node must match the type of node the parser is looking for–for instance, a method-call LAAST node compared to a method-call parser node.
2) The modified regex patterns within the parser node must return a match against the text of importance in the LAAST node–this can include variable names, method names, and so forth. The details of the extended regex syntax are discussed in the next subsection.
3) All child parsers (if present) are run against descendants of the node that was matched by the previous two criteria; if all child parsers denoted as *essential* match somewhere in the subtree, then the parser's callback (if defined) is run, and the node reports a match to its parent node.

If any of these conditions fail to hold, then the parser fails to match, no changes to the parser context are persisted, and the parent is alerted that the parser failed.

The only notable exception to the above rules is nodes marked as "transparent"–these ignore all logic in the node itself, instead transparently forwarding every node to its

[1]For a detailed description of how our ReSSA implementation works with regards to all the following topics, see https://github.com/cloudhubs/source-code-parser/blob/initial-paper/ressa_specification.md

```
class CallExample {
  void funcA() {}
  void funcB() { funcA(); }
}
```


Class: CallExample
Check root parser, no match
Method: funcA
Method: funcB
Check root parser, match! Switch to looking for subpattern
CallExpression: funcA
Match subpattern! No subpatterns for subpattern, so run subpattern's callback here.

```
[{
    "identifier": "Method",
    "pattern": "#{call_from}",
    "subpatterns": [
        {
        "identifier": "CallExpression",
        "pattern": "#{call_to}",
        "subpatterns": [],
        "callback": "/* Write to context */",
        "essential": true
        }
    ],
    "callback": "println(\"hello world!\");",
    "essential": true
  }]
```

FIGURE 2. **Left: a simple class. Center: a sample LAAST of the class with ReSSA's parsing process highlighted. Right: a ReSSA parse.**

subpatterns. This allows the user to expect one of a set of structures, instead of a single structure, and have a single callback operate on them.

Parsers are limited to matching LAAST nodes that contain user-defined information that may be of interest: for example, method calls, variable declarations, class definitions, and so forth. Other nodes–such as if statements and loops–transparently forward parsing to their child nodes instead of handling logic. Some ReSSA parsers treat their subpatterns specially (for example, method call expressions performing a subsequence match against the arguments to the method call based on their child patterns).[2]

2) PARSER PATTERNS

The modified regex within the parser's patterns is extended with extra syntax to define variables–substrings that are parsed that should be extracted into the parser context under a defined name. The second type of variable, important in many edge cases, is reference variables, where the matched string must name an existing object within the parser context. For the actual pattern matching, the variable's definition is replaced with a valid regex; while this defaults to a wildcard ('$.*$') in our implementation, it should be specifiable by the user within the ReSSA.

3) PARSER CONTEXT

The parser context stores information extracted by the parser's modified regexes and defined by the user. There are three types of information contained within the parser context:

- Variables: Variables behave similarly to local variables in programming languages. They come into existence upon being matched by a pattern and are not deleted until either A) the parser that matched it fails, or B) the root parser returns (at which point, all variables are cleared). Variables are overwritten when the modified regex defining them matches again.
- Objects: Objects behave similarly to objects within programming languages like Java, each having a name within the context and a set of named attributes. Objects cannot have methods, similar to structs in C, but whether they allow storage of callbacks is up to the implementation. All objects not marked transient will be returned to the user as the final output.
- Tags: An alternate name for an object. Tags allow information that is linked tangentially by variable names or other in-code conventions to be comfortably tied together. They can be seen as a strict subset of transient objects to handle a common use case.

4) CALLBACKS

Callbacks are scripts defined on various parsers, allowing the user to read and write information to the parser context while also executing complex logic if needed. These should be restricted from executing dangerous instructions, like file writes and network communication, to maintain the safety of running ReSSAs from untrusted sources.

V. CASE STUDIES

To evaluate the ReSSA approach and prove its merit in solving static-code analysis problems, we performed a case study on two large, community-adopted, and heterogeneous microservice benchmarks (Java and C++ based). For the baseline of our results, we utilized manual identification of endpoints and entities in DeathStarBench and TrainTicket to compare our results with. To do this sort of static code analysis without ReSSA, it would require separate implementations for DeathStarBench and TrainTicket due to language-specific ASTs and different targeted code patterns, while our approach only requires separate ReSSAs to be defined. In this study, we sought to show ReSSA's potential for detecting various components by creating ReSSAs to automatically detect endpoints and data entities.

[2]For a full description of what ReSSA parsers exist and what (if any) special rules apply to them, see https://github.com/cloudhubs/source-code-parser/blob/initial-paper/ressa_specification.md#parsers.

We chose these components for our case study for two reasons. First, identifying data entities and endpoints is an important problem to software development today. Architectural reconstruction, reverse-engineering the design of an already-written piece of software [36], is pivotal for creating centralized views of highly decentralized systems [6], [36], identifying many microservice smells [39], and detecting architectural degradation [40]. However, to perform an architectural reconstruction, one must first know the data entities and endpoints within a system. This makes such identification an important task. Second, identifying endpoints and data entities from code is nontrivial. As detailed in Section III, there is no universally applicable pattern to strip these out of source code. Interpreting how they are used and extracting them requires extensive creative thinking, ranging from piecing together chains of method calls to interpreting embedded statements at arbitrary points in the code. Creating a single tool that offers all constructs needed for this process, funnels the extracted results into a single output, and provides a uniform interface for downstream analysis, is no trivial task. While there are other relevant components one could focus on to demonstrate this, we feel the chosen types are complex enough to prove ReSSA's potential for solving component-identification problems without bogging down the study.

For our case study, we first show how to approach identifying endpoints throughout the system (e.g., REST calls or RPC calls) to detect inter-module interaction and reveal call dependencies. This detects the initial set of services and their dependencies across the distributed system's modules. Next, we demonstrate a more application-specific approach where we expect the system architect to assess the system for code design conventions and develop a custom, project-specific ReSSA specification for the particular system's component detection. Such an approach yields strong precision and recall to detect components in the benchmark code. Finally, we apply a framework-generic approach to detect data entities and their attributes. By framework-generic, we mean that the same rule can be applied to other systems built on the same framework.

Using two different benchmarks in this study demonstrates the versatility of ReSSA to abstract and involve the same intermediate program representation model to detect components on systems with no common language, framework, or conventions. The results listed here can be reproduced through our Postman test suite on GitHub, and the benchmarks are reproducible on Linux with instructions specified in the README file.[3]

A. TESTBEDS

Our two target testbeds, DeathStarBench [4] and TrainTicket [5], [41], are produced by third-party researchers and are meant to work as benchmarks.

[3]https://github.com/cloudhubs/source-code-parser/blob/initial-paper/ressa_examples/reproducibility_tests.postman_collection.json

```
ClientPool<
  ThriftClient<#{service_name}ServiceClient>
  > #{pool_name};
   ..
auto #{wrapper_name} = #{pool_name}->Pop();
   ..
auto #{client_name}
          = #{wrapper_name}->GetClient();
   ..
#{client_name}->#{endpoint_name}(...);
   ..
public class #{callee_name} {
      ...
        #{client_name}->#{endpoint_name}(...);
      ...
}
```

LISTING 3. Conventions: DeathStarBench internal RPC calls on microservices.

The heterogeneity of both testbeds allows us to demonstrate the versatility of ReSSA in analyzing varying languages with very different approaches for structuring services, endpoints, and data entities within the code. Details on the particular testbeds and ReSSA solutions are provided in the following subsections. Each subsection also reports on the precision and recall of our approach compared with the ground truth determined by manual analysis.

