Towards an Automated Approach to Detect Code Smells in ReScript

Maykon Nunes, Carla Bezerra, Emanuel Coutinho

Programa de Pós-Graduação em Computação (PCOMP)
Universidade Federal do Ceará (UFC) – Quixadá – CE – Brasil

`maykon.nunes@alu.ufc.br, carlailane@ufc.br, emanuel.coutinho@ufc.br`

Abstract. *Component-based development has become a dominant paradigm in front-end, with languages and frameworks evolving to support modular and reusable UI elements. While extensive research exists on detecting code smells in* JAVASCRIPT *frameworks like* REACT*, similar analyses for* RESCRIPT*, a statically typed functional language that compiles to JavaScript and integrates with* REACT*, remain unexplored. This work aims to fill this gap by presenting a prototype static analysis tool capable of detecting relevant code smells in* RESCRIPT *applications.*

1. Introduction

JavaScript's front-end ecosystem has grown extensively, with frameworks like REACT, ANGULAR, and VUE playing a central role in building modern web applications [Ferreira et al. 2021]. These frameworks promote a modular approach to interface design by breaking down the UI into reusable components, which serve as independent building blocks that can be combined to form complex user interfaces. However, despite these advantages, poor or suboptimal design decisions in component structuring and interaction can lead to significant degradation in code quality [Ferreira and Valente 2023].

Such suboptimal design decisions often give rise to code smells, which are indicators of potential issues in the code's structure that may hinder readability, maintainability, or scalability [Fowler 2018]. Therefore, identifying and addressing these smells becomes essential for preserving the overall quality of the application architecture. Despite prior research on code smells, most studies predominantly target typical design issues associated with the Java language [Zakeri-Nasrabadi et al. 2023].

Ferreira and Valente (2023) introduced a set of twelve REACT-specific code smells. They also proposed REACTSNIFFER, a static analysis tool capable of detecting these smells in REACT codebases. While this work offers valuable insights into quality issues in front-end frameworks, similar investigations for alternative front-end languages remain limited. One such language is RESCRIPT, a statically typed, functional programming language. Unlike JavaScript, which is dynamically typed, RESCRIPT enforces static typing and immutability by default, helping prevent common defects while maintaining seamless integration with the REACT ecosystem.

In this work, we adapt the detection of existing REACT smell analysis to RESCRIPT, enabling the identification of smells in the JavaScript code generated from RESCRIPT sources. Our approach can be applied during code reviews or continuous integration to automatically detect smells before they reach production, addressing a gap in current research and helping developers maintain code quality without manual inspection.

2. Study Design

In this section, we describe the methodology we use to support our research. To that end, we define an architecture that allows analyzing this output to identify patterns corresponding to known REACT-SPECIFIC code smells. The next subsection describe the detection strategy in detail.

2.1. Code smells detection architecture

Previous works detect code smells using static analysis [Fard and Mesbah 2013, Ferreira and Valente 2023]. These works rely on parsing the source code into an Abstract Syntax Tree (AST). Alternatively, some studies detect code smells by analyzing intermediate representations, such as Java bytecode [Walker et al. 2020], which allow detection based on symbolic or structural patterns present in the compiled code.

Our objective is to introduce an architecture for detecting code smells in RESCRIPT. Since the language offers utilities like `rescript-tools doc` that extract structural information from individual files but do not provide a complete or typed abstract syntax tree, our approach relies on analyzing the JavaScript code generated by the RESCRIPT compiler. The process begins by recursively scanning the project directory to locate all `.res` source files. After identification, these files are compiled into JAVASCRIPT using the RESCRIPT compiler. To guarantee expected compilation behavior, our method first verifies the presence of a valid configuration file, typically `bsconfig.json` or `rescript.json`.

