Communication Papers of the 20th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 127–134
DOI: 10.15439/2025F5264
ISSN 2300-5963 ACSIS, Vol. 45

Static components dependency graph detection with evaluation metrics in React.js projects

Łukasz Kurant
Department of Cyber Security and Computational Linguistics
University of Maria Curie-Sklodowska
Pl. M. Curie-Skłodowskiej 5, 20-031 Lublin, Poland
Email: lukasz.kurant@mail.umcs.pl

***Abstract*—The popularity of libraries and frameworks for JavaScript and Typescript introduces completely new problems and tasks that can be solved using code analysis. Static type of this process has a plenty of applications, and despite of dynamic or hybrid methods, it has the significant advantages of simplicity, high performance and does not require a list of tests to work properly. One of the frameworks for the mentioned languages is React.js, which introduces a component-based architecture that allows the creation of isolated parts of the user interface in the form of functions or classes that meet specific requirements. In this paper, we describe an algorithm we have developed to detect relationships between components and create a dependency graph. Its performance was validated by comparison with a manually created graph, achieving an average F1 value of 0,95. We also conducted a performance analysis of the proposed solution. In order to correctly assess the impact of a component on the rest of the system both locally and globally, we have introduced five component evaluation metrics that provide important information when designing and changing the architecture of a front-end application. The developed algorithm and metrics can be useful tools for software architects and engineers, providing information about design interdependencies and the influence of individual components on parts of the system.**

I. Introduction

JAVASCRIPT is currently one of the most popular programming languages [1]. Its popularity is due to its usability and cross-platform nature – code written in JavaScript can be run on a variety of devices including servers, rather than exclusively in browsers as in the past. The popularity of front-end frameworks and libraries such as [2], [3] or back-end frameworks such as [4] enables the code to be synchronized and easier to understand among software development teams. Also, the popularity of running code on different platforms, e.g. using frameworks such as [5], allows software production costs to be optimized, making it a frequent choice not only for smaller companies, but also for large corporations.

Because of the nature of the JavaScript language (its mechanisms that differ from most common languages and its memory management system, i.e. weak and dynamic typing), the community has led to the creation of a number of languages that are a superset of the language. An example is Typescript, which has seen a huge surge in popularity in recent years [1], or other languages compiled into JavaScript like CoffeeScript.

Static code analysis is challenging due to problems caused by dynamic types or asynchronous mechanisms, which only affect the real values in memory when the code is running. Among the purposes of such analysis, we can mention the detection of defects in code [6], automatic refactoring [7] or the detection of security threats [8]. Among the tasks that are useful to carry out such an analysis is the construction of a *call graph*, which describes the connections between different functions in a program. While the construction of such structure in the case of strongly typed languages such as Java is quite standardized and studied (due to the ease of analyzing the inheritance chains of individual classes), in the case of JavaScript [9], due to prototypical inheritance and lexical or dynamic scope of visibility (depending on the context), it is significantly difficult to construct such a graph in a static way [10]. Alternatives may be to use dynamic construction of such a graph or hybrid methods [6], which requires running the code and performing in-memory address analysis, which is sometimes difficult due to the need to build, for example, tests that will offer high code coverage.

All of these issues also lead to problems with event-based flow, i.e. HTML Document Object Model (*DOM*) operations often require the use of event listeners on DOM tree nodes that are reflected in the HTML document in the browser. Thanks to the use of frameworks or libraries, developers are able to create more clear code, and interfere more easily with the DOM tree. The main concept behind the React library is use of *components*, i.e. functions or classes that follow a certain life cycle and can be used to generate a node in the DOM tree. Components are independent parts of code that represent a way to encapsulate client-side / UI-related logic, i.e. they extract part of the code, but they work in an isolated way and have to return code that enables the generation (called *rendering*) of a certain part of the user interface. When we use components, we work on them independently, and then we can use them to create a more complex component, up to a parent component that contains the whole user interface.

The appropriate design of such components therefore has an impact on the performance and scalability of the entire system, making it necessary to skilfully design the entire architecture when working with them. However, due to the above properties of the JavaScript language, this can often be a difficult process, because of the possibility of dynamically changing the location in memory of such a component definition or the ambiguity of certain component names or properties. Hence, there is

a need to define the relationships between components in such a way as to assess how they affect the rest of the system and how potential changes carried out by the programmer will have unintended consequences. A detailed description of the applications and advantages of having component dependency graph information is presented in the next chapter.