B. DEATH STAR BENCH

DeathStarBench is a benchmark suite for cloud microservices written in C++ and built on Apache Thrift using MongoDB for the persistence of data entities. DeathStarBench includes five end-to-end services, four for cloud systems and one for cloud-edge systems running on drone swarms. It contains a set of six microservice systems: *Social Network, Media Service, Hotel Reservation, E-commerce site, Banking System* and *Drone coordination system*. However, not all systems were released at the time of this study.[4]

1) ENDPOINT DETECTION IN INTER-SERVICE COMMUNICATION

To identify inter-service communication and detect interacting endpoints from the source code of DeathStarBench as a test run for ReSSA, we identify a call graph of RPC calls within the microservice systems. While this approach inherently misses externally exposed endpoints and endpoints not called within the microservice system set we are analyzing, our goal is to prove that ReSSA can solve a problem like this.

To implement this ReSSA, we assessed the *Social Network* microservice collection to identify applied code design conventions of the benchmark. With the conventions identified, we extracted a generalized pattern for ReSSA to fit these endpoints. Then, we tested it against the *Media Service* microservice collection.

[4]The version we used had a commit hash of b509c933faca3e5b4789c6707d3b3976537411a9.

TABLE 1. Media service services and endpoints (identified by manual analysis).

- UniqueIdService
  - UploadUniqueId (uncalled)
- MovieIdService
  - UploadMovieId (uncalled)
  - RegisterMovieId (uncalled)
- TextService
  - UploadText (uncalled)
- RatingService
  - UploadRating (MovieIdService)
- UserService
  - RegisterUser (external endpoint)
  - RegisterUserWithId (uncalled)
  - Login (uncalled)
  - UploadUserWithUserId (uncalled)
  - UploadUserWithUsername (uncalled)
- ComposeReviewService
  - UploadText (TextService)
  - UploadRating (RatingService)
  - UploadMovieId (MovieIdService)
  - UploadUniqueId (UniqueIdService)
  - UploadUserId (UserHandler x2)
- ReviewStorageService
  - StoreReview (ComposeReviewService)
  - ReadReviews (UserReviewService, MovieReviewService)
- MovieReviewService
  - UploadMovieReview (ComposeReview-Service)
  - ReadMovieReviews (PageService)
- UserReviewService
  - UploadUserReview (ComposeReviewService)
  - ReadUserReviews (uncalled)
- CastInfoService
  - WriteCastInfo (external endpoint)
  - ReadCastInfo (PageService)
- PlotService
  - WritePlot (external endpoint)
  - ReadPlot (PageService)
- MovieInfoService
  - WriteMovieInfo (external endpoint)
  - ReadMovieInfo (PageService)
  - UpdateRating (uncalled)
- PageService
  - ReadPage (uncalled)

To highlight how DeathStarBench defines internal RPC calls on microservices within the source code, we follow the steps denoted by the lines in Listing 3:

Lines 1-3 A service client pool is defined as a class field. The service name referenced here serves as the service containing the endpoint.

Line 5 A local variable is assigned a client wrapper retrieved from the client pool.

Lines 7-8 A local variable is declared retrieving a client from the client wrapper.

Line 10 An endpoint call is made using the client, capturing the endpoint name.

Lines 12-16 The class making the endpoint call can finally be concluded to be the service making this call.

DeathStarBench's *Media Service* contains the following services, with their endpoints and services making calls on those endpoints highlighted in Table 1.

As can be observed, many endpoints were either only externally exposed (as determined from a Thrift file in the source code) or unused anywhere within the analyzed source code. As a result, we will be reporting two separate calculations for precision and recall: one indicating the success of the call-graph approach and one indicating the success of ReSSA in implementing this approach (the latter statistic being more important).

We designed a simple ReSSA to extract endpoints based off of the above listed Apache Thrift/DeathStarBench conventions. We built it to report output in the following rudimentary format:

```
"Service": {
  "Endpoint": "Service1Handler, Service2Handler, "
}
```

Here, the comma-delimited service list is the list of services calling the endpoint. Listing 4 shows the output of running our ReSSA.

Out of DeathStarBench's Media Services, we found 8 out of 13 services with endpoints. Looking into the services making calls on the endpoints, however, we find all files adhere to the `XHandler` naming convention. This, within DeathStarBench, was confirmed to be a connector to an `XService` by earlier manual inspection; factoring this information in, we find the remaining 5 services reported as making calls on the existing services, though not as having endpoints themselves. Thus, we find all services in total, with no false positives, giving us 100% for both precision and recall of services. This applies to both the call-graph approach and ReSSA's implementation thereof.

```
{
    "CastInfo": {
       "ReadCastInfo": "PageHandler, "
    },
    "ComposeReview": {
       "UploadMovieId": "MovieIdHandler, ",
       "UploadRating": "RatingHandler, ",
       "UploadText": "TextHandler, ",
       "UploadUniqueId": "UniqueIdHandler, ",
       "UploadUserId": "UserHandler, UserHandler,"
    },
    "MovieInfo": {
       "ReadMovieInfo": "PageHandler, "
    },
    "MovieReview": {
       "ReadMovieReviews": "PageHandler, ",
       "UploadMovieReview":"ComposeReviewHandler,"
    },
    "Plot": {
       "ReadPlot": "PageHandler, "
    },
    "Rating": {
       "UploadRating": "MovieIdHandler, "
    },
    "ReviewStorage": {
       "ReadReviews": "UserReviewHandler,
    MovieReviewHandler, ",
       "StoreReview": "ComposeReviewHandler, "
    },
    "UserReview": {
       "UploadUserReview": "ComposeReviewHandler,"
    }
}
```

LISTING 4. ReSSA results on inter-service endpoint calls.

With regards to the overall endpoints of the *Media Service*, things are a bit more complicated. After inspection,

TABLE 2. Missed endpoints in the media services.

- UniqueIdService
  - UploadUniqueId
- MovieIdService
  - UploadMovieId
  - RegisterMovieId
- TextService
  - UploadText
- UserService
  - RegisterUser
  - RegisterUserWithId
  - Login
  - UploadUserWithUserId
  - UploadUserWithUsername
- UserReviewService
  - ReadUserReviews
- CastInfoService
  - WriteCastInfo
- PlotService
  - WritePlot
- MovieInfoService
  - WriteMovieInfo
  - UpdateRating
- PageService
  - ReadPage

we determined that multiple endpoints were missing from our output as given by Table 2.

Of all these missed endpoints, however, all were uncalled in the C++ source code as per manual inspection; instead, they were defined in Lua[5] and Thrift files. All endpoints which were called somewhere in the C++ source code were captured. No non-endpoint calls were reported as endpoint calls.

In total, our precision on endpoint call detection was 100% once again. However, the recall of the approach suffered, finding only 14 of the 29 endpoints, or 48% for the call-graph approach. This proves the call-graph algorithm insufficient to identify all DeathStarBench's endpoints (including those not called from other system services). However, this is expected, as not all endpoints are meant for inter-service communication and are instead exposed to third-party systems.

Thus, a more important statistic is the strength of ReSSA's recall for implementing the algorithm. This statistic is computed based on how many times each endpoint was called within the code (instead of how many endpoints there are total) and how many of these endpoint calls we were able to identify. For this, we detected 16 out of 16 total calls made. Some endpoints were called repeatedly; specifically, there were 14 unique endpoints, with 16 calls total. This gives 100% recall.