An important aspect to highlight is that, if the configuration file is missing or lacks the necessary fields, our approach automatically inserts or updates the required entries to ensure a successful build process. These entries include the module format, the source directory, the output file suffix, and the inclusion of subdirectories. The Listing 1 summarizes the essential configuration used in our compilation setup.

```json
{
  "package-specs": {
    "module": "es6",
    "in-source": false
  },
  "suffix": ".bs.js"
}
```

Listing 1. Configuration file

After ensuring the project is correctly configured, the build process is triggered using the `rescript build` command, producing JavaScript files with the `.bs.js` extension. To analyze this compiled JAVASCRIPT, we employ the BABEL parser[1], which generates an Abstract Syntax Tree (AST) in JSON format. This AST is then analyzed by a set of smell detectors, each responsible for identifying a specific code smell. The system outputs a detailed report including the file paths and detected smells for each file. However, due to the lack of consistent source map support in the RESCRIPT compiler, it is currently not possible to pinpoint the exact line and column in the original `.res` source files where each smell occurs.

[1]https://babeljs.io/docs/babel-parser

3. Prototype tool

We implemented a command-line interface (CLI) tool that follows the architecture described in the previous section and made it available on GitHub[2]. For this study, we selected three REACT-specific code smells for detection: DIRECT DOM MANIPULATION, FORCE UPDATE, and TOO MANY PROPS. These smells were chosen because their structural characteristics remain semantically equivalent in both handwritten REACT code and the JavaScript generated by the RESCRIPT compiler. Listing 2 presents a RESCRIPT component that includes a code smell known as DIRECT DOM MANIPULATION. In this example, the component directly accesses and modifies the DOM using embed raw JAVASCRIPT.

```
module Form = {
  @react.component
  let make = () => {
    let showError = () => {
      // Direct DOM Manipulation
      let errorDiv = %raw(`document.getElementById('error')`)
      %raw(`errorDiv.textContent = 'Required field'`)
    }

    <div>
      <input type_="text" />
      <button onClick={_ => showError()}>{"Validate"->React.string}</button>
      <div id="error" />
    </div>
  }
}
```

Listing 2. ReScript component with Direct DOM Manipulation

After compilation, this code produces the JAVASCRIPT output shown in Listing 3. Unlike typical REACT projects written in JavaScript or TypeScript, which rely on JSX syntax, RESCRIPT emits plain JavaScript function calls to represent components. Although REACTSNIFFER was originally designed to analyze JavaScript with embedded JSX, our tool parses the compiled output using the BABEL parser, which generates a structured AST as illustrated in Listing 4. This AST is then processed by the smell detectors, which search for patterns to flag potential code smells.

```
function $Form(props) {
  var showError = function () {
    ((document.getElementById('error')));
    return (errorDiv.textContent = 'Required field');
  };
  // ...
}
```

Listing 3. ReScript compiled code

```
{
  "type": "File",
  "program": {
    "type": "Program",
    "sourceType": "module",
    "body": ["Node", "Node", "Node", "Node", "Node"],
    "directives": [],
  }
}
```

Listing 4. Compiled code structured AST

[2]https://github.com/maykongsn/resniff

To detect DIRECT DOM MANIPULATION, the detector traverses the AST in search of nodes that indicate direct interactions with the DOM. This includes the presence of global objects suach as `document` or `window`, method invocations like `getElementById`, as well as property assignments to attributes such as `textContent`. These patterns are recognized by analyzing specific node types, including `MemberExpression`, `CallExpression`, and `AssignmentExpression`, as in Listing 5.

```
const domPatterns = ["document", "window", "getElementById", /* ... */];
const smells = [];

const isDomPattern = (obj, prop) =>
  domPatterns.includes(obj) || domPatterns.includes(prop);

traverse(ast, {
  MemberExpression(path) {
    if (isDomPattern(path.node.object?.name, path.node.property?.name)) {
      smells.push(filePath);
    }
  },
  CallExpression(path) {
    if (isDomPattern(path.node.callee?.object?.name, path.node.callee?.property?.name))
         {
      smells.push(filePath);
    }
  },
  AssignmentExpression(path) {
    if (isDomPattern(null, path.node.left?.property?.name)) {
      smells.push(filePath);
    }
  }
});
```

