Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Code Clones in Scientific Software: A Case Study on Ray-UI

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Jannis Maier
geboren am: 27.10.1997
geboren in: Bonn

Gutachter/innen: Prof. Dr. Timo Kehrer
Prof. Dr. Lars Grunske

eingereicht am: ..... verteidigt am: .....

In scientific software, the specifications and requirements for a software project can quickly change. Since scientific software is often used to explore ideas that are hard to define precisely initially, it is challenging to know what the software should look like in the end.

Code clones in software can be an indicator of bad style and design. They also increase the effort to maintain the codebase, which means more programming hours away from adding features or enhancing the software. Manual detection of code clones is complex, and finding ways to automate that process is crucial.

This thesis explores Code Clone Detection (CCD) in scientific software. With code clones increasing code maintenance, there is a need for clone detection tools, especially in areas where software is developed in less streamlined processes.

We will look at RAY-UI, a software from Helmholtz-Zentrum Berlin, which has been developed and maintained for 30 years now. Many developers, new technologies, and limited documentation make it hard to keep this software usable with modern systems. Another challenge for RAY-UI developers is that the application's core was written in FORTRAN.

For most established programming languages, CCD tools exist. FORTRAN is not used in new software nearly as much as it was 15, 20, or 40 years ago, thus making it a niche language, with support growing sparse. Still, many older projects – especially in the natural science or financial sector – were written in FORTRAN. Rewriting an entire project might not be feasible or too expensive.

We explore a variety of tools for CCD and how they perform on RAY-UI. We also show the advantages and challenges of the process. Since most tools don't work for FORTRAN natively, we look at what the expansion of one could look like to support FORTRAN. We discuss how this expansion could look and what difficulties can arise. The example we look at in-depth is NiCad – a popular and effective CCD tool.

In the end, only three tools provided usable results: CCFinderSW, Duplo, and PMD's CPD. Of the three, Duplo and CPD were the easiest to set up, but CCFinderSW reported significantly more clones on a preprocessed codebase.

Contents

Acronyms

BNF Backus-Naur Form

CCD Code Clone Detection

LCS Longest Common Substring

TLOC Thousand Lines of Code

List of Figures

1. Introduction

In source code, a code clone is a fragment of code – meaning a few lines of code – that has a nearly identical counterpart somewhere else in the code. Let us assume that these two code fragments are supposed to do the same thing. If we now decide that one of the code fragments semantically needs to be changed, likely, the other one does too. Therefore, we now need to make changes in two locations, increasing the effort to maintain the code's correctness.

It has been shown that clones make up a significant amount [RCK09] of software code and that reducing this amount will lower maintenance costs significantly. Finding so many code clones in large codebases will likely consume much time and may lead to poor results and frustrated developers. Because of this, the automation of the process – code clone detection (CCD) – has gained more and more popularity in the last decades. Many tools have been written and tested to help developers locate code clones in the code. Further, many approaches have been proposed to improve on previous tools.

This thesis will explore a new area for code clone detection. We want to see whether we can apply CCD to scientific software, what challenges arise and what we can learn from this experiment. Scientific software is written close to mathematical models and principles of the natural sciences, which makes code for scientific software often complex and more challenging to understand.

Further, we find that scientific software is often not developed by people with a primary background in computer science. Therefore, it is an area with a great need for code clone detection but one where users may lack experience handling tools with minimal documentation and difficult-to-use command line interfaces.

We want to investigate whether these tools are usable for scientists developing software. Further, the differences between scientific and common software might increase the need for CCD and should also be investigated.

A popular language in scientific software was and is FORTRAN. It is used because of its strengths in writing mathematical formulas and delivering excellent performance. Since FORTRAN is not a common programming language in the present day, we will have an introductory look at that too.

Therefore, our primary goal is to survey CCD tools and find ones that are easy to use and produce valuable results for a developer of scientific software. From the language-specific tools looked at, only one supports FORTRAN natively (PMD CPD), and another we could expand somewhat successfully to support FORTRAN (NiCad). Three other tools are language-agnostic, or at least only need little data on the language syntax and structure. The tools we looked at can be found in the Tools section.

A representative example of scientific software was provided by the Helmholtz-Zentrum Berlin. The core application of RAY-UI was written in FORTRAN. As mentioned earlier, most tools do not support FORTRAN, so we can look at the challenges of finding a good tool and making it work.

The code and its development process will provide valuable insight into the challenges of code clone detection for scientific software since varying programming styles, through time and developers, make parsing the code complex. We must look at many exceptions

to standards, making the process more challenging. Language-agnostic approaches, on the other hand, have the upside of not needing to define convoluted syntax parsing for multiple languages. This research has yet to be done for CCD and motivates a big part of this work.

Only three of the tools we analyzed in-depth produced results. The others failed due to lacking language support, difficult usability, or an unfit detection strategy for our codebase. Of the three tools that successfully detected clones, CCFinderSW performed most effectively, and CPD was easiest to set up but yielded fewer valuable clones. CPD might also need preprocessing of the codebase, depending on the programming language. Details on how the tools perform on the codebase will be shown in the Experimental Results section and we will evaluate the usefulness of these results in the Evaluation section.

We will list all limitations and challenges we encountered during research in the Discussion section. The most significant challenges came up when we expanded the NiCad tool to support FORTRAN and when getting language-agnostic tools to run. One challenge we will not be able to tackle in this thesis is the general approach NiCad takes and how it interferes with the length of apparent code blocks in FORTRAN, rendering our process unable to produce results.

In the last chapter Conclusion, we will look at some lessons learned and what we, as computer scientists, can do better to allow for broader application of these tools. We will also summarize what challenges still have to be tackled to enable these tools to be used by more people.

All the code written for this thesis is available on a GitHub repository [1]. The RAY-UI code is not open source and cannot be openly shared. To allow the reproducibility of our results, we will provide a reduced version for anyone interested. Contact information will be provided in the GitHub repository.

2. Fundamentals

2.1. Scientific Software

Software in science is as vital as it is in the commercial sectors. Many processes in the natural sciences can be assisted with software, especially modeling. Often, already-implemented software significantly impacts how science is done and what science is done.

In the literature[HMS+09][SM08][SSC16], we see a differentiation between commercial and scientific software. But scientific software can be used commercially, and not all software that is not scientific must be for commercial use. Therefore, we will differentiate between scientific software and common software. [2]

[1] https://github.com/Atraxus/SciSoftClones

[2] Common is not meant in a qualitative sense here. It is also not intended to imply that scientific software is uncommon

When looking at scientific software, we need to focus our effort differently than on common software. First, we need the calculations to be easily reproducible and the processes to be transparent. Testing for correctness also needs to be easy, as sometimes critical decisions depend on the results of the software.

Developing scientific software is different from software for a consulting company or similar. Often a company will have a good idea of what its system needs to do. Scientists often use software to understand something they are working on further. Therefore, we need to model complex natural phenomena in the software, which also might need to change quite dramatically during research. This could lead to a unique prototype-like structure in the software code.

2.2. RAY-UI

The Helmholtz-Zentrum Berlin studies the dynamics of materials and investigates solar cell technology. An essential facility for this is the electron storage ring BESSY II, which emits synchrotron radiation, which scientists can use for many experiments. Since beamlines, which prepare the light for experiments, are expensive to build, planners must do a lot of preparation work. This complicated process is time-consuming, and assisting it with software can make it much easier and less error-prone.

Thus, they began developing software [Sch08] roughly 40 years ago, which was later expanded to RAY-UI. Its main goal is to simulate the synchrotron radiation emitted from BESSY II. The main challenge here was translating complex optical phenomena, which occur in the beamlines surrounding the ring, into code. This is one of the reasons why the core application of RAY-UI was written in FORTRAN. Its strengths in high-performance computing and a general fondness for the scientific community made it an easy choice.

The beamline's optical elements have many characteristics that need to be considered for the computation. Therefore, it fits with our definition of scientific software. To simulate a beamline, you need to take many parameters into account.

2.3. FORTRAN

FORTRAN [Ada92] was first released 65 years ago and was a popular language for a long time. Due to its high-performance computing capabilities and an excellent toolkit to model mathematical formulas in code, it found great interest in scientific computing. It was iterated over a lot, and modern versions offer more freedom in writing FORTRAN code. Older versions were strict with whitespace, where a line should start, and how long it could be, leading to sometimes unusual syntax.

FORTRAN versions always stayed backward compatible, so one major downside is a convoluted syntax, with many ways to do the same thing. This makes syntax parsing more difficult, as a project can have varying coding styles if written over a more extended period.

One example of this is the line continuation syntax. FORTRAN is (like Python) whitespace sensitive. This means dividing one instruction over multiple lines will not

compile. Therefore, the FORTRAN developers added syntax to support this feature. The following ways can all be used for line continuation:

- Numbers from $0 - 9$ as the first character following whitespace of a continuing line
- An ampersand (&) at the end of a line
- There is a special case for Format statements, where you can use a backslash (\) at the end of a line

Additionally, you can also use numbers at the beginning of the line to signal a jump point for a GOTO statement. This shows one of the difficulties of parsing FORTRAN, when multiple versions of it are used.

2.4. General Definitions

codebase The codebase is an aggregation of files that contain lines of code. Some tools allow one codebase to consist of multiple programming languages, but we will assume that only one language was used. Codebases with multiple languages are mostly structured in a way that allows easy division into multiple codebases, each with only one of the languages.

Code Fragment A code fragment is a collection of consecutive lines. A code fragment is uniquely identified by its file and the line numbers where it begins and ends [RCK09]. Different granularities of code fragments can be, e.g., blocks, functions, files, and more. In this work, we'll often refer to a code fragment simply as a fragment.

Clone Class, Clone Group A clone class or group is a group of code fragments that are pairwise similar to the degree that we define them as clones. Finding a clone class in a code base means having the same code in multiple locations.

Backus-Naur Form The Backus-Naur Form (BNF) is a syntax specification commonly used to describe programming languages in computer science. For a BNF, we need a replacement statement between a symbol and one or more sequences of symbols. Symbols that do not appear on the left side are non-terminals.

2.5. Code Clones

Code smells are parts in the source code that are likely not negatively affecting its correctness but impact the development process negatively in some way. Readability, robustness, and performance often suffer from code smells.

A specific type of code smell is a code clone. Code clones are fragments of the code that are identical (or similar), either semantically or syntactically.

Differentiating code fragments because of a different identifier or differences in whitespace is not always sensible. Slight variations in a copied fragment could make the code clones dissimilar. We need a similarity metric to categorize the code fragments as clones, even with small differences.

We can find a common taxonomy for code clones into four categories through a lot of the literature concerning code clones and CCD. The categories are enumerated 1 through 4, where a code clone of the first category is an exact match and a semantic match for the last one.

The following categorization is taken from the Chanchal K. Roy et al. paper "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach" [RCK09]

1. Type-1: Identical code fragments except for variations in whitespace, layout, and comments.

2. Type-2: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace layout, and comments.

3. Type-3: Copied fragments with further modifications such as changed, added, or removed statements and variations in identifiers, literals, types, whitespace, layout, and comments.

4. Type-4: Two or more code fragments that are implemented by different syntactic variants but perform the same computation.