The objective of this article is to create a tool to statically detect components in JavaScript and TypeScript code, thereby creating a dependency graph between the components themselves. With the knowledge of such relationships, we are able to introduce metrics for evaluating a component in terms of its impact on other components. The detection algorithm itself is based on a static analysis of the Abstract Syntax Tree *AST*. The use of such a structure, due to the information on the structure of the code, makes it easier to find the parts of the code that allow components to be identified. However, by also creating a plug-in for the IDE, we have the possibility of graphically representing a related group of components, which has a significant impact on the work of developers and software architects.

II. Motivation

When developing modern web or mobile projects in React, especially large enterprise projects, understanding the structure and interconnectedness of components becomes one of the critical challenges that directly affects the work of developers and software architects. As applications evolve, developers often lose a complete overview of the dependencies between components. Tools that offer component analysis of such applications allow proactive detection of potential architecture issues before they become costly to fix, enabling better planning of refactorings and code upgrades. In [11] the authors analysed data from 43 developers showing that a significant proportion of their *wasted* working time is spent managing technical debt, and that the prevention of technical debt has a direct impact on their morale. One method of preventing such debt is refactoring and risk and impact analysis.

In [12], the positive impact of project technical documentation on the error rate of developers was demonstrated. It is, therefore, an important task to create technical project documentation, and the use of any tools to facilitate this process can significantly improve the process. Knowledge from such documentation can be used by developers and architects to identify sensitive parts of the system and make informed decisions about breaking down or combining components at the design stage, or evaluating the solution at a later stage, e.g. during code review.

III. Definitions

A basic structure, commonly used by compilers and interpreters and therefore having a strong influence on their operation, is the Abstract Syntax Tree (*AST*), a data structure representing the abstract structure of source code written in a formal language, resulting from syntactic analysis of the text. Each node of this tree represents a selected language construct, and its descendants the components of such construct. Unlike the language code itself, such trees do not contain less important parts such as punctuation or delimiters. However, they very often contain information about the position of each element in the code, which has a positive effect on the work of the compiler by allowing useful error messages to be output [13].

A. Components

In the React library, a *component* is a function or class that contains some part of code, and which returns some user interface element. In JavaScript, classes are purely so-called syntax sugar and are an overlay that works with prototypes and functions, making it easier for developers. So we can reduce a component K purely to a certain function $K(P) \rightarrow X$, where P is a non-mutable set of component properties (also abbreviated as *props*) and the return type X is a certain interface element. The purpose of such a component is to allow the simple creation of some reusable element that will be used to render a node in the DOM tree.

To create a component, we can use the *JSX* syntax, which is an extension of the JavaScript syntax with the ability to insert markup code (this is the solution recommended by the React library creators, although not the only one). JSX resembles a template-based language, but it provides the full capabilities of JavaScript itself. An example component is shown in Listing 1. This component is called *Main* and returns some JSX code, using a dependency of another component. As components can refer to other components when returning a result, this allows the same component abstraction to be used at any level of detail. Any component that has been rendered is subject to certain component lifecycle mechanisms, i.e. we have the possibility to detect and react to situations occurring in the component, such as the moment after it has been mounted (rendered), updated or before it has been unmounted.

B. Component dependency relationship

If the rendering of component K_1 leads to the rendering with its use in the DOM tree of component K_2 (it is not a matter of importing a function of the component or using it in another context), then we can define that $K_1 \leftarrow K_2$, and that means there is a relationship in which K_1 is the ancestor of K_2. Let us call such a relationship as a *component dependency relationship*. K_2 can be rendered independently, but rendering K_1 in selected cases will lead to K_2 being rendered. Whether or not a component is rendered depends

```
1 import React from 'react';
2 import ChildComponent from './ChildComponent';
3
4 function Main(props) {
5   return (
6     <div>
7       <ChildComponent />
8       <span>{props.name}</span>
9     </div>
10  );
11 }
```

Listing 1. Code for a sample component using JSX syntax

```
import React from 'react';
import ChildComponent from './ChildComponent';

function Main(props) {
  if(props.shouldChildBeRendered) {
    return <ChildComponent />
  }

  return <div />
}
```

Listing 2. Component code with rendering condition

on the logic in ancestor itself. Listing 2 shows an example – component K_2 will only be rendered if the value passed to the component is true, but the usage relation is still fulfilled.