Listing 5. Direct DOM Manipulation detection algorithm

The tool also identifies occurrences of the FORCE UPDATE smell, which refers to practices that override the reactive data-binding mechanisms typically supported by modern JAVASCRIPT frameworks. Specifically, REACT employs a one-way data binding model that automatically propagates state changes to the view layer. Therefore, forcing component updates or triggering page reloads to reflect changes is considered a poor practice. To detect this smell, the detector traverses the AST searching for method calls on global objects such as `window` or `location`, recognizing these patterns through the analysis of `CallExpression` and `AssignmentExpression` nodes (see Listing 6).

```
const forceUpdatePatterns = ["forceUpdate", "reload", "refresh", "href", "assign", "
    replace"];
const forceObjects = ["window", "location"];
const smells = [];

const isForceUpdatePattern = (obj, prop) =>
  forceObjects.includes(obj) || forceUpdatePatterns.includes(prop);

const isLocationHref = (obj, prop) =>
  obj === "location" && prop === "href";

traverse(ast, {
  CallExpression(path) {
    const { callee } = path.node;
    if (callee.type === "MemberExpression" &&
        isForceUpdatePattern(callee.object?.name, callee.property?.name)) {
      smells.push(filePath);
    }
  },
  AssignmentExpression(path) {
    const { left } = path.node;
    if (left.type === "MemberExpression" &&
        isLocationHref(left.object?.name, left.property?.name)) {
      smells.push(filePath);
    }
  }
});
```

Listing 6. Force Update detection algorithm

Another smell addressed by our tool is Too Many Props, which refers to components receiving an excessively large number of props. This condition may indicate violations of the Single Responsibility Principle. Overloaded components tend to be tightly coupled with their context and often encapsulate more logic than necessary. To detect this smell, the detector analyzes the AST by traversing `FunctionDeclaration` nodes and collecting each distinct property accessed from the `props` object via `MemberExpression` nodes. If the number of unique properties exceeds a predefined threshold, the component is flagged as smelly, as illustrated in Listing 7.

```
const threshold = 13;

const collectUniqueProps = (path) =>
  const uniqueProps = new Set();
  path.traverse({
    MemberExpression(memberPath) {
      const { object, property } = memberPath.node;
      if (object.type === "Identifier" && object.name === "props" &&
          property.type === "Identifier") {
        uniqueProps.add(property.name);
      }
    }
  });
  return uniqueProps;

traverse(ast, {
  FunctionDeclaration(path) {
    if (collectUniqueProps(path).size > threshold) {
      return { path: filePath };
    }
  }
});
```

Listing 7. Too Many Props detection algorithm

4. Conclusion and future work

This work presented a CLI prototype that detects code smells in ReScript applications by analyzing their compiled JavaScript output. Following a structured architecture, the tool automates the verification and adjustment of project configurations, the compilation process, parses the resulting code into an AST, and applies a set of smell detectors. However, it also faces challenges such as the lack of reliable source maps, which limits precise mapping of detected smells to original source locations in output.

As future work, we plan to extend the set of supported smells by including general-purpose code smells common in component-based front-end frameworks and to explore techniques for correlating compiled JavaScript with ReScript source locations. Moreover, we intend to conduct a comprehensive evaluation of the detection algorithms to assess the effectiveness and precision through empirical studies on real-world ReScript projects.

References

Fard, A. M. and Mesbah, A. (2013). Jsnose: Detecting javascript code smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125.

Ferreira, F., Borges, H., and Valente, M. (2021). On the (un-)adoption of javascript front-end frameworks. *Software: Practice and Experience*, 52:947–966.

Ferreira, F. and Valente, M. T. (2023). Detecting code smells in react-based web apps. *Information and Software Technology*, 155:107111.

Fowler, M. (2018). *Refactoring*. Addison-Wesley Professional.

Walker, A., Das, D., and Cerný, T. (2020). Automated code-smell detection in microservices through static analysis: A case study. *Applied Sciences*, 10:1–20.

Zakeri-Nasrabadi, M., Parsa, S., Esmaili, E., and Palomba, F. (2023). A systematic literature review on the code smells datasets and validation mechanisms. *ACM Comput. Surv.*, 55(13s).