In summary, our results indicate that the inter-service call-graph algorithm is not sufficient to identify all system endpoints. However, it does identify endpoints involved in inter-service communication, which helps to resolve service dependencies. ReSSA was able to implement this algorithm perfectly.

2) SYSTEM ENDPOINT DETECTION

Before we digress from the points covered in the previous section, two other points need to be highlighted. First, all endpoints are defined within the Thrift and Lua files in a structured fashion. This can be expected, since software teams tend to adopt certain conventions. Thus, if we developed a Lua or Thrift LAAST parser module, we would be able to parse these endpoint definition files directly. This would allow us to extract all this information with no need to infer from a call graph. Second, more relevantly to our current case study, there are alternative approaches to extracting services and endpoints that capitalize on naming conventions within the repository. While this introduces system-specificity, it demonstrates the flexibility ReSSA provides, since this might be necessary across heterogeneous solutions.

In DeathStarBench, the names of all classes that represent a service end with the suffix "Handler." Within these classes, all methods with fully alphabetic names not ending in "Handler" are endpoints. Using this information, we created a second system-specific ReSSA to analyze DeathStar-Bench for its endpoints only. The resulting ReSSA is in Listing 5.

This yielded the full set of services and their endpoints, the only difference between the ground truth and our output being the order of services or endpoints within the services. In summary, this simple ReSSA focusing around naming conventions resulted in 100% precision and recall for service and endpoint detection within DeathStarBench.

Table 3 summarizes the results of our DeathStarBench service call/endpoint detection.

```
[{
  "identifier": "ClassOrInterface",
  "pattern": "#{service_name}Handler",
  "subpatterns": [
  {
    "identifier": "Method",
    "pattern": "#{endpoint}(^[a-zA-Z]*$)",
    "subpatterns": [],
    "callback": "
     let endpoint
       = ctx.get_variable(\"endpoint\").unwrap();
     let service
       = ctx.get_variable(\"service_name\")
          .unwrap();
     if (!endpoint.ends_with(\"Handler\")) {
       ctx.make_attribute(service,endpoint,None);
     }
     ",
     "essential": true
  }
 ],
 "essential": true
}]
```

LISTING 5. ReSSA specification to detect endpoints by naming conventions.

TABLE 3. Precision and recall of identification analysis on DeathStarBench.

	Precision	Recall
Microservice	100%	100%
Endpoint Call	100%	100%
Endpoint (call-graph based)	100%	48%
Endpoint (naming convention)	100%	100%

[5]Lua is a lightweight language designed to be embedded into other languages to extend functionality. [42]

```
auto collection =
  mongoc_client_get_collection(
      mongodb_client, "cast-info", "cast-info"
);
```

LISTING 6. MongoDB query example.

```
bson_t *query
  = BCON_NEW("$and", "[", "{", "user_id",
  BCON_INT64(user_id), "}", "{", "followees",
  "{", "$not", "{", "$elemMatch", "{", "user_id",
  BCON_INT64(followee_id), "}","}", "}", "}", "]"
);
```

LISTING 7. MongoDB query example.

3) DATA ENTITY DETECTION

We approached detection of entities within DeathStarBench the same way as endpoint detection. Thus, manually inspect the *Social Network* microservice collection to determine how entities are defined, write a ReSSA targeting this, and test it against the *Media Services* microservice collection to determine precision and recall.

However, DeathStarBench references database entities in its MongoDB in three distinct ways: (1) by an inline MongoDB query, (2) by the specification of an insert utilizing `BSON data`, and (3) specification of a retrieval utilizing `JSON data` and a data transfer object. Due to this being a case study, not a comprehensive treatise on entity identification, we chose to target only one of these formats: the inline MongoDB query.

At its core, the MongoDB query relied first on a call to access a collection showed in Listing 6.

From there, a MongoDB query is made by passing a set of MongoDB query tokens to a method to generate an object representing the query. An example is shown in Listing 7, pulled from `SocialGraphHandler` in the *Social Network* microservices:

Parsing this format requires implementing a simple MongoDB query parser within a ReSSA callback. The final script, which can be found in our GitHub repository,[6] was non-trivial to implement but proved functional for our case study.

While a large amount of the MongoDB schema can be extracted from the code, not all of it was referenced in the form of inline Mongo queries. Because of this, to be fair to our approach, we consider all statistics measured against the maximum entities and attributes that are referenced in such a format. This pared down the MongoDB schema, as inferred from manual inspection of the code, to the entities and attributes in Listing 8.

Running entity-detecting ReSSA against the *Media Service* microservices yielded the scraped schema in Listing 9.

To quantify these findings, it is best to split the statistics in two: one for entities/collections identified and one for entity attributes identified.

[6]https://github.com/cloudhubs/source-code-parser/blob/main/ressa_examples/mongoQueryEntityScraper.rn

```
{
  "social-graph": {
    "avg_rating": "DOUBLE",
    "num_rating": "INT32"
  }.
  "movie-review": {
    "movie_id": "UTF8",
    "reviews": "[]"
  },
  "move-review.reviews": {
    "review_id": "INT64",
    "timestamp": "INT64"
  },
  "user-review": {
    "user_id": "INT64",
    "reviews": "[]"
  },
  "user-review.reviews": {
    "review_id": "INT64",
    "timestamp": "INT64"
  },
  "cast-info": {},
  "movie-id": {},
  "movie-info": {},
  "plot": {},
  "review": {},
  "user": {}
}
```

LISTING 8. JSON version of the MongoDB schema identified by manual analysis; attribute values are replaced with the attribute's type, and dot notation is used to identify composites within collections.

```
{
  "cast-info": {},
  "movie-id": {},
  "movie-info": {},
  "movie-review": {
      "UTF8": "reviews",
      "movie_id": "UTF8",
      "reviews": "[]"
  },
  "movie-review.[": {
      "review_id": "INT64",
      "timestamp": "INT64"
  },
  "movie-review.reviews": {
      "review_id": "INT64",
      "timestamp": "INT64"
  },
  "plot": {},
  "review": {},
  "social-graph": {
      "avg_rating": "DOUBLE",
      "num_rating": "INT32"
  },
  "user": {},
  "user-review": {
      "reviews": "[]",
      "user_id": "INT64"
  },
  "user-review.reviews": {
      "review_id": "INT64",
      "timestamp": "INT64"
  }
}
```

LISTING 9. JSON scraped schema from ReSSA entity-detection.

With regards to root collections, the numbers are simple: there are no false positives or false negatives, yielding 100% precision and recall. On the other hand, attributes proved

```
 "UTF8": "reviews"
..
 "movie-review.[": {
  "review_id": "INT64",
  "timestamp": "INT64"
}
```

LISTING 10. JSON scraped schema from ReSSA entity-detection.

TABLE 4. Entity and attribute parsing precision/recall in DeathStarBench.

	Precision	Recall
Database Entity	100%	100%
Entity Attributes	75%	100%

```
@RestController
@RequestMapping("#{root_path}")
public class #{controller_name} {
    ...
}
```

LISTING 11. Sample controller class annotations in TrainTicket.

more challenging, showing the false positives showed in Listing 10.

Upon inspection, these were all determined to be from bugs in the Mongo query parsing callback script, as opposed to a flaw within DeathStarBench or our approach. Specifically, note the reversed attribute type and name, as well as the `.[` oddity shown in Listing 10. In the end, these resulted in 75% precision and 100% recall for our approach. Table 4 summarizes the findings of the DeathStarBench entity and attribute detection.