C. Components dependency relations graph (CDG)

A component graph is a directed graph $G = (V, E)$ that represents a usage relationship between components, where V is the set of vertices representing the components, e.g. $V = \{C_1, C_2, ..., C_n\}$ and $E \subseteq V \times V$ is the set of edges representing the relationships between the components. The edge (C_i, C_j) E exists if and only if component C_i imports and uses component C_j in its rendering structure. Any number of edges can come out of each vertex, which symbolises the connection of a component to another by being able to render it when rendering its ancestor. An example of a graphical representation of such a graph is shown in Figure 1.

In the case of projects that are a collection of independent components (such as UI frameworks), very often such a graph will consist of independent subgraphs not connected by any usage relationship. It is also possible for cycles to occur in specific cases – sometimes components have the ability to render themselves or other components using that component. Of course, cyclicity is not mentioned on the DOM tree, but in the definition of the component function itself, under appropriate conditions, such a situation can already occur.

IV. Approach

The basis of our proposed algorithm is an AST tree, which we have used the *@typescript-eslint/typescript-estree* library to generate. Going through the selected nodes of such a tree for a given file, we have the possibility to search and mark certain elements, which will be used to detect components and create relations between them. Once an AST tree has been prepared for a selected file (each such file is analysed only once), the algorithm proceeds to analyse the code present in such a file, detecting those elements that serve to identify the component code. The algorithm then proceeds to analyse the code of potential components detecting instances of other components based on JSX tags. The final step is to go to files that contain

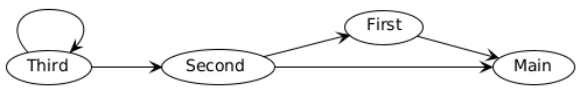

Fig. 1. Example graph of components dependency relations

definitions of components whose use has been detected, but whose definitions are missing from the file currently being analysed.

A. AST tree analysis process

The main purpose of the AST tree analysis is to search selected important structures from a problem-solving perspective. Among these, we can specify the analysis of imports, exports, functions, classes, expressions and variable declarations.

1) Imports: The algorithm processes all imported elements into the file. An important element of such an analysis is the support of both default and detailed imports, in addition to the possibility of adding aliases. It is also important to provide support for so-called *path aliases* provided by plug-ins for the *Babel* (the JavaScript compiler to its other standards), which are very common in projects to ensure code readability. Instead of using a very deep relative path, we can create an alias to a specific directory, which will be replaced at the compilation stage by Babel.

2) Functions and classes: Any file-level function can be treated as a potential component – it all depends on whether it contains key elements about it (JSX tag syntax). In some cases, anonymous functions or function expressions, such analysis is more difficult due to the heavier linking to the label and to another component. The same is true for classes. As the use of classes is currently less common in the React library, however, it is easier to detect that a class is a component due to the need to inherit from a *ReactComponent* or *ReactPureComponent* classes.

3) Expressions and variable declarations: In the case of expression analysis, it is not necessary to process every expression that is available in JavaScript or Typescript. However, many of them, such as function calls, conditional statements, loops, object expressions and others must be analysed. Among these are also markup expressions from JSX syntax, which may (but need not) indicate that the function or class they are in is actually used as a component. In addition, since certain expressions including function expressions may be assigned to variables, such analysis is also necessary.

4) Export expressions: As with import expressions, we need to know which potential functions and classes are available externally in other files and how we can link them on the component dependency graph. To this end, it is important to detect such declarations and link them to the file in which they are located.

B. Component detection

Detecting a potential descendant of a component, is notable in that, in order to add such a connection, we need to check whether the detected component in JSX syntax actually exists at file level. To do this, the node in which the component (parent) is located is first searched for elements whose name (or alias) overlaps with the component used. When it is not found, the other children of the file node are then checked, and finally the imported elements. A detailed implementation

```
Main (./Main.tsx)
    <- Common1 (./Common1.tsx)
Main (./Main.tsx)
    <- LocalComponent (./Main.tsx)
Common1 (./Common1.tsx)
    <- Common2 (./Common2.tsx)
```

Listing 3. Component code with rendering condition

of the algorithm has been made publicly available by us [14] and can be used to verify our results.