These categories help us study the impact and reasons for different code clones. From this, we could decide, which code clones are worth investigating further, and which are tolerable or even necessary.

2.6. Code Clone Detection

Since finding code clones manually is a strenuous task, the field of code clone detection (CCD) came up. In CCD, we are analyzing the code of a given software, either statically or dynamically, to find similarities. Our focus will be on static code analysis.

In static CCD techniques, we are parsing the text in some way and then comparing fragments to find out how similar they are. If a given similarity threshold has been reached for two code segments, we mark them as clones. Their similarity will also decide in which category they fall.

A general process to solve the problem of CCD is hard to define. Different techniques are used and sometimes even combined to find code clones in software. We still want to try to get an overview of the process. In Figure 1 you can find an adapted version of the graphic by Roy et al. in their paper [RCK09].

Most approaches will generally follow this process. Some steps might be skipped, and some may use additional techniques for more efficient or precise CCD.

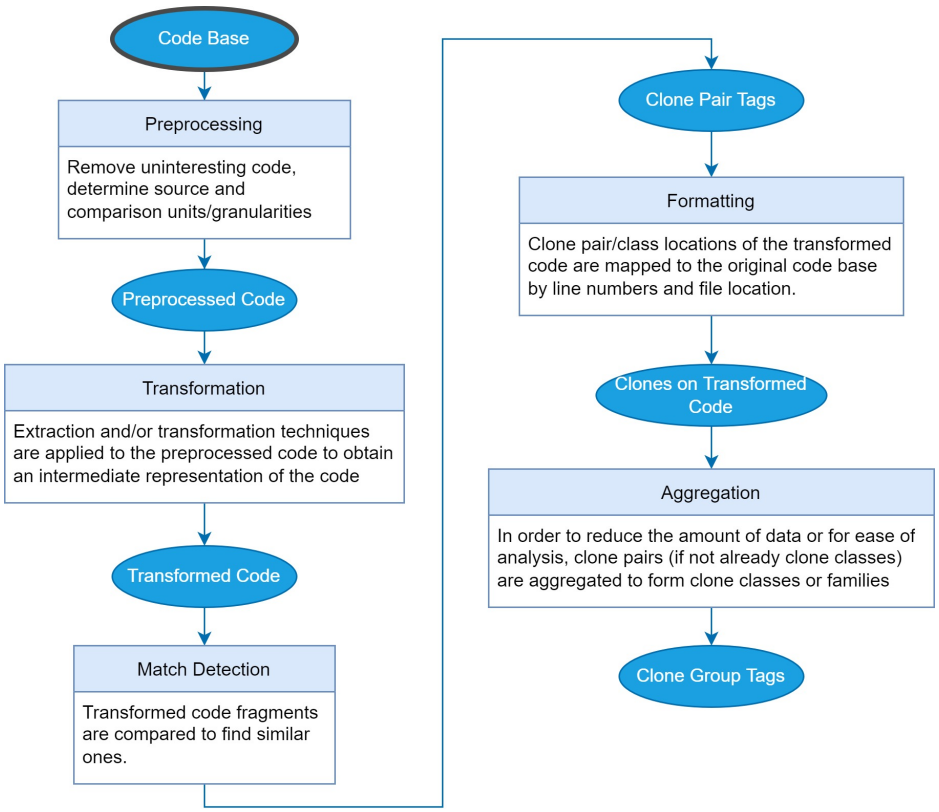


Figure 1: A general approach to CCD

2.7. CCD Techniques

Research has already yielded many approaches to detect clones in software code. Different techniques of preprocessing the code and matching fragments are used in the tools we present later, and we want to go over the background for them here.

Most approaches inherently cannot be language-agnostic and must be defined specifically for each programming language, the tool should support. From that follows that if a tool does not support the language we are looking for and is written with a language-specific approach, we can easily discard it or look into how to expand it to another language.

Only three of the following can be language-agnostic: text-based approaches, token-based approaches, and learning-based approaches. Since we are looking for a tool that supports the niche language FORTRAN in this work, we will focus on them.

Text-based approaches

In this approach, we divide the source code into lines and compare sequences of lines for similarity. If a sufficiently long sequence above the similarity threshold has been found, they will be marked as clones.

Simply comparing lines without any normalization and transformations would yield

low effectiveness. Preprocessing, like normalizing identifiers and whitespace, removing comments, and more, would be necessary for robust clone detection.

Token-based approaches

Token-based approaches are similar to text-based ones, but we look at tokens instead of lines. During the tokenization step, we divide the code into tokens. This step usually includes whitespace normalization since that is often used to divide lines into tokens. Once a sequence of tokens is long enough and passes the similarity threshold, the lines containing the token-sequence will be marked as a clone. Therefore, Token- and Text-based approaches report the same types of clones.

Tree-based approaches

Tree-based approaches convert the code to a parse- or an abstract syntax tree (AST). This conversion is language-specific because the goal here is to represent the code syntax in the tree structure. This abstraction removes variance in whitespace, identifiers, or even types.

Once we parsed the code and generated the tree, we can use tree matching to find clones in similar subtrees. We return the corresponding lines for every subtree, which surpassed our similarity threshold, and which is therefore marked as a clone.

Metrics-based approaches

Metrics-based approaches abstract the code to vectors of metrics. To calculate them, fingerprint functions are used. We use functions, classes, and files as input, and the resulting vectors can be used to compare the code they represent and look for clones. Often, tools create an AST before computing the metrics.

PDG-based approaches

Program Dependency Graph or PDG-based approaches go a step further than the previous methods as they compare semantic similarity. The PDG contains control flow and data flow information about the software code. In this approach, we use isomorphic sub-graph matching algorithms to find similar sub-graphs. While this approach is robust and can find semantic clones, it is computationally expensive. Therefore, it scales poorly to larger projects. Even functions with completely reordered instructions could still be correctly detected as clones.

Hybrid approaches

Hybrid approaches try to incorporate two or more complex approaches to improve their recall and precision. For example, combining Tree-based and PDG-based approaches can extract much information from the codebase.

Learning-based approaches

Learning-based approaches use machine learning and neural networks to improve their ability to detect if code fragments are clones. This neural network is used primarily to determine the similarity of two clone fragments or even guess if the two are clones.

3. Related Work

Code clones, their impact on the codebase, and their detection have been studied for quite a while now. Studies [JDHW09][RCK09] argue that code clones negatively affect code correctness and maintenance costs. Until now, the majority of research has been done on production code and plagiarism detection. Some studies investigate CCD for test code [BD18] or documentation [WELL10], and only a little or no research has been made for scientific software.

Research that differentiates between scientific and common software proposes the theory that there could be a difference for CCD too. Some research [KB14] has been done on how code smells, in general, appear in scientific software and how they affect it. One study covering the testing of scientific software mentions code clones and their detection but does not go into further detail. There does not seem to be a trend to differentiate scientific software for code clone detection.

Here is where this thesis tries to bring different research threads together. We will focus on CCD in scientific software and how well current tools can detect clones in it. Further, we will primarily do foundational work to find the answer if scientific software indeed differs when looking at it from a CCD standpoint.

To at least have some way of comparing the results we produced, we will look at research for CCD on common software, test code, and documentation. One of the most extensive surveys in the field is "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach" from Roy et al. [RCK09], where they compare 38 tools in great detail.

Another more recent survey, "A Systematic Review on Code Clone Detection," Ain, Qurat Ul, et al., focused on 26 current tools. The team investigated how well these tools detected different clone types and proposed that novel approaches are needed to catch all four clone types. Most tools do not support type-4 clones.

Both studies investigate the availability and usability of these tools. As we are using the RAY-UI codebase, we can investigate this further. We will explore how well the chosen tools can be applied to RAY-UI in the 4 and 5 sections.

4. Tools

Overall, we looked at 16 CCD-tools during research, but we will focus on only 5 in this section, and go over how they work and what techniques they use for pre-processing, normalization, and transformation. Further, we will look at their capabilities to detect

different categories of clones. As we established in the 2 section, we can divide these 16 tools into categories as follows:

Table 1: A table of all tools examined. In column 3, we can see which tools worked natively for FORTRAN. Some tools, marked with a star, will get a further examination.

Tool	Approach	Works natively	Source
NiCad	Text-based	No*	[RC08]
Dup	Text-based	No	[Bak95]
Duplo	Text-based	Yes*	[Lid22]
PMD CPD	Token-based	Yes*	[TDD22]
SourcererCC	Token-based	No*	[SSS+16]
CCFinder	Token-based	No	[SYCI17]
CCFinderSW	Token-based	Yes*	[KKI02]
CloneDR	Token-based	No	[BYM+98]
Boreas	Token-based	No	[YG12]
CP-Miner	Token-based	No	[LLMZ06]
iClones	Tree-based	No	[GK09]
GPLAG	Graph-based	No	[LCHY06]
CCLearner	Learning-based	No	[LFZ+17]
ConQat	Hybrid	No	[DPS05]
CDSW	Hybrid	No	[MHH+13]
FRISC	Hybrid	No	[MHH+12]

We will focus on the following tools: (1) NiCad, (2) PMD CPD (also referred to as CPD), (3) SourcererCC, (4) Duplo, (5) CCFinderSW. Most tools were discarded due to being language-specific and not supporting FORTRAN. The tools we chose to examine further were selected because (1) we could write a plug-in to expand them, (2) supported FORTRAN natively or (3, 4, 5) were (mostly) language-agnostic. More on this in the Evaluation section, where we evaluate our findings for the tools.

When surveying the current research, we can get a good view of how these tools could perform, even before applying them. There is a benchmark in the field called BigCloneBench [SIK+14], which is commonly used to check the quality of one's own CCD tool before publishing it. These benchmarks are not done for FORTRAN, so we can only get a rough idea of their performance.

First, we need to mention that during the time of writing this, there are no published results for CCFinderSW directly – only on its predecessor CCFinderX – so we will not include it in this comparison. We could find results for CPD, SourcererCC, Duplo, and NiCad.

For type-1 clones, all tools perform flawlessly, with only Duplo being at 89% recall. For type-2 clones, only NiCad had a 100% recall, with SourcererCC being a close second with 98% and CPD trailing closely behind with 94% recall.

For type-3 clones, these benchmarks differentiate between 4 categories:

- Very Strong Type-3 (VST3)
- Strong Type-3 (ST3)
- Moderately Type-3 (MT3)
- Weak Type-3/Type-4 (WT3/T4)

Clones from the WT3/T4 category only show 0-50% syntactic similarity and are not detected by any tools. Finding these kinds of clones requires a robust syntactic foundation of the tool, which would be hard to achieve with purely static analysis of the code. We can see that Duplo barely finds any, and CPD performs slightly better. NiCad and SourcererCC find a reasonable amount of the syntactically closer type-3 clones (VST3, ST3).

We can say that NiCad and SourcererCC perform best for this benchmark, with CPD and Duplo falling a bit behind but still showing promising performance. Finding most type-1 and type-2 clones in a codebase can help refactor it to an easily maintainable state. The predecessor of CCFinderSW (CCFinderX) performs similarly to CPD.

After getting an overview of the tools' performance, we can now take a more profound look at how they work.