C. TRAIN TICKET BENCHMARK

TrainTicket [5] is a microservice benchmark system comprised of 41 microservices, of which 37 are implemented in Java. For the purposes of the case study, we focused only on the 37 Java-based microservices. The project itself is a comprehensive train ticket booking system. The Java-based microservices in TrainTicket use Spring Boot [31] for their framework of choice; thus, the services are structured similarly by the nature of the enterprise annotations used in Spring Boot. This consistency makes it an ideal candidate for detection via ReSSA.

1) SYSTEM ENDPOINT DETECTION

The TrainTicket microservices expose their functions and communicate with each other via their HTTP APIs. Spring Boot makes defining the endpoints of such APIs straightforward. TrainTicket's endpoints are defined as follows:

A class is defined as a REST controller using the `@RestController` annotation, and the root path for accessing it is defined via the `@RequestMapping` annotation (see Listing 11). In TrainTicket, while one controller is defined per microservice, the name of the controller may not correspond to the name of the microservice; the name is not guaranteed unique within the project, either.

```
@#{http_method}Mapping("#{endpoint_path}")
public #{return_type} #{endpoint_method} (...) {
    ...
}
```

LISTING 12. Sample endpoint template in TrainTicket.

```
@#{http_method}Mapping(path = "#{endpoint_path}")
public #{return_type} #{endpoint_method} (...) {}

@#{http_method}Mapping(value ="#{endpoint_path}")
public #{return_type} #{endpoint_method} (...) {}
```

LISTING 13. Sample endpoint annotation templates in TrainTicket.

```
"API - service_name" : {
   "endpoint_method" :
       "http_method endpoint_path return_type"
}
```

LISTING 14. Reporting format for endpoints.

The names of the services, as defined on TrainTicket's wiki,[7] are stored in the project file names. However, this information is not accessible in the initial ReSSA implementation, and thus it is approximated in this first study. Since the package names correspond roughly to the service names and are intended as globally unique in Java, the root package of the application is used as an identifier for each service.

Methods inside the class that act as handlers for HTTP requests are annotated with a `@*Mapping` annotation. The exact annotation indicates which HTTP method this endpoint services; for example, `@GetMapping` or `@PostMapping`. The return type for the endpoint is simply the return type of the method. See Listing 12 as a template example.

There are two variants of the previous annotation to contend with: one defining the endpoint path with one of two named attributes, and one using an unnamed value. See Listing 13 as a template example.

Throughout the 37 microservices, 241 endpoints are defined. While we wanted to use the TrainTicket wiki's endpoint mapping to identify this, we determined that the wiki is for a more recent version of the system highlighting only "major" endpoints, and we thus had to reconstruct the full internal architecture manually[8] To detect these endpoints, we designed a ReSSA to extract the information as described above. The results were reported in the format in Listing 14.

Unlike DeathStarBench's more complex definitions of endpoint communication, the enterprise-standard annotations in Spring Boot allowed the TrainTicket ReSSA to be much more accurate without relying on naming conventions. All 241 endpoints were correctly identified, with no false positives, resulting in 100% precision and recall for endpoints. The identification of the services themselves suffered slightly,

[7]https://github.com/FudanSELab/train-ticket/wiki/Service-Guide-and-API-Reference (referenced December 2021)

[8]This reconstruction is available at https://github.com/cloudhubs/source-code-parser/blob/initial-paper/ressa_output/ground_truths/tt_endpoint_groundTruth.json.

```
@Service
public class #{service_name} {
    ...
}
```

LISTING 15. Service class in TrainTicket.

```
template.exchange("http://ts-contacts-service
    :12347/api/v1/contactservice/contacts",
                HttpMethod.PUT,
                requestEntity,
                Response.class);
```

LISTING 16. Simple HTTP call in TrainTicket.

as a few services were defined using the same root package within different projects, merging these services' endpoint sets together in the final output. Because our ReSSA did not take this into account, our search yielded a false negative. 36 services were reported by the ReSSA, of which one accidentally combined the information from 2 services. This yields 35 true positives, one false positive, and two false negatives, giving a precision of 97.2% and a recall of 94.6%.

2) ENDPOINT DETECTION IN INTER-SERVICE COMMUNICATION

Given the simplicity of endpoint detection in the TrainTicket system, detecting inter-service calls is the next step once the endpoints have been identified. Since the TrainTicket benchmark uses HTTP calls for its inter-service communication, we make the observation that a unique call from one microservice to another can be identified by the endpoint path and the HTTP method (GET, POST, etc.) used. This is the information we seek to extract with ReSSA, and Spring Boot again provides a consistent and easily recognizable pattern for detection.

In TrainTicket, inter-service calls are handled in the service layer of each individual microservice. These are defined using the Spring Boot `@Service` annotation and are consistently named with the name of the microservice (see Listing 15), so we can easily identify which microservice is performing the call.

The calls themselves are handled by Spring Boot's `RestTemplate` class, specifically its `exchange` method. As shown in Listing 16, the first two arguments passed to this method provide the information we need: the path and the HTTP method, respectively.

While the HTTP method is trivial to recognize, the path requires further processing. When the URL contains route parameters, the final path could be the result of appending multiple values together, including string literals and variables. Thus, we must detect an arbitrary sequence of these values being concatenated as in Listing 17.

The base path of the API may also be defined as a class field in the service. In this case, we have to store the definition of the original path and recall it when the variable is used, as shown in Listing 18.

```
template.exchange("http://ts-contacts-service
    :12347/#{path_base}" + "/" + #{path_variable}
    + "/#{path_literal}/" + #{path_variable},
                HttpMethod.PUT,...);
```

LISTING 17. Simple HTTP call in TrainTicket.

```
String #{path_base_var} = "http://ts-contacts-
    service:12347/#{path_base}"

template.exchange(#{path_base_var} + "/" + #{
    path_variable} + "/#{path_literal}/" + #{
    path_variable},
                HttpMethod.PUT,...);
```

LISTING 18. Path definition variant.

```
"Calls - service_name" : {
   "http_method endpoint_path" :
       "endpoint_controller.endpoint_method"
}
```

LISTING 19. Reporting format for endpoint calls.

Once the call was identified, it was matched with the path and HTTP method defined for an endpoint in the previous phase of the ReSSA. From this, we retrieved the name of the endpoint controller and the controller method to complete the information regarding the call.

TrainTicket contains 151 unique inter-service calls within the benchmark. Our ReSSA reported results as shown in Listing 19. Unlike the endpoints, we accurately identified only 130 inter-service calls. This was because certain inter-service calls were routed to one of the multiple services, depending on input data. This buried some URLs within conditional logic, which was not accounted for in our ReSSA. Because of this, these inter-service calls were missed.

Despite this, our ReSSA-based approach still yields strong results, with 100% precision and 86.1% recall. For the sake of space, the output of the ReSSA will not be reproduced here, but we have made the results available online.[9]

3) DATA ENTITY DETECTION

In the TrainTicket system, data entity definitions are much simpler than any other aspect in either benchmark. From manual analysis, data entities are classes provided as parameters to interfaces extending `CRUDRepository` or `MongoRepository` (and any additional types stored within these classes). However, LAAST does not currently support accessing this inheritance information, because the Prophet format it was built on did not feature it. Thus, a secondary heuristic must be used. Specifically, classes featuring Spring's `@Document` annotation should be data entities (documents) to be persisted to the (NoSQL) database [43]. By looking for these, one can–even without the proper repository information–identify such

[9]https://github.com/cloudhubs/source-code-parser/blob/main/ressa_output/train_endpoints.json (referenced December 2021)

```
@Data
public class #{entity_name} {
   #{attribute_type} #{attribute_name};
   ...
}
```

LISTING 20. Variation in path definition.

```
String requestUrl = "";
if (<condition>) {
   requestUrl = "<url_1>";
} else {
   requestUrl = "<url_2>";
}
HttpEntity requestEntity
   = new HttpEntity(request, headers);
ResponseEntity<Response> re
   = restTemplate.exchange(
        requestUrl,
        HttpMethod.PUT,
        requestEntity,
        Response.class);
```

LISTING 21. Example edge case from ts-admin-travel-service's admintravel.service.AdminTravelServiceImpl#addTravel.

entities accurately. Thus, the entity ReSSA only has to identify classes bearing this annotation, extract the class name, attribute names, and attribute data types, as shown in Listing 20.