C. Mode of operation

The algorithm operates in two modes: file and directory level. In the first, we start the analysis from a specific file, moving on to other files when necessary, i.e. when an imported element has been marked as a component. The choice of start file depends on the user's own choice of where to start the analysis. The second type detects all files in a given directory and subdirectories and performs a full analysis of all such files – this means that we can find such dependency graphs that are not related to each other. The choice of starting point is irrelevant here, as the algorithm will go through all the files in the selected directory anyway.

The result of the algorithm is a set of relations in a format resembling the DOT format, but also allowing the identification of nodes with the same names by additionally adding information about the file in which the component is located. An example of such a result is available in Listing 3.

V. Metrics

In the React library, the creation of versatile, scalable and reusable components is an important part of the developers' work and affects the entire application design. Each component should therefore follow certain rules and implement good practices of code writing. In order to make it possible to evaluate such portions of code in terms of their impact on parts of the system, we propose metrics that allow us to evaluate such parts of the system, based on the component dependency graph. Such information is extremely useful for the design of the architecture of the whole application and allows us to find fragile of the systems, the modification of which may cause unforeseen effects in different parts of the application, often separated from each other.

A. Component evaluation measures

For any component K, which is a node V_K in the component dependency graph, we can introduce the following metrics to assess its dependency:

1) **Local component complexity,** $LLC(K)$ – We can define as the number of external components that have been used directly in the K component, which is the same as the number of edges starting from the V_K vertex of the graph.
2) **Cluster component complexity,** $CCC(K)$ – We can define as the number of independent components that will be rendered when component K is rendered, the same as the number of vertices we can reach in the subgraph starting from vertex V_K.
3) **Local component dependence,** $LCD(K)$ – The number of external components that use the K component directly, the same as the number of edges entering the V_K vertex.
4) **Cluster component dependence,** $CCD(K)$ – The number of external components that use the component directly or indirectly, which is the same as the number of vertices from which we can go to vertex V_K in the dependency graph.
5) **Component dependency cyclicity,** $CDC(K)$ – The length of the smallest cycle in the graph of components from vertex V_K. For example, a value of 0, is a component that never renders itself when rendering, 1 when the component directly renders itself, 2 when another component whose component K uses renders it, etc.

Each of the metrics given will be used to individually assess the impact of the component on the system.

B. Results evaluation measures

Among the experiment proposed next, in order to compare the performance of the algorithm with a dataset prepared manually, let's also introduce standard metrics for evaluating the results:

1) **Precision** – Ratio of correctly classified elements to all positively classified elements:
$$Precision = \frac{TP}{TP + FP}$$
where TP means True Positive samples, while FP means False Positives.
2) **Recall** – Ratio of elements correctly recognized to all that should be classified as correct:
$$Recall = \frac{TP}{TP + FN}$$
where FN means False Negatives samples.
3) **F1** – The harmonic mean of precision and recall, expressed by the formula:
$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

VI. Experiments

To test the functioning of the algorithm, we prepared a set of experiments comparing the result obtained by manual code review to the results returned by the algorithm. Ten different open source projects from Github with different dependencies were used for this purpose, both in terms of language (JavaScript or TypeScript) and purpose (Web or Mobile). A detailed list of projects and information about them is presented in Table I. The projects have been chosen to provide real examples of projects developed in industry, incorporating different versions of the React or React Native library.Note that the Number of JS/TS Files (NoF) or number of associated Lines of Code (LoC) is not correlated with the