4.1. NiCad

NiCad is a language-specific CCD tool. It was developed by Chanchal K. Roy and James R. Cordy at Queen's University. The tool supports C, Java, Python, C#, PHP, Ruby, Swift, ATL, Rust, Solidity, and WSDL. It does not support FORTRAN, but has a plug-in-based architecture for language modules. Therefore, we will look at how such a plug-in could be written and if this could be feasible for scientists working on a software project.

TxI & Grammars

Before we begin dissecting NiCad, we need to understand an essential building block of it. The language plug-ins for the tool are built with TXL.

The TXL team has researched rule-based structural transformation for over 15 years. They developed the TXL language for efficient static code analysis and transformation. The language is a hybrid consisting of functional programming and rule-based definitions. It supports unification, implied iteration, and deep pattern matching. With unification, we can combine patterns, with implied iteration we can address all sequences of one type at once, and with deep pattern matching, we can match patterns inside patterns.

A TXL program typically consists of two components: (1) a BNF grammar, that describes structures to be transformed, and (2) transformation rules written as pattern and replacement pairs. In our case, we want to (1) describe what a code fragment looks like and (2) define how normalization and other refactorings could look like, to find specific clone classes. These components are put together via functional programming, where we give an algorithm for using the given rules and transformations.

The TXL language uses formal tree rewriting to translate the given source code into the transformed output efficiently. Most of this is hidden from the user, who works with the aforementioned rules and define-statements.

We will now look at an example define-statement. What we can see in algorithm 1 is rule 202 in our NiCad FORTRAN grammar, which we will take an in-depth look into later. The rule defines what a program unit can look like. It concatenates four possibilities with or-operators, indicated by the pipe symbol (|). It can either be the main program, a function, a subroutine, or a block. All of these are parts of the FORTRAN language we can use to define the language structure and ultimately define what a code fragment looks like.

Each of the four possibilities in the rules has its definition, which can be seen by the square brackets around a character sequence. The square brackets indicate a non-terminal symbol.

Algorithm 1: R202

```
define ProgramUnit
    [MainProgram]
    | [FunctionSubprogram]
    | [SubroutineSubprogram]
    | [BlockDataSubprogram]
end define
```

We would now define these rules until we define terminal statements for every branch in our grammar structure. This means defining a rule, which can be seen in algorithm 2. A character sequence, starting with the '-symbol, indicates a terminal symbol – a string that needs to match exactly in the source code. Ident here stands for an identifier, which the TXL language predefines.

Algorithm 2: R1105

```
define ModuleStmt
    'module [Ident] end define
```

NiCad Process

Before that, we will look at the process behind NiCad and the techniques it uses to do CCD. In Figure 2 - taken from a paper by Roy et al. [CR11] - we can see the general three-phase process, NiCad follows.

In the parsing/extracting phase, we start by tokenizing the codebase. Then the tool builds syntax trees by parsing the code with a given grammar and language rules. Similar to how a compiler operates, we build the trees by looking for terminal symbols in the tokens and building more complex patterns. For example, a terminal symbol could be an identifier like *name* or $number\_of\_elemts$ and a more complex pattern could be an assignment statement like $string\ name = "Anna"$ or $int\ number\_of\_elements = 4$.

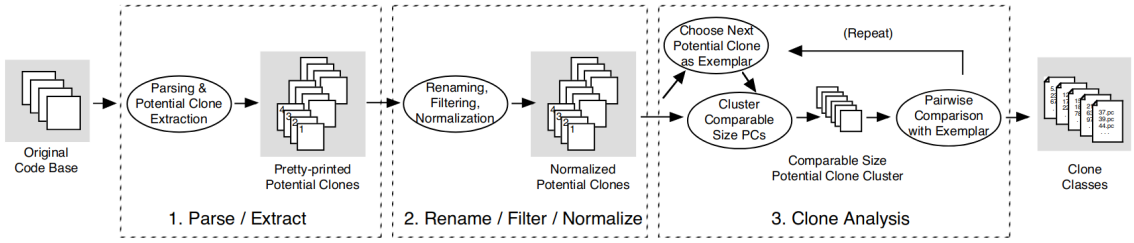


Figure 2: The NiCad CCD Process

In the rename/filter/normalize phase, we apply more rules to prepare the data for the clone analysis phase further. The operations we use here are

- Rename
- Filter
- Normalize

Renaming means we take identifiers for functions or variables and generalize them somehow. One example would be blind renaming, where we change every identifier to the same symbol (e.g., "X"). This helps us ignore variance in the naming of the clones and thus, allows identifying of category 2 clones.

Filtering means we take specific statements, patterns, etc., and remove them from the code fragments before doing clone analysis on them. This helps us to identify category 3 clones, where a few differing statements are allowed.

Normalizing means we address a language's ability to express the same thing in different ways. For example, if we want to check if a given pointer in C is not set, we can do this in two ways:

- $if\ (\ !pointer\ )$
- $if\ (\ pointer == NULL\ )$

Both ways accomplish the same but would not be recognized as the same by simple string comparison. Normalizing them to the same statement does not change the semantic meaning and allows for clone detection of some type-4 clones.

In the clone analysis phase, we do two things: compare potential clones and group resulting clone fragments into clone classes. One optimization used here is to cluster the fragments based on their size. Based on a similarity threshold given by the user, the size difference between two fragments, cannot exceed the threshold.

Now that we grouped the fragments, we can compare them using the Longest Common Substring (LCS) algorithm. The LCS algorithm might be familiar from the diff tool when working with git. It is a robust and efficient matching algorithm over two sequences, finding the longest subsequence between them. The subsequence can skip characters or in our case complete lines of code, to find a clone, even if a different

line has been added to one fragment. This helps when detecting Type-3 clones. It makes it possible for differing lines to exist in two clones, as shown in the example in Figure 3.

The metrics on the bottom are in order: the total number of lines of the fragment, the total number of lines dissimilar to the other fragment, and the percentage of unique items (UPI). The threshold we are using here to see if two fragments are clones is the same one we used for the size difference.

For an example threshold value of 30%, we can see the left example would be considered a clone, and the right one would not be. Despite one fragment having a UPI value of 22.22%, which is below our target, the other is above the threshold with 50%.

Item No.	Sequence 1 (Original Segment)	Sequence 2 (Copied and Edited Segment)	Similarity
1	void	void	1
2	sumTimes	sumTimes	1
3	(int n) {	(int n) {	1
4	float sum=	float sum=	1
5	0.0;	0.0;	1
6	double product =	double product =	1
7	1.0;	1.0;	1
8	for (	for (	1
9	int i=1;	int i=1;	1
10	i<=n;	i<=n;	1
11	i++){	i++){	1
12	sum=	sum=	1
13	**sum + i;**	**sum + (i * i);**	**0**
14	product=	product=	1
15	**product * i;**	**product * (i * i);**	**0**
16	fun	fun	1
17	(sum, product);	(sum, product);	1
18	} }	} }	1
	Total Items = 18 Unique Items = 2 UPI = 11.11%	Total Items =18 Unique Items = 2 UPI = 11.11%	

Item No.	Sequence 1 (Original Segment)	Sequence 2 (Copied and Edited Segment)	Similarity
1	void	void	1
2	**sumTimes**	**sumTimesExtended**	**0**
3	**(int n)** {	**(int n, int m, int x, int y)** {	**0**
4	float sum=	float sum=	1
5	0.0;	0.0;	1
6	double product =	double product =	1
7	1.0;	1.0;	1
		10 Unique Lines	**0**
8/18	for (	for (	1
9/19	int i=1;	int i=1;	1
10/20	i<=n;	i<=n;	1
11/21	i++){	i++){	1
12/22	sum=	sum=	1
13/23	**sum + i;**	**sum + (i * i);**	**0**
14/24	product=	product=	1
15/25	**product * i;**	**product * (i * i);**	**0**
16/26	fun	fun	1
17/27	(sum, product);	(sum, product);	1
18/28	} }	} }	1
	Total Items = 18 Unique Items = 4 UPI = 22.22%	Total Items =28 Unique Items = 14 UPI = 50%	

Figure 3: Comparing two code fragments as NiCad would

Now that we have found all pairwise clones, we can group the fragments that have multiple clones into clone classes because being a clone is a transitive property (at least for reasonable similarity thresholds).

Depending on how much is implemented for the language, NiCad can detect type-1, type-2 and type-3 clones, with renaming, filtering, and normalization. With enough normalizations implemented, we could even detect semantic type-4 clones to some degree, though this will be case-dependent and time-consuming to achieve.

4.2. PMD CPD

PMD CPD is another language-specific CCD tool, but it supports FORTRAN natively and thus can be used on our example project. Additionally, it supports CCD in Java, C, C++, C#, Groovy, PHP, Ruby, JavaScript, PLSQL, Apache Velocity, Scala, Objective-C, Matlab, Python, Go, Swift, and Salesforce.com Apex and Visualforce.

For parsing the code, a simple token-based approach is used. We divide the code into tokens based on one or multiple delimiters, which we can group into code fragments. CPD uses statement-level granularity.

For the matching part, the developers reworked PMD CPD a few times and are currently using the Rabin-Karp string matching algorithm [vBD20]. The algorithm works as follows:

Algorithm 3: Rabin-Karp Algorithm

```
Data: string s[1, ..., n], string pattern[1, ..., m]
Result: position p of an exact match or -1 if not found
hpattern := hash(pattern);
i ← 0;
while i ≤ n − m + 1 do
    hs := hash(s[i..i + m − 1]);
    if hs = hpattern then
        if s[i..i+m-1] = pattern[1..m] then
            return i;
        end
    end
    i + +;
end
return −1;
```

With a worst-case running time of $\mathcal{O}(nm)$, this algorithm does not look interesting at first glance, but the key performance impact comes from the hashing function quality. Assuming we get a perfect hashing function, we would only need to check the pattern against a substring once. Therefore, we would have to check at most $n - m$ positions and compare m characters once, giving us a running time of $\mathcal{O}(n)$.

The PMD-Team does not make any claims on what clones their tool detects, but it has been empirically shown to detect type-1 and type-2 clones. [RCK09]

4.3. SourcererCC

SourcererCC is a token-based clone detector, which is language-agnostic. The tool is well-optimized for large codebases and uses multithreading to increase performance. It uses a two-phase process that first tokenizes the codebase and then runs a clone-matching algorithm.

In Figure 4, we can see an overview of how SourcererCC detects clones in a codebase. After parsing and tokenizing the code, the codebase is divided into code blocks. The algorithm uses a partial index to speed up the comparisons. To create the partial index, they use a filtering heuristic, not all tokens of the code blocks. The index can then be used to find clone candidates. The returned candidates for each sub-block is rechecked to see if they genuinely are clones.