TrainTicket contains 26 data entities. Our ReSSA accurately identifies 24 of these with one false positive. For the 24 entities correctly identified, all attributes of the entities were correctly identified. The two missed entities are `ts-user-service`'s `user.entity.User` and `ts-auth-service`'s `auth.entity.User` entities, neither of which use the `@Document` annotation in favor of implementing the `org.springframework.security.core.userdetails.UserDetails` interface instead. The false positive is `ts-travel2-service`'s `travel2.entity.TrainType`. This class has the `@Document` annotation, but is not used in a Repository or as a field of an object used in one. From cross-checking other classes within the project, other objects named TrainType exist with the exact same fields and API, indicating this was likely a copy/paste error on the part of the creators of TrainTicket. However, for completeness sake, we do not discount it as a false positive. This yields 96% precision and 92% recall for data entities.

Table 8 summarizes the ReSSA results for the TrainTicket endpoints and entities.

D. CASE STUDY CONCLUSION

Our choices of testbeds show the ability of ReSSA to deal with challenging needs. We demonstrated solutions across two different languages and approaches (C++ versus Java), with different underlying frameworks (Apache Thrift versus Java Spring), remote call methodologies (RPC versus REST), and data persistence (functional versus object-relational mapping). There were not many points of common ground between the testbeds. However, ReSSA was able to adapt to these differences to render strong results on both testbeds

TABLE 5. Precision and recall of identification analysis on TrainTicket entities and endpoints.

	Precision	Recall
Microservice	97.2%	94.6%
Endpoint	100%	100%
Endpoint call	100%	86.1%
Database Entity	96%	92%

for the chosen problems, proving its flexibility and merit in solving future problems.

1) CASE STUDY BENCHMARKING

To confirm our solution's viability performance-wise, we implemented simple benchmarks for memory usage and total runtime. Both memory and runtime metrics were collected for LAAST, and runtime metrics were collected for ReSSA. The memory statistics for the LAAST structure for both DeathStarBench and TrainTicket are compared with the memory size and time to construct its Tree-sitter AST representation. We ran these tests against each of the five ReSSAs we created for the case study. All tests were run on an Intel Core i7-8750H CPU (2.20GHz), with 32 Gb of RAM provided by two 16 Gb DDR4 RAM sticks and 500 Gb of disk space from a Samsung 970 EVO Plus M.2 NVMe SSD. We will discuss each category on its own.

As outlined in Table 6, constructing the Tree-sitter AST representation and the LAAST representation were both on the order of milliseconds. However, we can see that the mean time taken to construct the LAAST from the Tree-sitter AST is consistently lower than constructing the Tree-sitter AST itself. That being said, the API exposed for constructing the Tree-sitter AST did not allow us to provide data from in-memory, instead accepting a file name to read from. This inflated the benchmark times for Tree-sitter, though this was minimized by our using a solid-state drive.

Additionally, Table 7 indicates that our LAAST structure required less memory to store than the Tree-sitter CST representations tested for DeathStarBench and TrainTicket. The increased memory usage in Tree-sitter's representation could feasibly be attributed to storing nodes at a very granular level, including nodes for keywords and other syntactical elements, while LAAST does not.

Table 8 outlines the time taken to run the ReSSAs created for this case study on DeathStarBench and TrainTicket. Our data shows that the complexity of the ReSSA's callback largely affects runtime. For DeathStarBench, the simple convention-based endpoint detection ReSSA had a mean execution time almost 10 times faster than the call-graph endpoint detection ReSSA. Similarly, the entity detection ReSSA for DeathStarBench, which had the most complex callback used in analyzing this benchmark, had a mean execution time 2.5 times slower than the call-graph ReSSA. We see similar results with TrainTicket, as the endpoint detection ReSSA contained a more complex callback than the

TABLE 6. **LAAST run-time statistics.**

	Mean	Standard Deviation	Median
Tree-sitter CST + Disk I/O (DeathStarBench)	159.91 ms	12.88 ms	156.79 ms
LAAST (DeathStarBench)	32.05 ms	3.13 ms	31.14 ms
LAAST Total* (DeathStarBench)	185.10 ms	17.26 ms	180.33 ms
Tree-sitter CST + Disk I/O (TrainTicket)	489.85 ms	136.31 ms	413.24 ms
LAAST (TrainTicket)	72.11 ms	16.90 ms	65.52 ms
LAAST Total* (TrainTicket)	507.23 ms	113.24 ms	457.77 ms

[*] LAAST reports the benchmark converting TreeSitter's format into LAAST's format. LAAST Total reports the whole process start to finish–reading the files, generating the TreeSitter CST, and converting it to a LAAST.

TABLE 7. **LAAST memory size statistics (in kilobytes).**

	Mean	Standard Deviation	Median
Tree-sitter CST (DeathStarBench)	7609.0 Kb	27.5 Kb	7607.4 Kb
LAAST (DeathStarBench)	3033.9 Kb	49.5 Kb	3038.9 Kb
Tree-sitter AST (TrainTicket)	23038.7 Kb	15.7 Kb	23040.2 Kb
LAAST (TrainTicket)	16809.6 Kb	76.0 Kb	16812.9 Kb

TABLE 8. **ReSSA run-time statistics.**

	Mean	Standard Deviation	Median
Endpoint–Simple (DeathStarBench)	7.12 ms	349.16 μs	6.97 ms
Endpoint–CG (DeathStarBench)	75.39 ms	1.81 ms	74.82 ms
Entity (DeathStarBench)	186.26 ms	19.49 ms	180.34 ms
Endpoint (TrainTicket)	8723.5 ms	1519.8 ms	9085.7 ms
Entity (TrainTicket)	93.53 ms	9.98 ms	91.94 ms

TrainTicket entity detection ReSSA and took nearly 10 times as long to complete.

2) LIMITATIONS

While our approach does offer a number of benefits over hardcoding the analysis on a per-technology level, it still does suffer from a number of drawbacks.

First, ReSSAs require the user to handle many edge cases in the structure themselves. For example, in the TrainTicket case study, the ReSSA to detect calls from one service to another had to be designed to handle multiple possible ways the endpoint URL could be defined in the call. When discussing the case study, we covered three different ways this was done and how we made our ReSSA to handle them. However, there are other ways one could define an endpoint call that would be missed by this ReSSA. For example, in parts of the codebase not among the microservices we studied to create the ReSSA, Train-Ticket's creators made calls like in Listing 21. This case followed a slightly different structure in the AST, resulting in calls following this structure not being detected. This does not mean ReSSA could not capture this case; it simply highlights that care must be taken while constructing a ReSSA to capture component definitions within the code.