TABLE I
LIST OF PROJECTS USED TO TEST THE ALGORITHM

Name	Link	Description	Libs versions	Stars	NoF	LoC
Ant design	https://github.com/ant-design/ant-design	component library	React 18.3.1	91.2k	3589	215020
Prismane	https://github.com/prismaneui/prismane	component library	React 18.2.0	335	603	20984
Whisper client	https://github.com/Dun-sin/Whisper/	chat app	React 18.2.0	354	60	11340
Noteslify (web)	https://github.com/bytemakers/Noteslify	digital note app	React 17.0.1 + React Native 0.64.3	125	120	56932
DeveloperFolio	https://github.com/saadpasta/developerFolio	portfolio template	React 16.10.2	5.2k	105	23415
React-play	https://github.com/reactplay/react-play	learning to program	React 18.2.0	1.3k	909	78208
Feelio	https://github.com/baqx/feelio/	digital diary	React 18.2.0 + React Native 0.74.3	32	48	17614
Linky	https://github.com/kwsong0113/imagine	gesture-based launcher	React 18.2.0 + React Native 0.71.4	165	186	10647
Peyara mouse	https://github.com/ayonshafiul/peyara-mouse-client	remote mouse	React 18.2.0 + React Native 0.74.3	6	83	3756
Chain React	https://github.com/infinitered/ChainReactApp2023	event app	React 18.2.0	108	126	11729

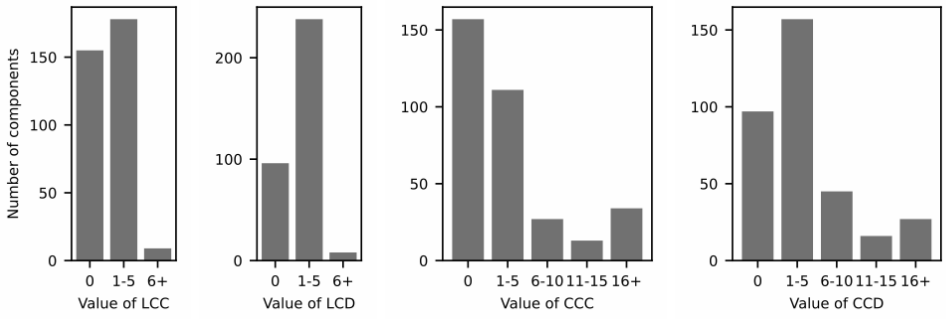

Fig. 2. Number of components in *ant-design* based on groups of metric values

number of components, as there may also be other files in the project, such as documentation or tests.

The projects were marked up by us manually and then a verification was performed. The data prepared in this way was saved in exactly the same format as the returned algorithm format described earlier. The algorithm was then run on the projects in directory mode and the results were compared with those done manually.

A. Performance measurement

In parallel, in order to measure the memory usage and execution time of the algorithm, three different designs were generated that contain a medium, large and very large number of components, so as to investigate how the resource requirements will increase as a function of the number of components, which are the same as the number of files (each file contains only one component, using a random number of other components). Details of the designs are described in Table III. The tools were run on a MacBook Pro with an 8-core Intel Core i9 processor (2.3GHz) and 16GB RAM. To measure the execution time of the algorithm, we used the recommended *performance.now()* function, and to measure the memory usage *process.memoryUsage()*, which allows us to not only examine the amount of memory used by the execution of the process, but also the memory heap and external memory usage of the JavaScript engine.

B. Results

The results of the experiments, together with the values of the main metrics for the open source projects, are presented in Table IV. The first columns compare the values of the number of manually marked components (MC) to the number detected by the algorithm (AC). As we can see, the precision (P) is in most cases always equal to 1, but the recall (R) counts drop in a size that depends on the selected project (thus also affecting the F1 metric). This is important as it indicates problems with the capture of certain groups of relations by the algorithm itself (described below). As we can see, the precision and recall results are very good, and most of the relationships from the manual tagging were included in the algorithm results.

Additional results for component measurement metrics are shown in Table II. These results were divided for each metric into three values: the arithmetic mean of the metric (avg), the median (med) and the maximum value (max). As can be seen,

