Token-based Plagiarism Detection for Statecharts

Bachelor's Thesis of

Jonas Strittmatter

at the Department of Informatics
KASTEL – Institute of Information Security and Dependability

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Prof. Dr.-Ing. Anne Koziolek
Advisor:	M.Sc. Timur Sağlam
Second advisor:	M.Sc. Jan Wittler

28. November 2022 – 5. April 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have submitted neither parts of nor the complete thesis as an examination elsewhere. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. This also applies to figures, sketches, images and similar depictions, as well as sources from the internet.

Karlsruhe, April 5, 2023

. . . .

(Jonas Strittmatter)

Abstract

In the field of software engineering, existing plagiarism detection systems have primarily focused on detecting cases of plagiarism in code. However, other artifacts, such as models, also play a crucial role in the development process. Statecharts, in particular, are used to model the behavior of a system. This thesis investigates the applicability and challenges of applying token-based plagiarism detection systems to statecharts. We extend the plagiarism detector JPlag to support detecting cases of plagiarism in statecharts. Our approach is evaluated using a dataset of student assignments from a modeling course, where we generate plagiarized statecharts by adopting common obfuscation attacks. We study the effects of the token-extraction strategy, sorting techniques, and the minimum token match parameter. The results suggest that an approach tailored to the specific kind of model, such as statecharts, works better than a generic solution for models.

Zusammenfassung

Im Bereich der Softwaretechnik haben existierende Plagiatserkennungssysteme hauptsächlich darauf abgezielt, Fälle von Plagiaten in Code zu erkennen. Allerdings spielen auch andere Artefakte, wie Modelle, eine entscheidende Rolle im Entwicklungsprozess. Insbesondere werden Zustandsdiagramme verwendet, um das Verhalten eines Systems zu modellieren. Diese Arbeit untersucht die Anwendbarkeit und Herausforderungen bei der Anwendung von Token-basierten Plagiatserkennungssystemen auf Zustandsdiagramme. Wir erweitern das Plagiatserkennungssystem JPlag, um die Erkennung von Plagiatsfällen in Zustandsdiagrammen zu unterstützen. Unser Ansatz wird anhand eines Datensatzes bestehend aus studentischen Abgaben aus einem Modellierungskurs evaluiert, wobei wir durch Anwendung gängiger Verschleierungsmethoden plagiierte Zustandsdiagramme generieren. Wir untersuchen die Auswirkungen der Token-Extraktionsstrategie, Sortierungstechniken und der minimalen Token-Länge. Die Ergebnisse legen nahe, dass ein auf die spezifische Art von Modell, wie Zustandsdiagramme, zugeschnittener Ansatz besser funktioniert als eine generische Lösung für Modelle.

Contents

1. Introduction

Software plagiarism is an issue that many academics have to deal with when teaching programming courses [11]. With an increasing number of assignments the task of detecting plagiarism by hand quickly becomes unfeasible. Numerous tools have been created over the years that are able to detect possible cases of plagiarism. Currently, the majority of existing plagiarism detection systems work directly on the source code of students' submissions.

In recent years, a different approach in software development has gained popularity: *model-driven software engineering* (MDSE). Models can be thought of as an abstraction of reality. A model can emerge from the observation of multiple objects that share some commonalities. It reduces the set of properties of the original object which makes it easier to reason about it [9].

Statecharts, as a specific kind of models, are visual formalisms to represent the behavior of complex systems. As such, they are applied across different industries such as the automotive, aerospace or telecommunications industries [34, 8, 22]. In academia, statecharts are used to teach MDSE. For example, students are asked to create statecharts from natural language requirements to model a system. The statecharts are then used to generate executable code that simulates the behavior of the system [10].

Established token-based plagiarism detection systems for programming languages such as Java or C++ have been in use for many years. In contrast, the application of the same techniques in the realm of models has not been well-studied and extensively evaluated in practice. For instance, the popular plagiarism detector JPlag only recently added support for metamodels [28].

In this thesis we extend JPlag with a new language module for statecharts. Since there are multiple formats to specify statecharts, each focusing on different aspects, we develop two independent language modules. The first module supports statecharts in the format used by *itemis CREATE* [3], a toolkit for creating and simulating statecharts. The second module can be applied to statecharts in a more general XML-based format (SCXML). For each module, two token extraction strategies with different levels of granularity are developed and compared.