Second, there is the issue that only information used or referenced within the source code can be identified. This was seen clearly by the `WriteCastInfo` endpoint within DeathStarBench's `CastInfoService`; this endpoint was not called by any internal services and thus was missed by our ReSSA. Similarly, given that ReSSA only uses a LAAST representation of source code, it will also miss the information stored within configuration files. This problem can be exacerbated in the presence of shared database models, as all entity information present in the backend might be split across multiple microservices. This is not the only problem with such an approach, given the observations by Pigazzini *et al.* [39] that this is a code smell, but it does provide yet another challenge to our approach.

Third, the approach itself can be somewhat slow to execute. This is because, in a worst-case scenario, every parser may be applied against more or less every node (save where the bottom levels of nodes ''fall off'' the end of the tree and are not executed). This speed, however, is a lesser concern thanks to this type of analysis likely being run only a few times.

Fourth, our implementation was limited by our modified Prophet AST structure not capturing all requisite data. Specifically, within our DeathStarBench case study, the `WriteHomeTimelineService` within DeathStar-Bench's *Social Network* microservice collection required global constants to be captured, and our current implementation does not parse these. Similarly, in TrainTicket, inheritance information was needed to perfectly detect data entities. These future expansions will require trivial modification of our LAAST implementation, and a minor extension of the ReSSA specification to provide access to the needed extra fields.

Fifth, our ReSSA implementation did not feature a strong awareness of what module code came from. This was thanks to not including module nodes as targetable for high-level concepts within code and not including module fields on ClassOrInterface components. During our case study, this yielded some problems, as seen in TrainTicket's requiring some degree of module-awareness to properly avoid false positives on service identification. However, this could be fixed in a future iteration of the ReSSA specification and implementation.

Sixth, our design and implementation of ReSSA did not handle control flow structures as targetable for high-level concepts within code. Instead, we searched the expressions and statements that could be contained in various parts of the control flow structures.

Finally, our approach requires ReSSA to be made using domain-specific knowledge of the high-level concept's LAAST structure within a given application. This requires a manual inspection of the codebase to determine how the technology structures targetable high-level concepts, plus the potentially non-trivial definition of how to extract this information programmatically. For a single isolated project, this somewhat defeats the purpose of automating the task, since the desired information would be identified in the manual inspection. However, it would pay off in the long run for system maintenance and tracking system evolution. Furthermore, unlike manual inspection, a completed ReSSA can be shared and reused to automate endpoint, entity, or other component detection in the future, for management, reporting, and continuous integration purposes.

E. THREATS TO VALIDITY

Case studies usually suffer from several threats to validity that need to be addressed. We tried to eliminate the effect of these threats on the study's outcome. We discuss the validity threats from the perspective of Wohlin's taxonomy [44]. It includes four potential threats: external validity, construct validity, internal validity, and conclusions validity.

1) CONSTRUCT VALIDITY

Construct validity is the degree to which our scales, metrics, and instruments measure the properties they are intended to measure [45]. Our case study intends to demonstrate how ReSSA can detect high-level programming constructs such as components with a specific purpose. By choosing two benchmarks, we aimed at reducing the threat to validity where one benchmark would significantly simplify the process. This is often the case for studies operating only on the Java platform, which can take advantage of industry-standard annotations. Thus, we picked two different representative benchmarks of up-to-date cloud-native systems recognized in the field of software and system engineering—one based on Java and the other on C++.

To avoid skewed results in our component identification, we considered two types of components: data entities and endpoints. Because these components are defined completely differently within the code, they are good representatives for our study. In addition, these are incredibly common components found in many frameworks and commonly used in system analysis. However, other component types which were not covered in our study do exist.

While we chose relatively representative benchmarks, there are many other languages and frameworks that utilize these components. These languages–such as Python and Go–may differ in the way components are defined within the code, likely preventing the ReSSAs we wrote from identifying components within them. Still, these languages have parsers to produce language-specific CSTs, and converting these to a LAAST could enable the identification of software components by creating new ReSSAs targeting the language's specific approach.

2) INTERNAL VALIDITY

Internal validity challenges the methods employed to study and analyze data (e.g., the types of bias involved). A potential threat to this study would be inaccurate manual identification of components in the benchmarks. To address this threat, two experienced developers assessed the results individually, matching the outcomes. However, all the benchmark code is publicly accessible and can be validated.

Another threat to validity is the performance analysis. This is because Tree-sitter, unlike our LAAST approach, does not offer an interface to load the source code from in-memory. This results in Tree-sitter's statistics being inflated by disk I/O operations. To handle this, we included a third statistic showing the combined time and memory consumption for creating a LAAST from start to finish, including the underlying Tree-sitter setup and disk I/O. All three statistics should be considered somewhat experimental; however, instead showing what time and memory statistics can be expected from our approach compared with a modern framework.

3) EXTERNAL VALIDITY

External validity concerns the ability to generalize knowledge. Our benchmarks were chosen as realistic examples of modern coding practices to avoid demonstrating our approach on a special case. This increases the general applicability of our conclusions.

Since ReSSA is meant to generalize component identification and operates on a LAAST, we must assume that all languages can be converted to a LAAST representation in order to utilize ReSSA. Because most languages support an AST representation, and there are common development practices behind the frameworks we need to identify components within, it seems logical that this would work. However, our case study only designed LAAST representations for Java and C++.

Both Java and C++ are object-oriented programming languages. Creating a LAAST structure that captures every single programming language may or may not prove to be feasible. Something that would need to be accepted is that there would be unused node types depending on the programming language the LAAST was parsed from. Effectively including other general-purpose languages like Rust, Pascal, and C# into the LAAST originally designed for Java and C++ may even on its own prove to be difficult. There may also be some operations in different programming languages with the same syntax that have different semantic meanings. For example, == will behave differently in JavaScript than in Rust, where the type will be enforced to be the same in the latter but not in the former. In these cases, it may be better to leave the ambiguity in the LAAST, given that separating it into a new LAAST node causes it to be more language-specific.

Extending LAAST to weakly typed languages may prove difficult while maintaining integration with other strongly typed languages. Specifically, in languages like JavaScript, objects can be defined inline with no prior type definition. Handling such cases may require extending LAAST to encompass two vastly different paradigms regarding what an object looks like, an extension that does not currently exist.

Due to the fact that ReSSA is aimed at identifying structures within code, some types of static code analysis will be much easier to implement manually. A prime example of this is the Rust [28] borrow checker and ownership system. In this, there is no single distinct node pattern being looked for, but rather invalid passing of value references or moving of value ownership. Since there may be more types of static analyses that are more easily implemented manually, the utility of ReSSA under those circumstances is less.

Finally, there is the relevant question raised by Ntentos *et al.* [24]. In their example, they stated that endpoints might be stored in configuration files and environment variables, making it difficult to determine URLs programmatically. If these values are not stored in any location where static analysis could read them, then performing a proper architectural reconstruction will become error-prone, if not impossible. However, if these values are stored in a config file parsed into the LAAST, the problem does not seem insurmountable. That said, it would require more advanced LAAST analysis tools than currently exist.

4) CONCLUSIONS VALIDITY

Conclusions validity concerns whether the conclusions are based on the available data.

Our study concludes that ReSSA can detect high-level code components through static analysis, as long as these components are describable using distinctive patterns within a LAAST. This conclusion is limited to what we have observed in the case study. Thus, it is limited to detecting endpoints and data entities in Java programs using Spring Boot and C++ programs using Apache Thrift. That being said, there is no reason to believe that it is not more generally applicable to extracting different types of high-level components from other languages and frameworks intended for enterprise system development. However, this has not been empirically assessed.