We use the *Overlap* similarity function to calculate similarity scores, which returns the absolute number of shared tokens from two code fragments. We can then use a threshold with this number to define at what point two fragments should be considered

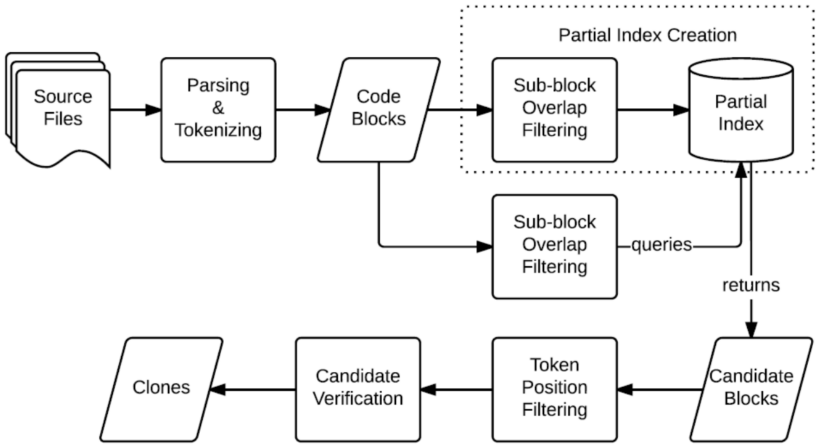


Figure 4: SourcererCC's clone detection process

clones. Formally, the similarity function would look like this:

$$OS(B_x, B_y) := |B_x \cap B_y|, \tag{1}$$

where B_x, B_y are clone fragments representing a set of tokens. This means if a token appears twice in one fragment and thrice in another, the OS-function would count it twice. With a given threshold θ, we would want $OS(B_x, B_y) \geq \theta \cdot max(|B_x|, |B_y|)$. One important takeaway here is that we are losing positional information with this. SourcererCC is agnostic to the specific position of tokens in a code fragment, which authors argue improves the ability to detect type-3 clones.

Other heuristics and techniques, omitted here, can be found in their paper. The tool uses a very sophisticated approach to achieve its performance, and the full extent of its algorithm would be too much to cover here.

4.4. Duplo

Duplo uses a language-agnostic text-based approach to find clones and match them. The language specificity mentioned on their GitHub page only concerns pre-processing of the codebase. We intend to do this ourselves, and therefore we can safely use this tool even though FORTRAN is not directly supported.

Duplo's technique for detecting clones is based on the algorithm used in Duploc, hence the name of the tool. In the following, we will go over Duploc's detection process, which you can see in Figure 5.

The Duploc-Team describes their tool in three parts: (1) algorithms for fragment comparisons, (2) visualization/organization of the comparison data and (3) pattern matching to condense the data.

In the first part, we will look at how Duplo extracts code fragments from files. This is the step from the "Entire File" to the "Effective File" in the Figure. Each line in the original file is treated as a code fragment. The tool can remove comments and

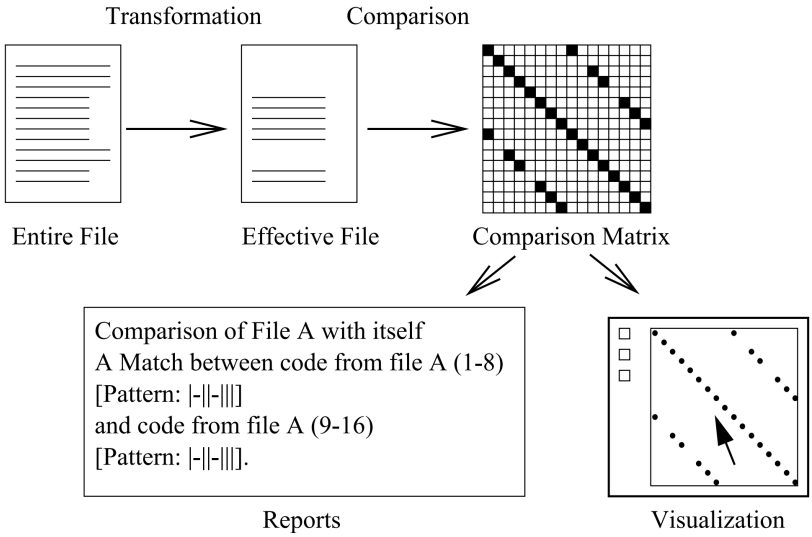


Figure 5: Duploc's clone detection process

whitespace with a little adaption to a specific language. This reduces the practical code fragments, as empty lines and comment lines get discarded.

This is all the language-specific implementation there is. The process from here on will be the same for every language. To get the comparison data, Duplo will now compare the effective file against another one (from other files) or itself.

The comparison itself is done via simple string matching and will produce a Boolean value as a result: $true$ means the two lines are an exact match and $false$ means there is at least one varying character.

Comparing $\mathcal{O}(n^2)$ many lines is costly; therefore, Duploc implements a hashing function to sort the lines into buckets. Two lines, which are the same, will produce the same hash and thus be put in the same bucket. Therefore, only lines in the same bucket need to be compared, and no false negatives occur because of this.

This process builds a matrix of size $(\#lines)^2$. Each cell in the matrix is a boolean value for the comparison of two corresponding lines – this can also be seen in the Figure as the "Comparison Matrix."

Duplo has a GUI in a different repository called DuploQ to visualize the comparison matrix. This idea also stems from the Duplo paper, in which the authors argue a visual inspection of the matrix rather than the returned textual report.

They make two arguments for this: the visualization can give a quicker overview of the situation (relative number of clones, size of clones, the evolution of the codebase) and the exploratory approach of human oversight can lead to findings a preprogrammed pattern-matching algorithm would not catch.

The following patterns can be of interest when examining the comparison matrix:

- Diagonal lines of dots, which indicate copied sequences
- Holes in lines, which indicate changed portions in clone fragments

- Rectangles, which indicate periodic occurrences of the same code
- And others

The extraction algorithm used to create the textual reports allows for holes in diagonal lines, as otherwise a single character change in a single line could break one clone into two.

4.5. CCFinderSW

CCFinderSW is a token-based language-specific CCD tool. We can use it in our study, though, as it supports FORTRAN well enough that we could easily expand it (a FORTRAN grammar is already present). The tool is based on CCFinderX, which in turn is a redesign of CCFinder, a tool that has been developed in 2001, by Toshihiro Kamiya, Shinji Kusumoto and Katsuro Inoue.

In the following, we will go over the clone detection process behind CCFinderSW, which is visualized in Figure 6.

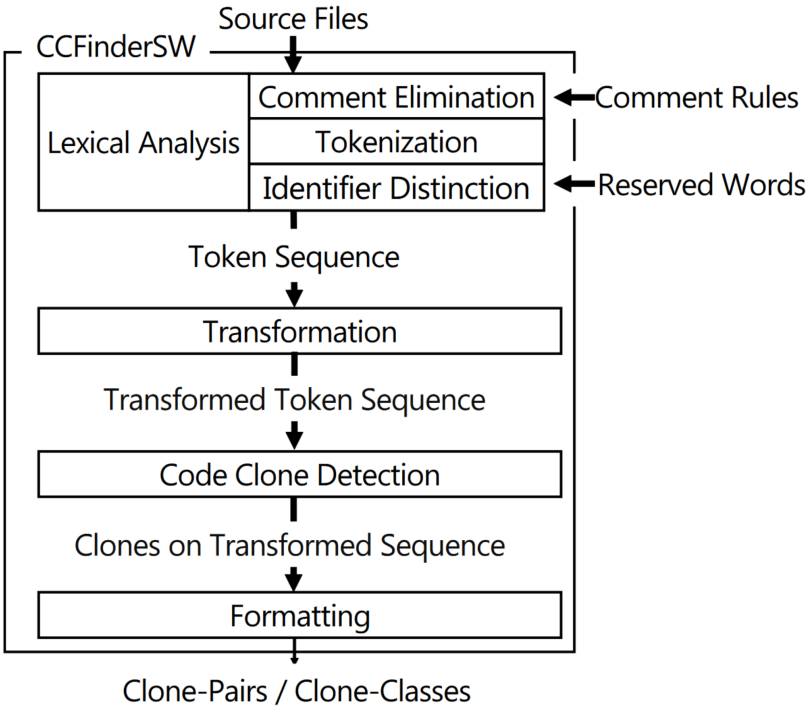


Figure 6: CCFinderSW's clone detection process

As we can see in the Figure, the tool's process can be divided into four steps: (1) Lexical Analysis, (2) Transformation, (3) Code Clone Detection, and (4) Formatting.

For the first step – the lexical analysis – the tool removes comments based on comment rules. These rules are provided in a text file and describe what syntax comments can use for specific languages.

Further, the source files are tokenized, and the tool looks for identifiers. To use this feature, a regular expression describing the variable syntax and a file with reserved words from the language must be present so the tool can distinguish them. Reserved words are usually keywords like if, for, etc., and mathematical constants.

FORTRAN does not in and of itself have reserved words. You could still use them as variable names. Since this would lead to poor readability, most programmers would avoid doing it.

For the second step – Transformation – the tool transforms all identifiers into the same token. This means that all variable and function names will end up as the same symbol (e.g., '$') in the processed string.

In the third step – Code Clone Detection – Duplo uses n-grams. They are commonly used in investigating the occurrence of contiguous strings or finding matching lines. In this case, the n-grams are made up of the preprocessed tokens from the previous steps. Each n-gram is assigned a hash value to accelerate the comparison process. The user defines how large the n-grams are supposed to be.

Here is an example of this process: Let a, b, c, d be tokens and *abcdbabcdbaba* be a sequence of tokens. The threshold for the n-grams is set to 4. From this string, we generate ten 4-grams. This will lead to 6 unique hash values, which can be seen in table 2. Like other tools, we can merge clones of increasing indices to get larger clones.

Substring	Hash Value	#
abcd	1234	1
bcdb	2342	2
cdba	3421	3
dbab	4212	4
babc	2123	5
abcd	1234	6
bcdb	2342	7
cdba	3421	8
dbab	4212	9
baba	2121	10

Hash Value	Appearance
1234	{1, 6}
2342	{2, 7}
3421	{3, 8}
4212	{4, 9}
2123	{5}
2121	{10}

Table 2: n-grams for the example: *abcdbabcdbaba*

In the fourth step – Formatting – the tool outputs its gathered clones into a text file. This can be done either in the format of CCFinder's or CCFinderX' output. Further information on how the files are formatted and general documentation can be found on the GitHub page of the tool.

5. Experimental Setup

In this section, we will go over the steps needed to prepare the tools and the codebase for the clone detection process of the different tools.

Beginning with the codebase, we will provide two versions: preprocessed and not preprocessed. Since some tools are aware of the programming language and can preprocess the codebase to varying degrees, and some can not, we will remove some of these challenges for one test. This is mainly done to directly compare the techniques of the tools and not their ability to parse the code base. Especially in the case of RAY-UI, where the syntax and formatting differ quite a bit between files.

To have a more consistent codebase, we used a formatting tool to get standardized formatting. This tool enforces a FORTRAN 77 style, where column positions of statements are essential, and everything is indented following the same strategy.

Further, we removed all comments and preprocessor statements via a simple python script. The NiCad FORTRAN plug-in had difficulties parsing some uncommon syntax, so we changed a few syntax passages.

An example can be seen in algorithm 4, where the placement of spaces made it difficult to differentiate if a dot belonged to the operator or a number (AND, OR, GE, and other operators are surrounded by dots in FORTRAN). The first line is the hard-to-parse one; the second is the new version.

Algorithm 4: AND-Operator Syntax

```
IF (HVA.GE.5000..AND.HVB.GT.30000.)
IF (HVA .GE. 5000. .AND. HVB .GT. 30000.)
```

We will also do all tests on an unprocessed codebase, as they can be found in the project RAY-UI. This includes varying formatting styles regarding whitespace, comments, preprocessor directives, and more.