Obfuscation attacks are a type of plagiarism technique in which an individual modifies a submission in such a way as to hide the fact that the work was copied from someone else. The goal is to evade detection by plagiarism detection tools. We adapt common obfuscation attacks to be applied to statecharts and develop a program that can automatically generate modified statecharts from existing ones.

This program is applied to a real-world dataset of statecharts to generate a set of plagiarized statecharts. Using these plagiarized statecharts, the ability of both language modules to detect plagiarisms is compared to that of an existing language module that supports general instances of models created with the *Eclipse Modeling Framework* (EMF)

[27]. It is demonstrated that our specialized language modules perform better than a generic approach for models. Further, the importance of sorting of statechart elements is discussed as a measure against certain obfuscation attacks.

2. Foundations

2.1. Plagiarism

Plagiarism is commonly defined as the act of copying someone else's work and presenting it as one's own without giving credit to the original author [19]. Plagiarized programs represent a particular point of interest. Novak et al. analyzed 150 papers on plagiarism detection and found out that the most used definition on what constitutes plagiarism in software is the following: "A plagiarized program can defined as a program which has been produced from another program with a small number of routine transformations." [24]

2.2. Plagiarism Detection

Among the widely used plagiarism detection tools today are Moss [5, 35] and JPlag [18]. This thesis shall focus on JPlag for the simple reason that it is an established and open-source program. The source code of Moss, on the other hand, is not accessible to the public and can only be used as a web-service hosted on Stanford servers, making it challenging to contribute our work implemented in this thesis.

2.2.1. JPlag

The following section provides a brief overview of some of the characteristics of JPlag relevant for this thesis. JPlag has been developed in 1995 at the University of Karlsruhe [25]. In 2000 when the first technical report on JPlag was written, only four programming languages were supported: Java, C, C++ and Scheme [26]. Today, JPlag supports plagiarism detection for many more languages (albeit with varying levels of maturity). Recently (in June 2022), a language module for EMF metamodels has been added [28].

JPlag works in four main steps: first, the language module for the selected language parses (for languages such as Java) or scans (for languages such as C++) the input programs. Existing language modules are typically structured as seen in Figure 2.1. Next, an *abstract syntax tree* (AST) is created that describes the structure of a program.

The *Parser* class takes in a set of files that it parses and turns them into a list of tokens which form an abstraction from the underlying language. In the case of the Java language module, a statement such as *if (s == null) return -1;* is turned into the three tokens *J_IF_BEGIN*, *J_RETURN* and *J_IF_END*. This step entails filtering out certain semantical information such as types of variables or method signatures.

As a last step, to figure out the similarity between two programs, a pair-wise comparison of the two token streams is performed. JPlag uses an optimized version of Michael Wise's

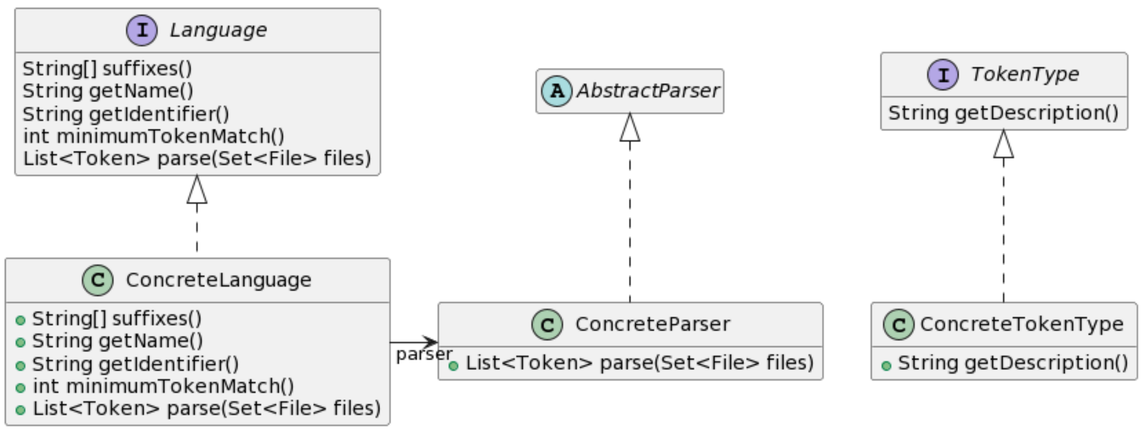


Figure 2.1.: The basic structure of a JPlag language module

Greedy String Tiling algorithm [36]. The algorithm outputs a list of matching sections found within the two token streams with the additional constraint that the matches must have a minimal length (the following subsection discusses this parameter).