VI. CONCLUSION

This paper presents a novel approach to statically extracting high-level components from software system code written in different languages. The proposed solution is twofold. First, the source code is converted into a Language-Agnostic Abstract Syntax Tree (LAAST). Next, the LAAST is analyzed using an algorithm designed to extract components from this tree regardless of the original language input, though still reliant on language-specific Relative Static Structure Analyzers (ReSSA). As a result, the component extraction is handled in a declarative way with minimal coding demands, thus enabling engineers to focus fully on identifying and codifying the structure to be extracted rather than on the process. This approach was implemented[10] and evaluated on a case study using two different respected testbeds, DeathStarBench and TrainTicket, verifying its applicability to extract components defined in radically different ways across languages.

The approach provides a unified interface to extract a broad range of component types and can scale across platforms. It shows strong potential to transform static code analysis practices. Currently, static analysis operates on low-level programming constructs. However, this is not sustainable given the growing complexity of modern systems and related challenges (software evolution, architectural degradation, understanding the system-centralized view, software architecture reconstruction, holistic system assessments, etc.). With our approach, new advancements could emerge with tools operating on high-level system constructs in a platform-independent manner, possibly spanning decentralized and heterogeneous systems such as cloud-native systems. Such problems would be very difficult to solve using current static analysis practices.

In our future work, we aim at extending our previous work on software architecture reconstruction of cloud-native systems [36], and automated code-smell detection for the Java platform [37]. ReSSA will enable us to cope with heterogeneity and thus allow us to analyze heterogeneous cloud-native benchmarks. All of this contributes to our long-term goal of utilizing static analysis to handle architecture degradation [40].

Furthermore, many useful extensions could be made to ReSSA in the future:

- The LAAST navigation is currently a naive, comprehensive search. Indexing the tree to easily find nodes of a specified type in a subtree could potentially slash the runtime, especially for more complex ReSSAs. Similarly, applying search optimizations would likely create a useful performance enhancement.

[10] Available as an open-source project on GitHub at https://github.com/cloudhubs/source-code-parser

- Currently, only Java and C++ are supported. Adding more languages–like Javascript, Go, and Python–would be a major improvement. Such extensions are currently in development.
- Currently, ReSSA implements code that allows for regimented visitation of a LAAST, but does so in a hardcoded manner. Abstracting this to allow for tools similar to ReSSA, but using different internal logic, would broaden the applicability of LAAST and help other developers not need to reinvent the wheel with regards to this complex concept.
- The ReSSA context, in our first draft, only supports storing string objects. This limits what our current callbacks can implement. Enabling the context to store arbitrary data types would enable far more complex algorithms and useful analyses.
- Currently, we have only applied ReSSA to identify data entities and endpoints. However, there are many other relevant components we could target in microservice systems, such as Kafka connections, Java Native Interface calls, and more. Creating ReSSAs for these would reinforce ReSSA's general applicability, as well as enable far more complex analysis tools to easily scrape systems for more complex concepts.
- Performing downstream language-agnostic code analysis using ReSSA over heterogeneous systems by dynamically detecting languages and frameworks within the code to construct the appropriate ReSSAs automatically.

AUTHOR CONTRIBUTIONS

Micah Schiewe conceived the concept of ReSSA and led the design and implementation for the case study experiments. In particular, he performed the Java LAAST, ReSSA implementation. He analyzed and validated the study results, predominantly DeathStarBench, and participated in the TrainTicket study revision. In addition, he also authored the original and reviewed drafts of the manuscript.

Jacob Curtis conceived the concept of LAAST and led the design and implementation for the case study experiments, and led the benchmarking. In particular, he implemented LASST for C++ and co-implemented ReSSA; also, he analyzed and validated the study results, predominantly for DeathStarBench. In addition, he also authored the original and reviewed drafts of the manuscript.

Vincent Bushong designed the experiments performed on the TrainTicket benchmark and validated the results. He led ReSSA implementation for the TrainTicket and participated in LAAST implementation for Java. In addition, he authored and reviewed early drafts of the manuscript.

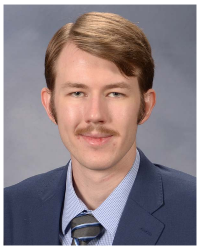
Tomas Cerny managed and supervised the methodology, conceptualization, design, and experiments. He coordinated the collaborative work, authored the original and reviewed manuscript drafts, and approved the final version. In addition, he was responsible for correspondence, administration, and funding acquisition.

All authors have read and agreed to the published version of the manuscript.

REFERENCES

[1] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 15, no. 3, pp. 182–211, 1976.
[2] H. Mumtaz, P. Singh, and K. Blincoe, "A systematic mapping study on architectural smells detection," *J. Syst. Softw.*, vol. 173, Mar. 2021, Art. no. 110885. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302752
[3] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: Current and future directions," *ACM SIGAPP Appl. Comput. Rev.*, vol. 17, no. 4, pp. 29–45, Jan. 2018, doi: 10.1145/3183628.3183631.
[4] Y. Gan, Y. Zhang, D. Cheng, and A. Shetty, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, New York, NY, USA, 2019, pp. 3–18, doi: 10.1145/3297858.3304013.
[5] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," in *Proc. 40th Int. Conf. Softw. Eng., Companion*, New York, NY, USA, May 2018, pp. 323–324, doi: 10.1145/3183440.3194991.
[6] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Industry practices and challenges for the evolvability assurance of microservices," *Empirical Softw. Eng.*, vol. 26, no. 5, p. 104, Sep. 2021, doi: 10.1007/s10664-021-09999-9.
[7] G. Mathew, C. Parnin, and K. T. Stolee, "SLACC: Simion-based language agnostic code clones," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, New York, NY, USA, Jun. 2020, pp. 210–221, doi: 10.1145/3377811.3380407.
[8] Z. Mushtaq, G. Rasool, and B. Shehzad, "Multilingual source code analysis: A systematic literature review," *IEEE Access*, vol. 5, pp. 11307–11336, 2017.
[9] *ISO/IEC/IEEE International Standard—Systems and Software Engineering–Vocabulary*, Standard ISO/IEC/IEEE 24765:2017(E), 2017, pp. 1–541.
[10] T. Cerny, J. Svacina, D. Das, V. Bushong, M. Bures, P. Tisnovsky, K. Frajtak, D. Shin, and J. Huang, "On code analysis opportunities and challenges for enterprise systems and microservices," *IEEE Access*, vol. 8, pp. 159449–159470, 2020.
[11] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proc. Int. Conf. Softw. Maintenance*, Nov. 1998, pp. 368–377.
[12] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann, "Language-agnostic representation learning of source code from structure and context," 2021, *arXiv:2103.11318*.
[13] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim.*, Mar. 2004, pp. 75–86.
[14] Y.-G. Gueheneuc, "Abstract and precise recovery of UML class diagram constituents," in *Proc. 20th IEEE Int. Conf. Softw. Maintenance*, Sep. 2004, p. 523.
[15] Y. G. Guéhéneuc, "A systematic study of UML class diagram constituents for their abstract and precise recovery," in *Proc. 11th Asia–Pacific Softw. Eng. Conf.*, Dec. 2004, pp. 265–274.
[16] A. Sutton and J. I. Maletic, "Recovering UML class models from C++: A detailed explanation," *Inf. Softw. Technol.*, vol. 49, no. 3, pp. 212–229, Mar. 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584906001844
[17] K. Satoh, K. Kaneiwa, and T. Uno, "Contradiction finding and minimal recovery for UML class diagrams," in *Proc. 21st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2006, pp. 277–280.
[18] G. Costagliola, A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery by visual language parsing," in *Proc. 9th Eur. Conf. Softw. Maintenance Reengineering*, 2005, pp. 102–111.
[19] M. Zabriskie. *Axios*. Accessed: Aug. 28, 2021. [Online]. Available: https://axios-http.com/
[20] J. M. P. Cardoso, T. Carvalho, J. G. F. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov, "LARA: An aspect-oriented programming language for embedded systems," in *Proc. 11th Annu. Int. Conf. Aspect-Oriented Softw. Develop. (AOSD)*, New York, NY, USA, 2012, pp. 179–190, doi: 10.1145/2162049.2162071.