The tools were all interfaced via the terminal.

5.1. Hardware and Measurements

For running the tools, we are using a Linux system running natively on the following Hardware: a Ryzen 5 3600, 32 GB of 3200 MHz RAM, and a 1 TB NVMe SSD. None of the tools utilize the GPU, so it has no relevance here. The Linux distro we use is the current version of EndeavourOS as of writing this (19.6).

5.2. Tool-Setup

5.2.1. Duplo

Since Duplo is text-based, we do not need to take many steps to prepare the tool. Duplo needs a list where all files we want to check for clones are listed – one file per line.

```
./duplo /path/to/filelist.txt /path/to/output.txt
```

5.2.2. CCFinderSW

Since CCFinderSW is token-based and uses grammars for parsing the codebase, we need to do a few small steps to make the tool work. First, we need to define the syntax of comments in a text file. How this is done for FORTRAN can be seen in algorithm 5, where we defined one inline comment and multiple full-line comments.

```
#start
!
#linestart
#
#linestart
c
#linestart
C
#linestart
*
```

Algorithm 5: Excerpt from FORTRAN comment file for CCFinderSW

Then, we need to compile a list of reserved words, which are easy to find on the web. We get these words from the FORTRAN specifications for the versions we are interested in and copy them to a text file. Each line contains one word, which will be treated as a reserved word.

5.2.3. NiCad Plug-In

Writing a plug-in for NiCad consists of writing a set of rules on how to parse, rename, filter, and normalize the source code to prepare it for the clone analysis stage. The minimal set-up one can do is write how to parse the language and not write rules for the other three. Renaming, filtering, and normalizing are only needed if we look for near exact clones, but we can start with category 1.

Now, for parsing, we need a grammar of the language and a few rules telling NiCad, how to divide the source code into fragments (blocks, functions, etc.) Writing a grammar is quite time-consuming, but luckily some languages already have one on the TXL website.

Since the given FORTRAN grammar could not parse the codebase, we will need to expand it to allow for special cases. This means going through the grammars rules and adding special cases where required.

The following expansions were needed to parse the majority of the codebase:

- allow dollar signs ($) in variable names
- filter pre-processor statements
- allow single integer line continuation

- add $bind(c)$ statement parsing from Intel FORTRAN
- add unit locator *newunit*
- allow dollar sign ($) at the end of a format statement
- allow 1 to be a format statement separator for special multiline syntax
- allow for $\_p2$ syntax indicating double precision
- allow *dreal* as type specification
- allow $USE,\ INTRINSIC\ ::\ ISO\_C\_BINDING$ for use statement
- allow $real^*8$ notation (indicates precision)
- some robustness for imperfect syntax (that still compiles, e.g., wrong white spaces . *or.*)
- additional special syntax parsing

After the codebase gets parsed correctly, we can write a simple TXL file to define which kind of code fragments we want to break the code into. For example, if we want to detect clones on the block level, we could use the Function, Subroutine, Block, and Module statements used throughout the FORTRAN language. Each code fragment could represent a block we want NiCad to extract.

After dividing the codebase into the desired granularity, we add tags to the code fragments, with the filename and their start and end line number in the file. We do this, so we can later map the fragments to the codebase and find the original code fragments. This is especially relevant after the parsed fragments have been normalized, filtered, and renamed since it could be difficult to recognize them afterward.

Now that we can parse the language and could, in theory, look for type-1 clones, we can go a step further and add some processing to make type-2 and type-3 clones detectable. A usually easy step is to add blind-renaming, where we transform variables and literals in the code to be the same. For example, we change integer literals into a *1*, string literals into an *X*, and float literals into a *1.0*.

Blind-renaming makes it so that small implementation details are not considered when looking for a clone. This might be especially useful if the clone is a function that only differs in one or a few literals. Because we could efficiently rewrite the function to take a few more parameters and save a lot of duplicated code.

Sadly, we could only get the variable names to be renamed for our FORTRAN plug-in. All other literals stayed the same, despite having rules to replace them. The reason behind the bug is unknown to us, but it might be possible to achieve this with a few changes to the base grammar.

5.2.4. SourcererCC

We can quickly test SourcererCC with a provided Linux Image. This was tested on the Virtual Machine and the previously mentioned Linux install. The tool was easy to use on the example project, but it would simply stop after the tokenization phase on our codebase. Because of the software's lack of error messages and support from the developers, we cannot get results from this tool.

5.3. Research Questions

In the following sections, we intend to answer questions about accessibility, usability, and effectiveness of CCD in scientific software. The following three research questions are foundational for our research:

- **RQ1:** Are there differences between scientific software and common software, which handicap the CCD tools?
- **RQ2:** How easily can CCD tools be applied to our codebase?
- **RQ3:** Are the results from these tools relevant to our codebase?

6. Experimental Results

As mentioned earlier, most tools don't run on the RAY-UI codebase. We tried many tools to find out why they failed. The most significant failed tools will be mentioned here, to answer **RQ1**. Seeing why the tools fail, gives us insight into the difficulties of our codebase. Adding what we know so far about scientific software, we can extrapolate difficulties, tools might encounter in general for scientific software.

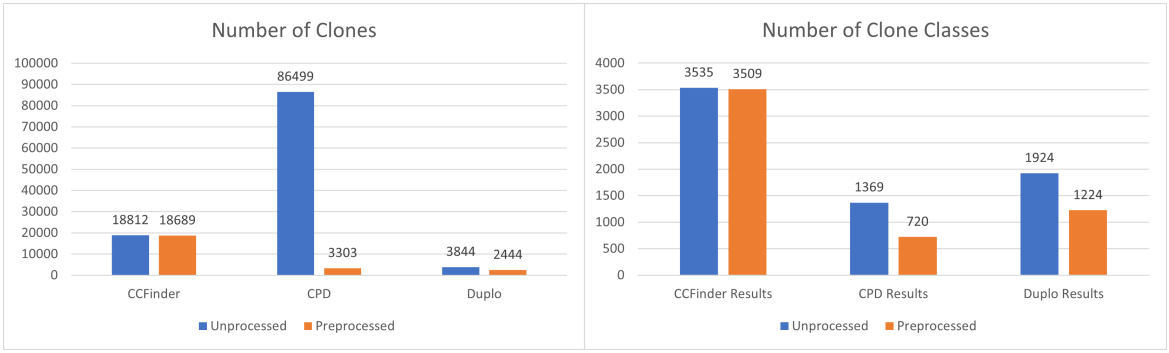
We tried different approaches to see how easily these tools could be used. First, we tried simply running the tool, then we tried modifying the tool slightly, and in one case (NiCad) we tried to write a plug-in, to expand the tools' language support. These experiments provide us with more information to answer **RQ2**.

Answering **RQ3** motivates most of the results we gathered, and the metrics calculated from them. Since relevance cannot be easily measured via one metric, we will look at average clone length, reported clones, agreement scores between the tools, and more. When evaluating relevance, we need to make sure the results: are not random, not only consist of few-line clones, not only report false positives and others.

PMD CPD, Duplo, and CCFinderSW worked on our codebase and reported clones. We are going to analyze their outputs and apply the previously mentioned metrics.

6.1. NiCad

With our plug-in, we can extract 111 code fragments from the codebase: functions, blocks, subroutines, or modules. These are often quite large due to the nature of the codebase.



(a) Number of clones for each tool (b) Number of clone classes for each tool

Figure 7: Comparing the tools on the unprocessed and preprocessed codebases

With quite a bit of the codebase extracted, we were still unable to detect any clones, though. Even with a threshold of 30% for code fragment differences, and blind renaming of the variable names, there still were no clones detected. NiCad reported that in total it made 1292 potential clone comparisons.

As we will see later, most of the clones other tools found were in the RAY.FOR file from our codebase, which NiCad was unable to parse fully. In future work, this could be explored further. We could look at the example in Figure 11 to see if our assumption on why NiCad doesn't report clones is correct.

6.2. SourcererCC

SourcererCC was only partly successful in the case of RAY-UI. The tool does not continue matching clones after tokenizing the codebase successfully. Going through the log files, the reason for this was not apparent, and the team behind it only offered support to paying customers. Therefore, we cannot take this tool into account for the following analysis.

6.3. CCFinderSW, Duplo, and PMD CPD

The three remaining tools: CCFinderSW, Duplo and PMD CPD were able to find clones. The results for the unprocessed and preprocessed codebases vary quite a bit. In the following section, we will use the word *finding* in the sense, that the tool reported a fragment as a clone. These are not verified clones at this point.

General metrics

In Figure 7 and 8, we can see the results of the tools on the unprocessed and preprocessed codebases. The metrics are the same for all tools: the number of clones found, the number of clone lines, the number of clone classes, and the average number of lines per clone.

In Figure 7, we can see the results of the tools on the number of clones found and the number of clone lines total. The number of clone lines is higher than the number of lines in the codebase, as some lines are part of multiple clones.

The first outlier we can see is the number of clones found by CPD on the unprocessed codebase. With roughly one magnitude more clones than the other tools, it is an unusual result. The other tools found roughly the same number of clones for the unprocessed and preprocessed codebases.

Further, we can see that the number of clone classes is much lower than the number of clones found for CPD on the unprocessed codebase. For the preprocessed codebase, we have a ratio of roughly 4.6 clones per clone class, while for the unprocessed codebase, we have a ratio of roughly 63.2 clones per clone class.

Going through the tool's output, we can see that many clones are short comments with one or two lines of code. These clones are found in high numbers, leading to this high number of clones per class.

CCFinderSW found the most clones for the preprocessed codebase, where it found 5.7 times more clones than CPD and 7.6 times more clones than Duplo. It also reported a disproportionately lower amount of clone classes than the other two tools. It had roughly 2.9 times more clone classes than Duplo and roughly 4.9 times more clone classes than CPD. Therefore, we can see, that CCFinderSW grouped more clones together per clone class.

Duplo found the least number of clones overall but had more clone classes than CPD.

In Figure 8, we can see the number of total clone lines found, which includes duplicates and the average clone length for each tool.

All tools report more clone lines in the unprocessed codebase. The difference is especially significant for CPD, with roughly 5.9 times fewer lines reported. CCFinderSW overall reported the most clone lines, with an average of 6 clone lines per line of clone.

Looking at CPD again, we can see a significant increase in average clone length as we go to the preprocessed codebase. Comparing this with the other two tools, we can see, that CPD still detects smaller clones.