The final output is a similarity score that depends on the number of tokens matching. Additionally, JPlag can show the matching sections side by side. This makes it easy for reviewers to confirm cases of plagiarism.

2.2.2. Minimum Token Match

Minimum token match (also sometimes referred to as minimum matching token length) is an important parameter in JPlag that can be set via the command line parameter *t*. When JPlag internally compares the token streams of two submissions, only sections of at least this size contribute to an increase in the similarity value of the submission pair. It allows users to control the sensitivity of the system for plagiarism.

There is no optimal value for this parameter, as it highly depends on the selected language, the granularity of the token set that is used and the structure of the submissions (e.g. in the case of models such as statecharts the size of the reference lists, or in the case of Java code the size of method blocks). A value that is too small leads to many false positives since almost all sections can be found in both submissions resulting in a high similarity value. Conversely, choosing a value that is too large results in many false negatives as there is a smaller amount of matching sections within submission pairs.

2.3. Statecharts

Harel statecharts [16] (created by the computer scientist David Harel) are a commonly used formalism for describing complex reactive systems. They were created as an extension of state diagrams by introducing concepts such as super-states, orthogonality and actions which make it possible to model hierarchies, concurrency and communication within a system.

A slightly modified version of statecharts has since become part of the *Unified Modeling Language* (UML) specification [1].

The most fundamental statechart concepts are states, transitions and events. A state represents the condition of an object or process. When an action occurs in the world, an event may be triggered that causes a transition to be taken, subsequently changing the active state. A transition is only possible if all preconditions, called guards, are satisfied. Regions are used to group together a set of states and transitions.

2.3.1. Itemis CREATE

Itemis CREATE [4] (formerly YAKINDU Statechart Tools), which shall be referred to as *Create* in this thesis, has been developed by the German consulting company itemis AG that specializes in model-driven development [3]. The software is based on the development environment Eclipse and contains a visual editor for creating statecharts as well as a simulation engine. It also provides a code generation feature that allows users to turn a model into executable code for e.g. the Java, C or C++ programming languages.

Figure 2.2 shows an example of a statechart created with Create. The featured statechart demonstrates several important statechart elements such as states, transitions and events. In the definition section of the statechart there is an interface *user* with a *press_button* event as well as an integer variable *t*. As the event *user.press_button* is fired, the statechart transitions from the initial state *Start* to the composite state *Blinking*. The initial state of this composite state is a so-called shallow history state. That means the state will remember in what substate it was in when it is being exited and resume from this state upon reentering.

Every time the *Blinking* state is entered, the code following the *entry* directive is executed. Similarly, there is an *exit* directive that is run when the state is left. In the example, the variable *t* is reset to zero. The blinking state contains the two states *Light* and Dark that it alternates between every second. Once the *Light* state has been entered five times without exiting, the graph reenters the *Start* state. This is modeled by using the guard condition *[t == 5]*.

2.3.2. SCXML

SCXML, short for State Chart extensible Markup Language, is a language created for specifying state machines that has been standardized by the World Wide Web Consortium (W3C) [31]. It is XML-based and can therefore be parsed with existing XML parsers. There are several tools that can be used to create statecharts in the SCXML format. An example includes the statechart editor that is part of the cross-platform software development platform Qt [13].

Using Create, a statechart created in the .ysc format can be exported to SCXML.

The remainder of the section provides an overview of how some fundamental statechart elements are represented in SCXML (using the concrete example of the statechart in Figure 2.2).