TABLE II
RESULTS OF COMPONENTS METRICS FOR PROJECTS

Name	LLC			CCC			LCD			CCD			CDC		
	avg	**med**	**max**	**avg**	**med**	**max**	**avg**	**med**	**max**	**avg**	**med**	**max**	**avg**	**med**	**max**
ant-design	1.25	1	13	4.52	1	51	1.25	1	13	4.52	1	47	0.02	0	2
prismane	1.39	1	7	3.41	2	18	1.39	0	42	3.41	0	108	0.00	0	0
whisper	1.14	0	13	3.54	0	56	1.14	1	3	3.54	3	7	0.00	0	0
noteslify	1.84	1	17	2.61	1	35	1.84	1	9	2.61	1	17	0.00	0	0
developer-folio	1.54	1	18	3.44	1	38	1.54	1	14	3.44	2	18	0.00	0	0
react-play	1.31	0	19	4.15	0	132	1.31	1	11	4.15	4	19	0.00	0	0
feelio	2.53	2	11	2.86	2	13	2.53	1	19	2.86	1	19	0.00	0	0
linky	3.16	2	15	7.74	3	37	3.16	1	31	7.74	2	52	0.01	0	1
peyara-mouse	1.89	1	12	2.15	1	17	1.89	1	19	2.15	1	19	0.00	0	0
chain-react	2.40	2	7	6.37	4	28	2.40	1	38	6.37	2	54	0.02	0	1

TABLE III
LIST OF GENERATED PROJECTS

Name	**NoF**	**LoC**
medium-sized-project	181	3003
large-sized-project	431	7916
extra-large-sized-project	1141	26885

in special cases a component can be related to up to dozens of other components. Also worth adding is the value of the CDC metric, which only in two projects has a value greater than 0. From this it follows that cycles at the component level are very rare. Also an important finding is that the average number of components used directly or indirectly by a component (CCC metric) is equal to the average number of components that have a relationship with one of their ancestors (CCD metric) – this follows directly from the graph structure itself.

TABLE IV
RESULTS OF KEY METRICS FOR PROJECTS

Name	**MC**	**AC**	**P**	**R**	**F1**
ant-design	470	427	0.944	0.886	0.914
prismane	228	188	1.000	0.825	0.904
whisper	76	65	1.000	0.878	0.935
noteslify	71	70	1.000	0.986	0.993
developer-folio	65	63	1.000	0.969	0.984
react-play	204	197	1.000	0.975	0.987
feelio	100	91	1.000	0.910	0.953
linky	335	307	1.000	0.927	0.962
peyara-mouse-client	148	123	1.000	0.837	0.911
chain-react-app	231	197	1.000	0.864	0.927
Average	**193**	**173**	**0.994**	**0.906**	**0.947**

In addition, we carried out a detailed analysis for the largest library *ant-design* used. The graphs in Figure 2 show the number of components in groups for the LLC, CCC, LCD, CCD metrics. As we can see, the largest group are components having 0-5 descendants, and components having 16 and more are a smaller percentage. Such components are much more complex and have higher dependencies so changing them in the future may cause more problems – this is important information on which components or parts of the system should receive more attention in regression testing.

For performance tests, the results look as in Table V. To describe memory consumption, the following four measures were introduced: resident set size (RSS) – the total memory allocated to process execution; total allocated heap size (SAH); actual memory used during execution (AME) and V8 external memory (EM) - the memory used by the JavaScript Engine. As can be seen, the algorithm works efficiently even with large projects. This is a definite advantage of static solutions, as it allows real-time monitoring of changes in the component relationship graph even during the code development process. The memory consumption is also not excessive, as the main purpose of the memory is to keep the graph modelling structures on the heap. Loading the files themselves, once the analysis is complete, is not necessary and the resources reserved for processing them can be released.

VII. LIMITATIONS

Based on the results presented, we can conclude that the algorithm is performing well enough. However, it encountered some problems in its effects, resulting in an inability to recognise the correct relationship between components.

1) Assigning a component to another memory location: The biggest problem with component detection is assigning a definition to a different location in memory, whether using a variable or an object. For example, if we have code that looks like below. This is a rather simplified example, but it is nevertheless very difficult to statically check what *New-*

TABLE V
PERFORMANCE TEST RESULTS

Name	Time (ms)	RSS	SAH	AME	EM
medium-sized-project	241	122.14 MB	73.69 MB	47.64 MB	1.92 MB
large-sized-project	456	137.46 MB	82.94 MB	56.44 MB	1.94 MB
extra-large-sized-project	740	174.04 MB	119.69 MB	88.42 MB	1.95 MB

NameOfComponent actually points to in memory, due to the dynamic properties of JavaScript.

```
const NewNameOfComponent = OldComponent;
const Component = () =>
    <NewNameOfComponent />;
```

Listing 4. Example of different memory location

2) Compbound components: A very popular design pattern used in the design of component architectures is the so-called *compbound components*. Using this design pattern, we create a single shared state that is made available to all components that require it in order to work together to achieve a specific result. Since in JavaScript a function is also an object, we have the possibility of assigning another component to a selected field, which also makes it difficult to find the right link.

```
const Select = () => <div />;
const Option = () => <span />;
Select.Option = Option;

const Component = () => (
  <Select>
    <Select.Option />
  </Select>
);
```

Listing 5. Example of compbound component pattern

3) Parts of the code rendered natively: In the case of React Native applications, components are transformed to their native counterparts, so that the use of certain native components can result in a lack of relationship detection if the rendering process is behind a JavaScript thread. This is particularly evident when creating navigation using the *react-navigation* library.