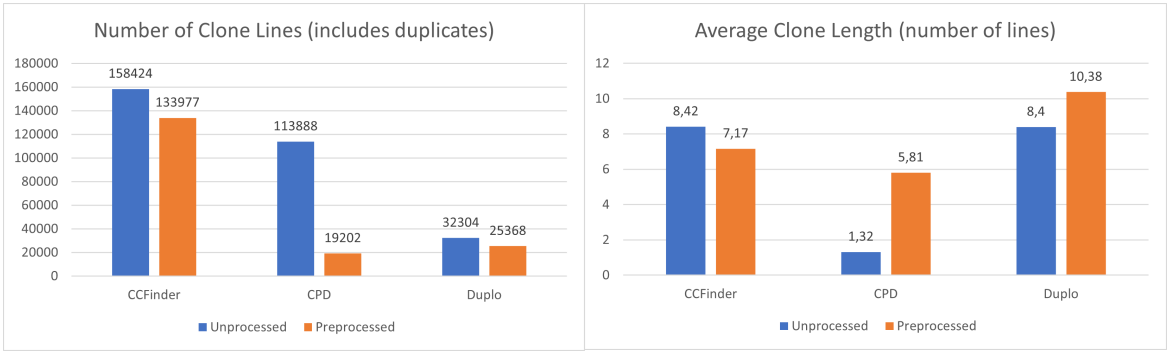
Duplo found the least amount of clone lines, and the average clone length for it is the highest of the three. CCFinderSW's average clone length decreased from the unprocessed to the preprocessed codebase.

Clones in one file

For the largest file in the codebase, we will map every line of code to the number of times this line is part of a clone. In Figure 9, we can see the results for the unprocessed $RAY.FOR$ file on the left and on the right for the preprocessed one: $RAY.f90$.

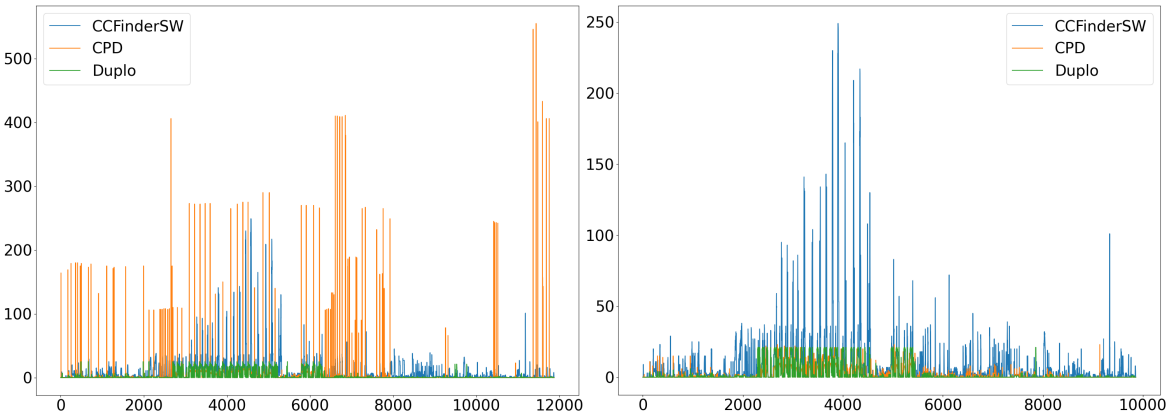
The line numbers on the x-axis vary from the left to the right graph, because of the preprocessing, where we remove comments and more. The number of clones a line is a part of on the y-axis also varies, where some lines on the left are part of over 500 clones, and the highest on the right is not higher than 250.

The previously discovered reduction of clone lines detected from unprocessed to the preprocessed codebase can easily be seen going from the left to the right graph. The



(a) Number of lines for each tool (b) Average clone length in lines for each tool

Figure 8: Comparing the tools on the unprocessed and preprocessed codebases



(a) RAY.FOR file (b) RAY.f90 file

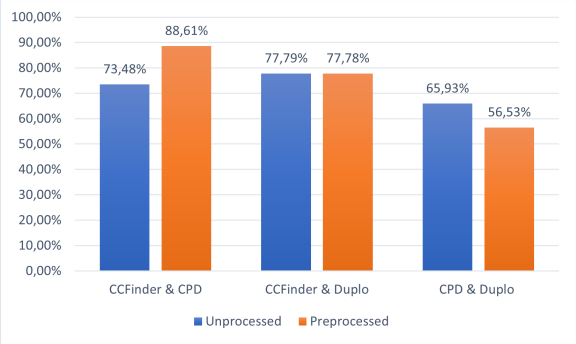
Figure 9: A heat graph of the clones found per line for each CCD tool on the RAY.FOR-file of our codebase. On the left are the results for the unprocessed codebase, on the right for the preprocessed one.

overall higher amount of clone lines detected from CCFinderSW is harder to see in the unprocessed file.

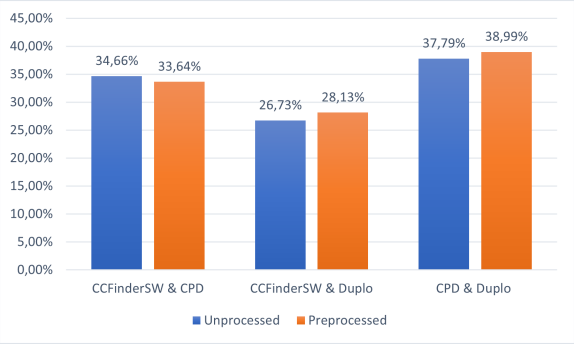
Agreement between tools

To get a better understanding of whether the tools work at least similarly, we want to look at the overlap in their results. In Figure 10, we can see the agreement between the tools. This metric is calculated by checking which lines are at least part of one clone found by both tools. This is done automatically by a script, as a manual comparison for the size of this codebase is infeasible.

The formulas used are defined as follows:



(a) First agreement metric



(b) Second agreement metric

Figure 10: Agreement between tools (for metrics, see formula 2 and 3)

$$\text{agr}_1 = \frac{\hat{c}}{\min(c_1, c_2)} \tag{2}$$

$$\text{agr}_2 = \frac{\hat{c}}{\max(c_1, c_2)}, \tag{3}$$

where agr_1 and agr_2 are the agreement scores, $\hat{c}$ is the total number of common clone lines, and c_1 is the number of clone lines for the first tools and c_2 for the second.

With the agr_2 metric, the tools can only achieve 100% agreement if they detect clones in the same lines. This means the agreement will be lowered if the number of detected lines varies strongly between the tools. The agr_1 metric is more forgiving. It takes the smaller number of detected clone lines as the denominator. Therefore, we can better see on what clone lines the tools agree.

The calculated agreement score gives us an idea of how much the tools agree on clones. It would be best to match clones between the tools and calculate an agreement score. It would give a better picture of how similar the results are, but it would also be unrealistic to evaluate in the time frame of this thesis.

Concerning the results, we can see high agreement between CCFinderSW and Duplo for both the unprocessed and the preprocessed codebase. Only CCFinderSW and CPD achieved a higher score for the preprocessed codebase. In contrast, CPD and Duplo seem to work quite differently, and preprocessing the codebase further decreases their agreement.

Example Clone

Figure 11 shows two code fragments from the RAY.FOR file in the codebase. On the left, we can see the *sum_nen* subroutine; on the right, we can see the *sum_nxy* subroutine. Each code fragment has 45 lines of code.

The colors (red and green) highlight changes, a brighter color shows lines with changes, and the darker color show, where the actual change in a line is located. Everything

that is not colored is the same from one fragment to the other. Both subroutines are identical apart from their names.

```
subroutine sum_nen(ii,ino)

      INCLUDE 'RAY_DCL.CMN'
      INTEGER NMAX(15),NXY(15,100,100)
      INTEGER NXYHU(15,100,100)
      REAL*8 XMIN(15),XMAX(15),YMIN(15),YMAX(15),DX(15),DY(15)
      REAL XPL(100),YPL(100),xymax
      REAL XHU(100,2),YHU(100,2)
      COMMON /OUT/ NXY,XMIN,XMAX,YMIN,YMAX,DX,DY,NMAX
      common /storeplot/ XPL,YPL,xymax
      COMMON/HU/NXYHU,XHU,YHU

      DO I=1,100
         YPL(I)=0.
      END DO
      XYMAX=0.

      DO I=1,100
         if(ino.eq.1)XPL(I)=XMIN(II)+(FLOAT(I)-0.5)*DX(II)
         if(ino.eq.2)XPL(I)=YMIN(II)+(FLOAT(I)-0.5)*DY(II)
         YPL(I)=0.

         DO J=1,100
            if(ino.eq.1)YPL(I)=YPL(I)+NXY(II,I,J)
            if(ino.eq.2)YPL(I)=YPL(I)+NXY(II,J,I)
         end do
         IF(YPL(I).GT.XYMAX)XYMAX=YPL(I)
      end do
      ypl(1)=0.
      ypl(100)=0.
      XYMAX=XYMAX*1.1

      SS = SPARAM(2)
      IF ((SS.EQ.10..OR.SS.EQ.12.).AND.SPARAM(28).EQ.0.) THEN
         DO I=1,100
            YHU(I,2)=0.
            DO J=1,100
               if(ino.eq.1)YHU(I,2)=YHU(I,2)+NXYHU(II,I,J)
               if(ino.eq.2)YHU(I,2)=YHU(I,2)+NXYHU(II,J,I)
            end do
         end do
      ENDIF

      return
   end
```

```
subroutine sum_nxy(ii,ino)

      INCLUDE 'RAY_DCL.CMN'
      INTEGER NMAX(15),NXY(15,100,100)
      INTEGER NXYHU(15,100,100)
      REAL*8 XMIN(15),XMAX(15),YMIN(15),YMAX(15),DX(15),DY(15)
      REAL XPL(100),YPL(100),xymax
      REAL XHU(100,2),YHU(100,2)
      COMMON /OUT/ NXY,XMIN,XMAX,YMIN,YMAX,DX,DY,NMAX
      common /storeplot/ XPL,YPL,xymax
      COMMON/HU/NXYHU,XHU,YHU

      DO I=1,100
         YPL(I)=0.
      END DO
      XYMAX=0.

      DO I=1,100
         if(ino.eq.1)XPL(I)=XMIN(II)+(FLOAT(I)-0.5)*DX(II)
         if(ino.eq.2)XPL(I)=YMIN(II)+(FLOAT(I)-0.5)*DY(II)
         YPL(I)=0.

         DO J=1,100
            if(ino.eq.1)YPL(I)=YPL(I)+NXY(II,I,J)
            if(ino.eq.2)YPL(I)=YPL(I)+NXY(II,J,I)
         end do
         IF(YPL(I).GT.XYMAX)XYMAX=YPL(I)
      end do
      ypl(1)=0.
      ypl(100)=0.
      XYMAX=XYMAX*1.1

      SS = SPARAM(2)
      IF ((SS.EQ.10..OR.SS.EQ.12.).AND.SPARAM(28).EQ.0.) THEN
         DO I=1,100
            YHU(I,2)=0.
            DO J=1,100
               if(ino.eq.1)YHU(I,2)=YHU(I,2)+NXYHU(II,I,J)
               if(ino.eq.2)YHU(I,2)=YHU(I,2)+NXYHU(II,J,I)
            end do
         end do
      ENDIF

      return
   end
```

Figure 11: Two code fragments, which all three tools identified as a clone pair

7. Evaluation

One of the motivations for this work was to see how well the current variety of code clone detection tools can be used on a real-world software project. Specifically, a scientific software project in a niche language.

As previously mentioned, we had to filter the tools quite dramatically because most of them are language-specific and do not support FORTRAN. Some tools claimed to be language-agnostic but still had quirks that made them hard to use for RAY-UI. These were filtered as well. After that, we chose five tools: NiCad, PMD CPD, SourcererCC, Duplo, and CCFinderSW for further investigation.