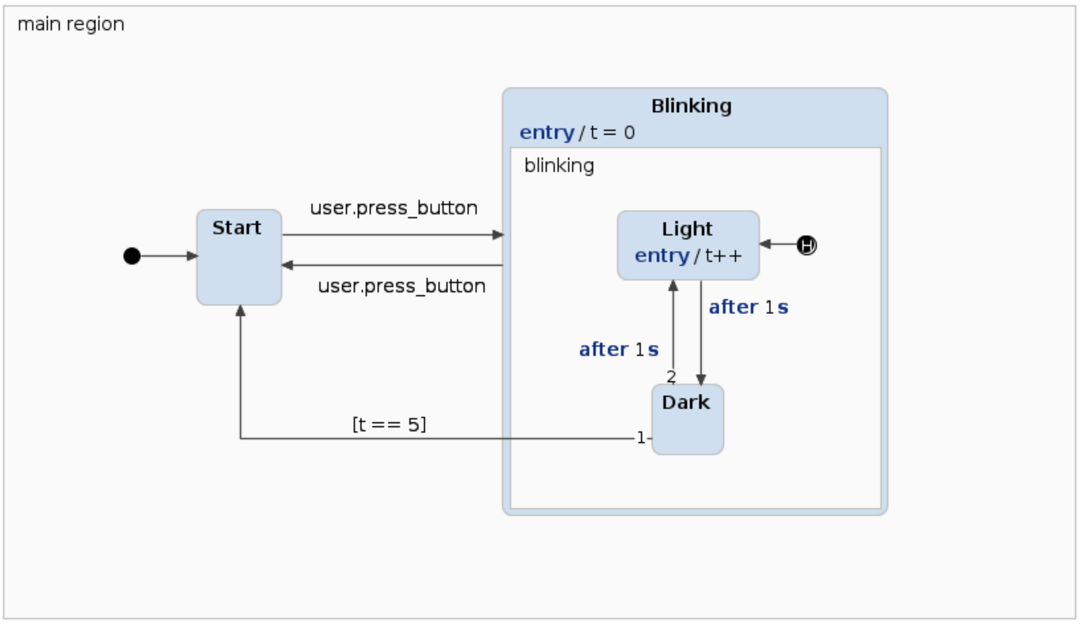


Figure 2.2.: An example statechart created with Create

States and transitions

The following SCXML code is generated for the transition from the Start state to the Blinking state:

```
<state id="Start">
  <transition event="user.press_button" target="Blinking">
  </transition>
</state>
```

The *<initial>* element is used to denote the initial state of a region / state. Regions do not exist as separate entities in SCXML, they are instead represented using nested *<state>* elements.

```
<state id="main_region">
  <initial>
    <transition target="Start" type="internal">
    </transition>
  </initial>
  ...
```

It is possible to execute some operations when entering or exiting a state or upon transitioning from one state to another as part of the transition. The statement *t++* is converted to an *assign* element:

```
<onentry>
  <assign location="t" expr="t + 1"/>
</onentry>
```

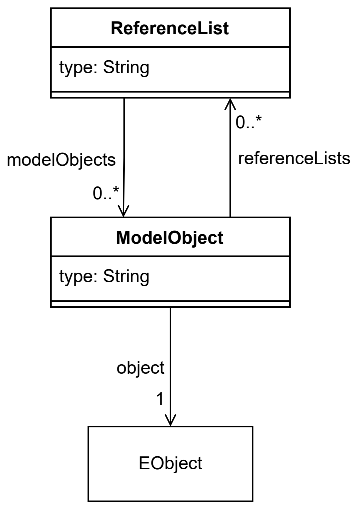


Figure 2.3.: The model object as an abstraction on top of EObjects

Transitions can also contain guard conditions as a separate attribute. Here is the SCXML code for the *t == 5* condition from the example:

```
<transition cond="t == 5" target="Start">
</transition>
```

Executable content

Executable content is used to modify a statechart's data model or to interact with external entities. The data model enables the modification of the statechart's internal state by reading or writing to variables. The SCXML specification does not specify any particular language that must be used for simple expressions or *<script>* elements (see the *Data Models* section in [31]). This decision is instead left to the implementation. For example, Create uses a custom statechart language.

Other executable content includes *raise* to raise events, *if* / *else* or *elseif* to execute actions conditionally, *script* to run script code, or *assign* to modify the data model.

Terminology

In this section we introduce some terminology that is used in the thesis. Figure 2.3 shows an abstraction on top of the *EObject* class (and its subclasses) used in EMF. We shall refer to any *EObject* within the containment tree of an instance of the statechart model as a *model object*. A model object can contain any number of *reference lists*, each of which contains a list of other model objects. If a model object contains at least one reference list, it shall be referred to as *nested*.