[21] P. Pinto, T. Carvalho, J. Bispo, M. A. Ramalho, and J. M. P. Cardoso, "Aspect composition for multiple target languages using LARA," *Comput. Lang., Syst. Struct.*, vol. 53, pp. 1–26, Sep. 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S147784241730115X
[22] G. Rakić, Z. Budimac, and M. Savić, "Language independent framework for static code analysis," in *Proc. 6th Balkan Conf. Informat. (BCI)*, New York, NY, USA, 2013, pp. 236–243, doi: 10.1145/2490257.2490273.
[23] P. Klint, T. van der Storm, and J. Vinju, "RASCAL: A domain specific language for source code analysis and manipulation," in *Proc. 9th IEEE Int. Work. Conf. Source Code Anal. Manipulation*, Sep. 2009, pp. 168–177.
[24] E. Ntentos, U. Zdun, K. Plakidas, P. Genfer, S. Geiger, S. Meixner, and W. Hasselbring, "Detector-based component model abstraction for microservice-based systems," *Computing*, vol. 103, no. 11, pp. 2521–2551, Aug. 2021, doi: 10.1007/s00607-021-01002-z.
[25] *Swagger*. Accessed: Sep. 9, 2021. [Online]. Available: https://swagger.io/
[26] R. Kennard and J. Leaney, "Is there convergence in the field of UI generation?" *J. Syst. Softw.*, vol. 84, no. 12, pp. 2079–2087, Dec. 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121211001348
[27] T. Cerny, K. Cemus, M. J. Donahoo, and E. Song, "Aspect-driven, data-reflective and context-aware user interfaces design," *ACM SIGAPP Appl. Comput. Rev.*, vol. 13, no. 4, pp. 53–65, 2013.
[28] L. Ardito, L. Barbato, M. Castelluccio, R. Coppola, C. Denizet, S. Ledru, and M. Valsesia, "Rust-code-analysis: A rust library to analyze and extract maintainability information from source codes," *SoftwareX*, vol. 12, Jul. 2020, Art. no. 100635. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352711020303484
[29] Apache. *Apache Thrift Software Framework, for Scalable Cross-Language Services Development*. Accessed: Aug. 28, 2021. [Online]. Available: https://thrift.apache.org/
[30] gRPC Authors. *gRPC: A High Performance, Open Source Universal RPC Framework*. Accessed: Aug. 28, 2021. [Online]. Available: https://grpc.io/
[31] *Spring Framework*. Accessed: Aug. 28, 2021. [Online]. Available: https://spring.io/
[32] *Mongodb: Cloud Document Database*. Accessed: Aug. 28, 2021. [Online]. Available: https://www.mongodb.com/
[33] *Hibernate. Everything Data. - Hibernate*. Accessed: Aug. 28, 2021. [Online]. Available: https://hibernate.org/
[34] *Sqlalchemy—The Database Toolkit for Python*. Accessed: Aug. 28, 2021. [Online]. Available: https://www.sqlalchemy.org/
[35] *Prophet: Code Representation in Graph Database*. Accessed: Aug. 28, 2021. [Online]. Available: https://github.com/cloudhubs/prophet
[36] A. Walker, I. Laird, and T. Cerny, "On automatic software architecture reconstruction of microservice applications," *Information Science and Applications (ICISA)*, vol. 739. Singapore: Springer, 2021, pp. 223–234. [Online]. Available: https://link.springer.com/chapter/10.1007/978-981-33-6385-4_21
[37] A. Walker, D. Das, and T. Cerny, "Automated code-smell detection in microservices through static analysis: A case study," *Appl. Sci.*, vol. 10, no. 21, p. 7800, Nov. 2020, doi: 10.3390/app10217800.
[38] D. Das, A. Walker, V. Bushong, J. Svacina, T. Cerny, and V. Matyas, "On automated rbac assessment by constructing centralized perspective for microservice mesh," *PeerJ Comput. Sci.*, vol. 7, p. e376, 2021.
[39] I. Pigazzini, F. A. Fontana, V. Lenarduzzi, and D. Taibi, "Towards microservice smells detection," in *Proc. 3rd Int. Conf. Tech. Debt*, Jun. 2020, pp. 92–97.
[40] V. Bushong, A. S. Abdelfattah, A. A. Maruf, D. Das, A. Lehman, E. Jaroszewski, M. Coffey, T. Cerny, K. Frajtak, P. Tisnovsky, and M. Bures, "On microservice analysis and architecture evolution: A systematic mapping study," *Appl. Sci.*, vol. 11, no. 17, p. 7856, Aug. 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/17/7856
[41] FudanSELab. (2021). *Train Ticket—A Benchmark Microservice System*. [Online]. Available: https://github.com/FudanSELab/train-ticket
[42] *LUA: About*. Accessed: Aug. 28, 2021. [Online]. Available: https://www.lua.org/about.html
[43] C. Walls, *Spring Action*. Greenwich, CT, USA: Manning, 2018.
[44] C. Wohlin, *Experimentation in Software Engineering: An Introduction* (International Series in Engineering and Computer Science). Norwell, MA, USA: Kluwer, 2000. [Online]. Available: https://books.google.com/books?id=nG2UShV0wAEC
[45] P. Ralph and E. Tempero, "Construct validity in software engineering research and software metrics," in *Proc. 22nd Int. Conf. Eval. Assessment Softw. Eng.*, New York, NY, USA, Jun. 2018, pp. 13–23, doi: 10.1145/3210459.3210461.



MICAH SCHIEWE is currently a Senior Student in computer science at Baylor University. Since fall 2018, he has been featured on the Baylor University Dean's List. He helped develop the ACM SAC 2021 online platform, which served an international audience with more than 300 participants. His research interest includes log and static code analysis. In addition, he received the UPE/ACM Student Chapter Scholarship Award in 2021.

JACOB CURTIS is currently pursuing the bachelor's degree in computer science with Baylor University. He has previously coauthored two publications related to both log analysis and static code analysis. His research interests include log analysis and static code analysis.

VINCENT BUSHONG received the bachelor's degree from Evangel University, Springfield, MO, USA, and the master's degree in computer science from Baylor University, Waco, TX, USA. He is currently a Software Engineer. His research interests include microservices and code analysis.

TOMAS CERNY received the master's and Ph.D. degrees from the Faculty of Electrical Engineering, Czech Technical University in Prague, and the M.S. degree from Baylor University.

In 2009, he started his academic career at the Czech Technical University, FEE, from where he transferred to Baylor University, in 2017. He is currently a Professor of computer science at Baylor University. He worked more than ten years as the Lead Developer of the International Collegiate Programming Contest Management System. He authored nearly 100 publications mostly related to code analysis and aspect-oriented programming. His research interests include software engineering, code analysis, security, aspect-oriented programming, user interface engineering, and enterprise application design. Among his awards is the Outstanding Service Award ACM SIGAPP 2018 and 2015 or the 2011 ICPC Joseph S. DeBlasi Outstanding Contribution Award. He served on the committee of multiple conferences in the past few years, including ACM SAC, ACM RACS, and ICITCS.