4) Component factories: Another quite common pattern used in React.js is the factory, allowing a component to be built based on an additional function that returns a component definition. In this case, it is difficult to define such a link between components if its definition is somewhere deeper in the code block.

```
const componentFactory(params) = () => {
  // ...
  function newComponent(props) {
    return <div />;
  }
  // ...
  return newComponent;
}
```

Listing 6. Example of component factory

5) Components created without JSX: Since JSX is currently the most popular solution for creating projects, we omitted from the algorithm support for creating components using the createElement function built into React.js. This is a very rare solution, used only in special cases in commercial projects.

6) Other import mechanisms: JavaScript prior to ES6 modules using the import keyword, made it possible to create modules and import them in other ways, such as using the *require* keyword, which is now widely used in Node.js libraries. However, due to React.js, and the practical lack of use of such a method in projects, we skipped support for this type of syntax.

VIII. RELATED RESEARCH

In the literature, we can find many examples of the use of static code analysis to detect various elements in JavaScript and TypeScript code, but due to the specific nature of the language and its dynamic behaviour, they differ significantly from examples for other languages, especially strongly typed languages such as Java or C++. Among the many applications of such analysis, we can mention the detection of bugs [6], dead code [15] (code not used in the project) or security vulnerabilities [16]. Often, in combination with dynamic methods, they give significantly better results [6]. Due to the large number of libraries and frameworks for JavaScript, there are many problems that need to be solved.

Among the problems that still have not been fully solved is, for example, call graph detection, i.e. the creation of relations between functions - due to the nature of the language, this is still a very difficult process, for which static [17], dynamic, hybrid [6] and even machine learning methods are used [18]. As JavaScript often works with other technologies, it is also a challenge to create multilingual links between programs using other solutions [19].

Missing from these problems, there are considerations for building component graphs and analysing their results. This is important because React.js is currently the most popular front-end library [20], so optimising the architecture process is an important and highly relevant task for later development. Among the solutions to this problem, one [21] library can currently be found, but it is not supported and offers a limited ability to build links based on a single file only. It also lacks support for class components and other expressions, making it difficult to build a sufficiently accurate relationship graph between components. Thus, it is not sufficient to take a holistic view of the architecture of the entire application. In [22], the authors have proposed a Component Graph (CoG), which

allows the creation of a graph of the data flow in a React component, but it is a graph that shows the processes in a single component based on the component's life cycle, rather than the relations between components.

IX. Future work

Potential further developments include the use of dynamic or hybrid (static together with dynamic) methods to detect relationships between components. This has the potential to partially solve the problems mentioned in the previous sections. Another potential tool to verify in the future could be the use of [23] to extract relevant information from the component code and compare it with the solution used. Another direction is the combination of methods to detect component usage between different technologies, e.g. React Native allows components to be rendered on the native side, making the detection of connections between JavaScript and Typescript code and native code, for example in Java and Kotlin on Android or Objective C and Swift on iOS, also a very challenging task. When developing code, tools such as [24] are often used to dynamically check the relationships between components in the component tree, but this does not give full information about the conditional relationships that we can learn about when statically analysing the code, but using this method in practice could also be a good direction for research.

X. Conclusions

In this article, we presented a method for component detection using a proprietary algorithm analysing JavaScript and TypeScript code to detect potential component candidates and then marking connections between them based on an AST tree analysis. We compared the results of the algorithm with the analysis performed manually by a human. In addition, we introduced metrics for assessing component complexity and dependencies, thus introducing the possibility of evaluating a component in terms of its impact on other parts of the system. The code of our algorithm is available on a public repository [14]. In addition, in order to verify its use in practical applications, we have created a plug-in for Visual Studio Code [25], which allows simple use of the program for practical purposes.