3. Related Work

Systems for detecting plagiarisms have been around for at least three decades. At the beginning, rather simple techniques that compared certain metrics of the input programs (e.g. the number of lines or the number of *if* statements) [24] were used. Later, the focus shifted towards approaches that look at the structure of the programs. A method used for detecting plagiarisms between Pascal programs developed by Jankowitz parses the input files and creates two static execution trees that represent the interconnections between different procedures of each program. These trees are then checked for similarities between different branches [17].

Today, most plagiarism detectors work in a similar way by first parsing the program into an intermediate representation. All these plagiarism detectors work directly on the source code of the programs.

The first work specially created for the task of detecting plagiarisms in the domain of modeling is described in a paper by Martínez et al. [20]. Their approach is based on a modified version of the Locality Sensitive Hashing (LSH) technique [6]. Given a set of n points $P = p_1, \dots, p_n$ and a point $q \in X$ in a metric space Q, LSH approximates the nearest neighbor $p \in P$ of q.

Martínez et al. apply LSH to a repository of models in order to group them into a series of buckets in a process called clustering. Models that are similar to each other are mapped to the same bucket. Their approach is computationally efficient because no pairwise comparisons of model instances takes place.

3.1. Clone Detection

Clone detection [12] is related to the field of plagiarism detection. It deals with finding model fragments that are similar with respect to some definition of similarity.

While both plagiarism detection and clone detection for models can be used to detect duplicate or highly similar parts of models, clone detection looks at the problem from a software engineering standpoint. When it is known where duplicate parts are present in a model, it can indicate potential ways for refactoring in order to improve the quality of the software [7].

Recent years have seen some contributions to the field of clone detection for models, most notably by Babur et al. In their paper, the authors extend the statistical analysis of models (SAMOS) framework, in order to facilitate clone detection of (meta-)models, in particular Ecore models created with EMF. They use the type information and names of the metamodel identifiers and apply sophisticated techniques such as natural language processing (NLP) in order to cluster the models based on their similarity [7].

Approaches towards plagiarism detection can be assigned to two kinds: lexical and structural.

Lexical or linguistic approaches include techniques based on NLP [15] that is employed to calculate similarities between statecharts based on attributes such as state names or transitions. Inter-semantic similarity measures are used to compute similarity measures across different graphs [2] that can also be applied to statecharts.

Nejati et al. [22] propose a Match operator for statecharts. This operator can be used to find correspondences between individual states that are part of a pair of input statecharts. The method uses a combination of static (lexical) and behavioral heuristics to measure similarities. The static approach looks at attributes such as element names while the behavioral approach assigns a similarity value to two states depending on how similarly they behave. Although not discussed in their paper, their research has the potential to be employed for plagiarism identification by computing matching sections between statecharts.

Regarding structural approaches, one possibility is to consider a statecharts as a directed graph. A metric such as the graph edit distance (GED) can then be used to compute a similarity score between two graphs. The GED was originally formalized by Sanfeliu et al. [29] in 1983. It is calculated based on the number of modifications required to transform one input graph into another.

Clone detection approaches cannot be directly employed to detect plagiarisms because an attacker can attempt to conceal plagiarism. In the domain of statecharts, this may be achieved by changing attributes such as names of statechart elements. Creating plagiarized models in this way can be performed fully automatically (see section 6.2). This makes lexical similarity measures less useful in the area of plagiarism detection.

Sağlam et al. are the first to propose a token-based plagiarism detection system for EMF models [28]. A language module for generic EMF model instances has since been added to JPlag [27] though there is no published research on this approach yet. This language module works by first reading in one or more .ecore files that are the metamodels to the input models. The token set is dynamically constructed based on the *EClass* of the encountered *EObjects* in the containment tree of the model instances.

4. Analysis

4.1. Differences between Models and Code

Existing token-based plagiarism detection systems such as JPlag mainly support plagiarism detection for code. It is therefore useful to discuss differences between models and code. The aim is to gain an understanding for potential issues before implementing a token-based system for statecharts.

One inherent difference is that code generally has a linear structure, while models or model instances generally adhere to a more nested and arguably more complex structure. For example, a statechart can be represented as a directed graph where the root is the whole statechart and the nodes are child elements such as states. Mapping this structure to a one-dimensional token stream as used in token-based systems inevitably involves a loss of information. In code, each statement typically maps to exactly one token where the token type is often determined by the grammar of the language. For models, there is no such obvious way to extract tokens since each model object may contain several attributes and references to other model objects.