One concern in this investigation was the time constraint a software developer has. How much setup time can a tool take up until it is not economically viable anymore? The question we want to answer now is: How good are the results we can get from these tools in a realistic time-frame for RAY-UI?

7.1. NiCad

As we have seen in the previous section, NiCad did not present us with any code clones. The main reason behind this is the fact that NiCad makes LCS comparisons on the fragments we extract. Since we are extracting subroutines, blocks, functions, and

modules, the fragments can be immense. This means we are comparing fragments that are sometimes a few hundred lines long, and therefore we will not be able to detect clones, which are only a few lines long. The few identical or similar lines do not allow the fragments to surpass the threshold; therefore, the two clones are not considered clones as a whole.

We have seen an example clone in Figure 11, which contradicts this reasoning. It is important to note that NiCad did not parse all files in the codebase, and the example is from one that was not parsed correctly. Therefore, this reasoning is still valid.

The problem is that the subsequent smaller blocks we can easily find are if-else-condition statements and other control statements. This could lead to many tiny fragments that do not offer much value when detected as clones. This leads to the question: Could we find a different approach to extracting code fragments from our codebase?

Because of the way FORTRAN is structured, and the often deprecated programming styles used with it, we found that it would be difficult to get good results out of NiCad for the codebases we are looking at in this work.

After the significant time investment of writing a plug-in for FORTRAN and understanding TXL and NiCad to make this possible, it raises the question: Would this be economically viable for a developer team? Since we found other tools that support FORTRAN, a viable option would be to rely solely on their results and not write a plug-in. Looking at how similar the tools perform in other languages in BigCloneBench underlines this point.

The process is very straightforward as soon as you gain an understanding of TXL and NiCad, but it is time-consuming and depending on the state of the codebase takes a lot of trial and error to get working.

7.2. SourcererCC and similar

Having created great tools sadly does not help when they are hard to use and they do not offer support for users. A lot of the tools we looked at lacked documentation to some degree. SourcererCC and CCLearner have easy-to-follow guides but do not offer support when problems arise for the user, which they did in both cases.

Granted, we are looking at an edge case here, and the errors that arose likely have something to do with the FORTRAN language, but this could be viewed as an opportunity to improve the tools.

Having good documentation is critical, and sadly many teams took shortcuts, leading to hard-to-read READMEs, which are often hidden in the source structure of the project. In this scenario, scientists who develop their software will likely be unable to use these tools.

7.3. CCFinderSW, Duplo and PMD CPD

In the previous section, we looked at the results of the three tools: CCFinderSW, Duplo, and PMD CPD. We will now analyze them to answer **RQ3**; whether the

reported clones show relevance.

Unprocessed

When looking at the unprocessed codebase, which is in the original state, that is currently used and developed, we can see a very high number of clones reported by CPD. The reported clones primarily consist of one or a few comment lines and code lines. Code fragments like this, which are similar, are spread throughout the code, contributing to the high clone count of CPD.

We can gather that CPD does not filter these comments on its own, while the other tools do so, as they do not pick up on these clones. We also must remember that Duplo only considers fragments of at least three lines with our settings, which might contribute to this result.

Another factor we can look at concerning CPD's results: is the average clone length. We can see that most clones for CPD are just one line, with an average clone length of 1.32 lines. Comparing this to CCFinderSW's 8.42 and Duplo's 8.4 average clone length in lines, we can easily see that these tools handle the unprocessed codebase differently.

The other tools are closer together when it comes to the clone count, but CCFinderSW still finds significantly more clones.

From previously viewed benchmarks for other languages, we could expect CCFinderSW to perform best. And indeed, it reported the most clones when disregarding CPD's comment clones we discussed previously. In question now is the quality of the results.

Previous benchmarks showed that CCFinderX – which CCFinderSW is based on - has better results for type-3 clones than the other two tools. This could be one factor in this codebase. Slight variations in formulas are indeed present in the codebase and might only be detected by CCFinderSW.

Overall, CCFinderSW reports the most clones, whereas Duplo seems more selective. The results from CPD are less useful, as it does not filter comment lines. Filtering CCFinderSW's results might be time-consuming, but the possibility of finding more type-3 clones might be valuable for some projects. Depending on the time budget, we would choose either CCFinderSW or Duplo for the unprocessed codebase.

Preprocessed

Looking at the preprocessed codebase, we can see the results are much closer for the three tools. CCFinderSW reports the most clones, with CPD and Duplo reporting similar numbers.

It stands out that CCFinderSW reports more clones and groups more of them. With 5.32, it has the highest number of clones per class, compared to CPD with 4.59 and Duplo with 2.0. This could either mean the tool found some types of clones, which are more frequent throughout the codebase, or its threshold for differentiating/detecting clones is lower.

When looking at the clone lines reported, we see more evidence for this. CCFinderSW reports the most clone lines, with an average of 6 clones per line reported and a maximum of almost 250 lines for one line in the RAY.f90 file (see Figure 9).

We can also see that Duplo reports the longest clones. This fits the smaller number of clones reported, showing a more selective process of finding clones. It also finds fewer total clones than CPD and has more clone classes.

When looking at the agreement, we can see that CCFinderSW nearly marks all lines reported by CPD as clones, where it agrees less with the results by Duplo. This is partly because Duplo reports more clone lines overall, but it also indicates that the tools find different kinds of clones.

For the preprocessed codebase, CCFinderSW too finds the most clones and with an average length of 7.17, which is between CPD and Duplo, its results show reasonable significance.

If preprocessing is an option or if the project already has a clean codebase, CPD and Duplo might be more interesting, though, as the more selective approach, could give a more accessible starting point to refactor the codebase.

7.3.1. Agreement

We can use the agreement scores measured to get more insight into CPD's results. When comparing CCFinderSW and CPD, we know that CCFinderSW reported more clone lines. The massive decline in reported clone lines from CPD, when preprocessing the codebase, means:

- The divisor in our agr_1 metric gets smaller because CPD reports fewer clone lines after the preprocessing. The dividend stays roughly the same since we filter only comments.
- The divisor in our agr_2 is slightly reduced since CCFinderSW's reported clone lines do not diminish much. The dividend is slightly reduced as we lose some reported clone lines that overlap between the tools for the unprocessed codebase.

This shows us that a lot of the clones CPD reported for the unprocessed codebase were not reported by CCFinderSW. It further indicates that CCFinderSW returns more relevant results.

When analyzing CPD and Duplo, we see a somewhat different behavior. The agr_1 metric decreases when comparing both codebases. This means that the common clones decrease more strongly relative to the clone line count. This indicates that some actual clones now fall below the threshold for CPD and don't get reported anymore.

8. Discussion

When doing this kind of research, the apparent difficulty is validating and analyzing the data acquired. These tools output vast amounts of data, with many clones reported by

each. Manually checking the results is infeasible, and automatic analysis (with scripts) will always lack in-depth analysis. This thesis will only be able to introduce the field of CCD and an overview of the challenges that arise while using CCD tools.

Checking the reports for correctness can only be done via validating samples. We can at least tell the results are not random. We can see that the results can even be helpful with the example we have shown in the 6 section. A clone of this size indicates a problem in the code, which should be investigated. We checked other samples during validation, and all showed that the tools reported real clones. Since this constitutes only a fraction of the reported clones, we could have easily missed false positives or otherwise useless results.

With the unprocessed codebase, we see diverse results from the tools, making it difficult to evaluate the quality of these results. But, comparing this to the preprocessed codebase, we can see a difference in behavior from CPD to CCFinderSW and Duplo – mainly CPD's lack of comment filtering.

We showed that CCFinderSW, Duplo, and PMD CPD could be used for the codebase. The results find significant clones, and with some preprocessing, we can further assist the tools' functionality, to focus on the essential parts of the code.

How useful it is to have a tool report roughly seven times the amount of clone lines, as there are lines in the codebase is questionable and should be investigated further. Here it might be better to increase the similarity threshold or the minimum clone size in the tools configurations to avoid information overload for developers.

Experimenting further with the tool configurations would have been interesting to find out how they impact the results. In future work, we could improve the results and filter the reported clones to the most relevant ones. The sheer number of clones the tools find may take a long time to go through.

Further, investigating the results concerning the types of clones detected would give more insight into the quality of the results. It would also be interesting if the tools find the same clones rather than if they report the same clone lines.

For cross-checking results from tool to tool, three is enough to have some confidence in the results. Other studies opted for higher tool counts, but the lack of support made it difficult to get more to run on our codebase. The tools we chose to compare the results were the only ones that worked on RAY-UI, with realistic time investment.

Getting NiCad to run would have been a great addition to the results, as it is an established tool in this field. But because of the way NiCad works, it was not an excellent tool for this codebase. At least for the approach we used in writing the plug-in. The very long code blocks lead to only a few code fragment comparisons, which leads to no clones found by the tool. Using inter-fragment comparisons or better ways to extract code fragments from the code would be needed to get good results.

9. Conclusion

In this thesis, we looked at many tools to find one we could use for CCD for RAY-UI. Further, we looked at six (NiCad, CPD, Duplo, PMD CPD, SourcererCC and

CCLearner) in detail to understand the process of CCD and why it was so challenging to get tools to work for our codebase. We did this to answer our first research question (**RQ1**): Are there differences between scientific software and common software that handicap the CCD tools?

The first obvious problem is that FORTRAN is a niche language, and most developers of these tools opt to support more popular languages. Scientific software often has niche applications and goals, which makes developers more likely to choose these languages.

Another problem we see in RAY-UI's codebase is the length of functions or subroutines, uncommon use of syntax, and nonstandardized formatting. The code is closely designed after the formulas and the physics behind the simulations. This is a design that a developer from a computer science background likely iterates on to improve modularity and readability. The different backgrounds lead to different ways of thinking, which could impact CCD tools. We experienced this with NiCad. Its strategy for detecting clones makes it challenging to apply to our codebase.

We had a particular goal in mind for investigating these tools. They need to perform well on a (probably difficult to parse) codebase that is manageable in size (22 Thousand Lines of Code (TLOC)). Therefore, we are not looking for the most efficient but effective tool. From a developer standpoint, we do not care too much about the runtime as long as it does not take longer than a few minutes or even hours.

As mentioned previously, we focused on three research questions. We want to answer **RQ2** here. The first problem we want to discuss here is the lack of support for FORTRAN or, in general, niche languages by these tools. Supporting more languages means more work for a language-specific tool. Still, even the tools, which claimed to be language-agnostic, had requirements that made them unusable for the RAY-UI codebase. Finding tools that worked was time-consuming, with little support or documentation on expanding language-specific tools or getting language-agnostic tools to run on the RAY-UI codebase.

With the six tools we looked at in detail, we learned the following lessons:

- Writing a language Plug-In for NiCad takes a long time and much trial and error. This kind of work is likely only economically viable for some developer teams.
- Getting CCD tools to run mostly failed due to their language support and often due to the lack of proper documentation or useable error messages.
- PMD CPD is an easy plug-and-play solution to find code clones in a FORTRAN codebase, and it might be the best option to get quick results.
- Duplo and CCFinderSW took a bit more time to set up but reported higher quantities of clones.

Many user experience improvements are needed to expand to other users without primary computer science and software development expertise. Even more so, as they could use these tools the most to write good software.