References

[1] "Octoverse: The top programming languages: https://octoverse.github.com/2022/top-programming-languages," 2023. [Online]. Available: https://octoverse.github.com/2022/top-programming-languages

[2] "React, the library for web and native user interfaces: https://react.dev." [Online]. Available: https://react.dev

[3] "Angular framework website: https://angular.dev." [Online]. Available: https://angular.dev

[4] "Node.js website." [Online]. Available: https://nodejs.org/en

[5] "React native website: https://reactnative.dev." [Online]. Available: https://reactnative.dev

[6] G. Antal, Z. Tóth, P. Hegedűs, and R. Ferenc, "Enhanced bug prediction in javascript programs with hybrid call-graph based invocation metrics," 2024. [Online]. Available: https://arxiv.org/abs/2405.07244

[7] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip, "Tool-supported refactoring for javascript," *SIGPLAN Not.*, vol. 46, no. 10, p. 119–138, oct 2011. [Online]. Available: https://doi.org/10.1145/2076021.2048078

[8] V. Haratian, P. Derakhshanfar, V. Kovalenko, and E. Tüzün, "Refexpo: Unveiling software project structures through advanced dependency graph extraction," 2024. [Online]. Available: https://arxiv.org/abs/2407.02620

[9] M. Chakraborty, R. Olivares, M. Sridharan, and B. Hassanshahi, "Automatic root cause quantification for missing edges in javascript call graphs (extended version)," 2022. [Online]. Available: https://arxiv.org/abs/2205.06780

[10] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for javascript ide services," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 752–761.

[11] T. Besker, H. Ghanbari, A. Martini, and J. Bosch, "The influence of technical debt on software developer morale," *Journal of Systems and Software*, vol. 167, p. 110586, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220300674

[12] D. Che, "Automatic documentation generation from source code," Ph.D. dissertation, 01 2016.

[13] J. Jones, "Abstract syntax tree implementation idioms," *Pattern Languages of Program Design*, 2003, proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003) http://hillside.net/plop/plop2003/papers.html. [Online]. Available: http://hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf

[14] L. Kurant, "Component dependency graph." [Online]. Available: https://github.com/lukaszkurantdev/components-dependency-graph

[15] I. Malavolta, K. Nirghin, G. L. Scoccia, S. Romano, S. Lombardi, G. Scanniello, and P. Lago, "Javascript dead code identification, elimination, and empirical assessment," *IEEE Transactions on Software Engineering*, vol. 49, no. 7, pp. 3692–3714, 2023.

[16] A. Møller and M. Schwarz, "Automated detection of client-state manipulation vulnerabilities," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 749–759.

[17] G. Antal, P. Hegedűs, Z. Tóth, R. Ferenc, and T. Gyimóthy, "Static javascript call graphs: A comparative study," 2024. [Online]. Available: https://arxiv.org/abs/2405.07206

[18] A. M. Mir, M. Keshani, and S. Proksch, "On the effectiveness of machine learning-based call graph pruning: An empirical study," 2024. [Online]. Available: https://arxiv.org/abs/2402.07294

[19] A. M. Bogar, D. M. Lyons, and D. Baird, "Lightweight call-graph construction for multilingual software analysis," 2018. [Online]. Available: https://arxiv.org/abs/1808.01213

[20] "Developer ecosystem javascript survey," 2023. [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2023/javascript/

[21] "React component analyzer library: https://github.com/activeguild/react-component-analyzer." [Online]. Available: https://github.com/activeguild/react-component-analyzer

[22] Z. Guo, M. Kang, V. Venkatakrishnan, R. Gjomemo, and Y. Cao, "Reactappscan: Mining react application vulnerabilities via component graph," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 585–599. [Online]. Available: https://doi.org/10.1145/3658644.3670331

[23] "Codeql: https://codeql.github.com." [Online]. Available: https://codeql.github.com

[24] "React developer tools: https://react.dev/learn/react-developer-tools." [Online]. Available: https://react.dev/learn/react-developer-tools

[25] L. Kurant, "Component dependency graph vscode plugin." [Online]. Available: https://github.com/lukaszkurantdev/components-dependency-graph-vscode