Another aspect is that specific kinds of models such as statecharts can be defined and stored in different ways. These various notations include SCXML or the format used by Create depending on the application. This makes it harder to compare statecharts across multiple formats. While one format can be converted into another, this is not always possible since some concepts are unique to a specific application such as Create.

There are many semantically equivalent ways to specify the textual representation of a statechart by simply reordering elements on the same levels of the containment tree. In contrast, reordering lines of code cannot be achieved as easily since the order of statements is often important for the semantics of the code.

This difference between models and code has to be accounted for when extracting tokens. If the tokens are simply extracted in the same order as the model objects are defined in the input file, the resulting token streams for each permutation differ significantly.

4.2. Applying Obfuscation Attacks to Statecharts

Obfuscation attacks in the context of plagiarism detection refer to the deliberate modification or manipulation of software artefacts where the specific aim is to conceal the act of plagiarism. A successful obfuscation attack exists if the similarity of the original submission and the plagiarized submission are sufficiently low to avoid detection of the presence of plagiarism (e.g., by a human).

A common classification scheme for model clones defined in the literature [7, 33], is the following:

- **Type A (exact)**: identical except for secondary notation and internal modifiers
- **Type B (modified)**: small changes to names or attributes plus a small amount of insertion / deletion of model parts
- **Type C (renamed)**: large amount changes to names or attributes plus insertion / deletion of model parts
- **Type D (semantic)**: semantically equivalent models (they may have a completely different structure but behave the same as the original model)

For types A, B and C the distinction between few and many changes to the attributes (renaming attacks) is less relevant for this work since token-based systems are inherently resilient against attacks that do not change the structure of the models. For the sake of completeness, we still take a look at the *rename* attack in the evaluation (chapter 6). Type D clones are also not considered here as it is expected that all statecharts within each dataset behave similarly since they are created as part of the same task. Common obfuscation attacks that cover this classification include renaming, insertion, deletion, moving and swapping of model elements [28, 7]. The attacks can be further subdivided to only consider certain types of elements within the model.

However, models often have certain constraints that need to be fulfilled for the model to be considered valid. This imposes restrictions on the types of obfuscation attacks that can be applied.

For example, a transition in a statechart can only be inserted into compatible reference lists, and not into any other parts of the model. Apart from structural constraints, if the semantics of the model are to be validated as well, there are even more conditions that apply to the kinds of attacks that can be performed. Of course there are multiple definitions of what constitutes an invalid statechart. Create, for instance, enforces that each state within the statechart must be reachable (i.e. there has to exist at least one incoming transition to this state).

More precisely, Create performs a series of checks to validate a statechart as the statechart diagram is edited or before conversions to other formats [37]. The error messages are defined in the STextValidationMessage class while the actual tests are performed in STextValidator.

Some attacks, like the *move* attack (which Babur et al. define as “simply moving a model element elsewhere” [7] in their work on clone detection for metamodels) can be seen as a combined attack of a deletion plus an insertion operation. Thus, the same restrictions as for the individual attacks apply. In our work we shall define the *move* operation slightly differently, namely as an operation that moves a model object within the *same* reference list. This makes the attack more restrictive than a regular *move* attack or the similar *moveSimilarContainer* proposed by Babur et al. An implication of our definition is that a *move* attack does not change the semantics of the statechart at all since only the order of model elements on the same level of the containment tree is changed (see the discussion in section 4.1 on the properties of models).

In SCXML, multiple *<onentry>* or *<onexit>* elements may be present in an element such as *<state>* [31] (W3C SCXML specification, section 3.3). However, there is no functional difference between a state that has a single *onentry* element versus a state with several *onentry* elements, provided that the combined executable contents in all of them are the same. An attacker could use this property to change the token sequence by changing the number of existing *onentry* or *onexit* elements. A token-based system should therefore account for this possibility by extracting the same tokens in all cases.

The above considerations are important when evaluating the resistance of plagiarism detection systems for statecharts against obfuscation attacks. As such, they are again referenced in the evaluation section (section 6.2) when discussing the automatic generation of plagiarized statecharts.