Concerning **RQ3**, we can confidently say that CCFinderSW, CPD, and Duplo all find relevant results. The metrics we analyzed indicate that the reported clones are

not random and do not only consist of a few lines. The samples we looked at also back up our assumption that these results are relevant.

Finding the right tools is complex, and it takes a lot of sifting through papers and GitHub repositories. A few great surveys list many of the tools used, but knowing if the listed tool would support your codebase takes a few more steps. The last is to try the tool out and see if it runs.

There is extensive research in the field, and we can find many good tools that detect type-1, type-2, and some type-3 clones. Teams interested in refactoring their code to make it more maintainable will find a tool to suit their needs, even if it might take some research.

References

[Ada92] Jeanne C. Adams, editor. *Fortran 90 handbook: complete ANSI/ISO reference*. Intertext Publications : McGraw-Hill Book Co, New York, 1992.

[Bak95] B.S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 86–95, Toronto, Ont., Canada, 1995. IEEE Comput. Soc. Press.

[BD18] Brent van Bladel and Serge Demeyer. Test behaviour detection as a test refactoring safety. In *Proceedings of the 2nd International Workshop on Refactoring*, pages 22–25, Montpellier France, September 2018. ACM.

[BYM+98] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. page 11, 1998.

[CR11] James R. Cordy and Chanchal K. Roy. The NiCad Clone Detector. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 219–220, Kingston, ON, Canada, June 2011. IEEE.

[DPS05] F. Deissenboeck, M. Pizka, and T. Seifert. Tool Support for Continuous Quality Assessment. In *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)*, pages 127–136, September 2005.

[GK09] Nils Göde and Rainer Koschke. Incremental Clone Detection. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 219–228, Kaiserslautern, Germany, 2009. IEEE.

[HMS+09] Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. How do scientists develop and use scientific software? In *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8, Vancouver, BC, Canada, May 2009. IEEE.

[JDHW09] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do Code Clones Matter? In *2009 IEEE 31st International Conference on Software Engineering*, pages 485–495, 2009. arXiv:1701.05472 [cs].

[KB14] Upulee Kanewala and James M. Bieman. Testing Scientific Software: A Systematic Literature Review. *Information and software technology*, 56(10):1219–1232, October 2014.

[KKI02] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002. Conference Name: IEEE Transactions on Software Engineering.

[LCHY06] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '06*, page 872, Philadelphia, PA, USA, 2006. ACM Press.

[LFZ+17] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. CCLearner: A Deep Learning-Based Clone Detection Approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260, Shanghai, September 2017. IEEE.

[Lid22] Daniel Lidström. Duplo (C/C++/Java Duplicate Source Code Block Finder), September 2022. original-date: 2010-09-21T16:36:08Z.

[LLMZ06] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, March 2006. Conference Name: IEEE Transactions on Software Engineering.

[MHH+12] Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Folding Repeated Instructions for Improving Token-Based Code Clone Detection. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 64–73, Riva del Garda, Italy, September 2012. IEEE.

[MHH+13] Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Gapped code clone detection with lightweight source code analysis. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 93–102, May 2013. ISSN: 1092-8138.

[RC08] C.K. Roy and J.R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *2008 16th IEEE International Conference on Program Comprehension*, pages 172–181, Amsterdam, June 2008. IEEE.

[RCK09] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, May 2009.

[Sch08] Franz Schäfers. The BESSY Raytrace Program RAY. In Alexei Erko, Mourad Idir, Thomas Krist, and Alan G. Michette, editors, *Modern Developments in X-Ray and Neutron Optics*, volume 137, pages 9–41. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. Series Title: Springer Series in optical science.

[SIK+14] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480, Victoria, BC, Canada, September 2014. IEEE.

[SM08] Judith Segal and Chris Morris. Developing Scientific Software. *IEEE Software*, 25(4):18–20, July 2008.

[SSC16] Dan Szymczak, Spencer Smith, and Jacques Carette. A knowledge-based approach to scientific software development: position paper. In *Proceedings of the International Workshop on Software Engineering for Science*, pages 23–26, Austin Texas, May 2016. ACM.

[SSS+16] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. SourcererCC: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1157–1168, Austin Texas, May 2016. ACM.

[SYCI17] Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. CCFinderSW: Clone Detection Tool with Flexible Multilingual Tokenization. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 654–659, Nanjing, December 2017. IEEE.

[TDD22] Copeland Tom, Dixon-Peugh David, and Craine David. PMD - source code analyzer, September 2022. original-date: 2012-07-11T18:03:00Z.

[vBD20] Brent van Bladel and Serge Demeyer. Clone Detection in Test Code: An Empirical Evaluation. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 492–500, London, ON, Canada, February 2020. IEEE.

[WELL10] Anna Wingkvist, Morgan Ericsson, Rudiger Lincke, and Welf Lowe. A Metrics-Based Approach to Technical Documentation Quality. In *2010 Seventh International Conference on the Quality of Information and Communications Technology*, pages 476–481, Porto, Portugal, September 2010. IEEE.

[YG12] Yang Yuan and Yao Guo. Boreas: an accurate and scalable token-based approach to code clone detection. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, page 286, Essen, Germany, 2012. ACM Press.

A. Appendix

Agreement per file

For the unprocessed code base

In table 3, 4 and 5 are the results of the agreement comparisons per file and per two tools.

Table 3: Agreement between CCFinderSW and CPD

File	Agreement	CCFinderSW Clone Lines [%]	CPD Clone Lines [%]
mod_raycore_Paul.f90	88,89	67,10	36,31
RAY.f90	100,00	60,00	60,00
gnu_routines.f90	0,00	51,46	1,99
rayxml.f90	94,29	59,76	31,09
REFLEC_IN.f90	96,43	67,76	9,53
OEINPUT.f90	43,21	53,61	14,33
CRYSUB.f90	82,50	70,72	17,35
REFLEC.f90	95,90	73,01	26,58
RAYLIB.f90	59,88	40,43	9,71
OPTICS.f90	76,92	51,20	15,60
mod_raycore.f90	0,00	42,01	0,36
SOURCE.f90	95,15	57,13	17,42
OPTCON.f90	100,00	0,00	0,00
REFLEC_OUT.f90	68,97	44,94	10,87
REFLEC_CAL.f90	100,00	64,37	37,93

For the preprocessed code base

In table 6, 7 and 8 are the results of the agreement comparisons per file and per two tools.

Table 4: Agreement between CCFinderSW and CPD for the preprocessed code base

File	Agreement	CCFinderSW Clone Lines [%]	Duplo Clone Lines [%]
mod_raycore_Paul.f90	91,79	67,10	26,97
RAY.f90	93,94	60,00	83,64
gnu_routines.f90	76,47	51,46	4,51
rayxml.f90	94,18	59,76	13,41
REFLEC_IN.f90	52,94	67,76	1,93
OEINPUT.f90	67,88	53,61	5,44
CRYSUB.f90	84,38	70,72	6,94
REFLEC.f90	90,82	73,01	33,93
RAYLIB.f90	46,67	40,43	1,75
OPTICS.f90	50,00	51,20	3,20
mod_raycore.f90	83,56	42,01	6,55
SOURCE.f90	86,75	57,13	8,76
OPTCON.f90	100,00	0,00	0,00
REFLEC_OUT.f90	80,06	44,94	12,95
REFLEC_CAL.f90	67,39	64,37	52,87

Table 5: Agreement between CCFinderSW and CPD for the preprocessed code base

File	Agreement	CPD Clone Lines [%]	Duplo Clone Lines [%]
mod_raycore_Paul.f90	82,55	36,31	26,97
RAY.f90	93,94	60,00	83,64
gnu_routines.f90	26,67	1,99	4,51
rayxml.f90	76,72	31,09	13,41
REFLEC_IN.f90	23,53	9,53	1,93
OEINPUT.f90	66,42	14,33	5,44
CRYSUB.f90	75,00	17,35	6,94
REFLEC.f90	76,55	26,58	33,93
RAYLIB.f90	46,67	9,71	1,75
OPTICS.f90	100,00	15,60	3,20
mod_raycore.f90	0,00	0,36	6,55
SOURCE.f90	87,95	17,42	8,76
OPTCON.f90	100,00	0,00	0,00
REFLEC_OUT.f90	39,08	10,87	12,95
REFLEC_CAL.f90	93,94	37,93	52,87

Table 6: Agreement between CCFinderSW and CPD

File	Agreement	CCFinderSW Clone Lines [%]	CPD Clone Lines [%]
mod_raycore_Paul.f90	100,00	84,62	84,62
RAY.f90	91,36	67,39	40,76
gnu_routines.f90	100,00	48,08	0,88
rayxml.f90	82,50	72,48	17,90
REFLEC_IN.f90	81,13	58,43	2,43
OEINPUT.f90	93,48	66,67	6,23
CRYSUB.f90	100,00	48,23	6,01
REFLEC.f90	100,00	0,00	0,00
RAYLIB.f90	100,00	59,37	22,48
OPTICS.f90	97,34	71,10	38,69
mod_raycore.f90	100,00	80,00	47,14
SOURCE.f90	100,00	34,96	2,44
OPTCON.f90	90,48	40,12	15,27
REFLEC_OUT.f90	0,00	58,12	0,00
REFLEC_CAL.f90	92,86	41,38	5,07

Table 7: Agreement between CCFinderSW and CPD for the preprocessed code base

File	Agreement	CCFinderSW Clone Lines [%]	Duplo Clone Lines [%]
mod_raycore_Paul.f90	100,00	84,62	71,79
RAY.f90	94,84	67,39	28,51
gnu_routines.f90	77,25	48,08	14,01
rayxml.f90	84,38	72,48	7,16
REFLEC_IN.f90	80,00	58,43	3,44
OEINPUT.f90	52,94	66,67	2,30
CRYSUB.f90	81,82	48,23	8,47
REFLEC.f90	100,00	0,00	0,00
RAYLIB.f90	92,14	59,37	13,45
OPTICS.f90	91,85	71,10	37,38
mod_raycore.f90	100,00	80,00	40,00
SOURCE.f90	0,00	34,96	0,00
OPTCON.f90	71,43	40,12	8,48
REFLEC_OUT.f90	100,00	58,12	5,10
REFLEC_CAL.f90	40,00	41,38	2,17

Table 8: Agreement between CCFinderSW and CPD for the preprocessed code base

File	Agreement	CPD Clone Lines [%]	Duplo Clone Lines [%]
mod_raycore_Paul.f90	100,00	84,62	71,79
RAY.f90	91,75	40,76	28,51
gnu_routines.f90	62,50	0,88	14,01
rayxml.f90	75,00	17,90	7,16
REFLEC_IN.f90	41,51	2,43	3,44
OEINPUT.f90	0,00	6,23	2,30
CRYSUB.f90	20,51	6,01	8,47
REFLEC.f90	100,00	0,00	0,00
RAYLIB.f90	68,57	22,48	13,45
OPTICS.f90	79,57	38,69	37,38
mod_raycore.f90	100,00	47,14	40,00
SOURCE.f90	0,00	2,44	0,00
OPTCON.f90	68,57	15,27	8,48
REFLEC_OUT.f90	0,00	0,00	32,00
REFLEC_CAL.f90	40,00	5,07	2,17

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den January 9, 2023 .....