4.3. Immunity Against Selected Obfuscation Attacks

For code, token-based plagiarism detection systems are inherently immune against certain obfuscation attacks such as renaming of variables. This is due to the fact that the token types only depend on the type of syntactic constructs (e.g., a variable declaration) but not on other properties such as variable names. The same benefits are inherited for models as well (here renaming refers to the modification of identifier names such as the name of a state in a statechart).

4.3.1. Sorting of Model Objects

As explained in section 4.1, statechart files can be written in many different ways without changing the behavior. In a naive implementation of a token-based system, an attacker can employ the *move* attack to obfuscate cases of plagiarism. The following section describes sorting of model objects as a way to defend against such attacks.

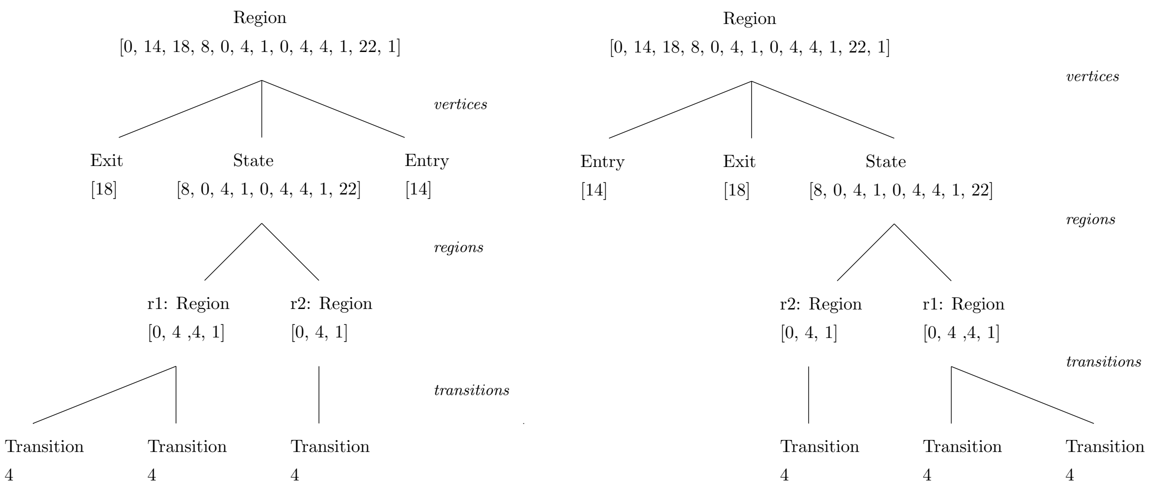
Note that the sorting procedure does not need to be performed as a separate step before the token extraction but can instead take place at the same time, requiring only a single recursive iteration.

Two possible sorting algorithms are discussed: the first is able to achieve resistance against any number of *move* attacks. The second algorithm is simpler yet performs better for other kinds of attacks.

4.3.2. Recursive Sorting Algorithm

This section describes a novel sorting algorithm that, when performed on input statecharts before the token extraction step, avoids susceptibility against the reordering of model elements on the same level of the containment tree in the input file. In other words, it avoids a targeted modification of the token sequence using the *move* attack.

This procedure sorts all model objects within each reference list of a model in a deterministic manner, considering their reference lists and other attributes. Sorting is achieved by comparing model objects using a token sequence obtained by iterating over each object and its child objects. This particular token sequence is solely used during the sorting step and does not contribute to the final token sequence for the statechart. Model objects are then sorted lexicographically by mapping each token type to an integer. Those with a



(a) Model object before recursive sorting (b) Model object after recursive sorting

Figure 4.1.: Recursive sorting of a *Region* model object

higher depth in the containment tree are sorted first as sorting occurs recursively. At some point in the iteration, a visited model object no longer contains child elements, which breaks the recursion. As a result, two statecharts containing identical model objects in equal quantities at each level of the containment tree are normalized into the same statechart. This, in turn, produces equivalent token sequences as tokens are extracted. The algorithm is explained in more detail in the following paragraphs.

Sorting a list of model objects requires a priority to be assigned to each of the possible permutations. Each permutation can be uniquely identified by the list of tokens extracted from the model object. We say that token *t1* is smaller than *t2* if the ordinal of its type is smaller than that of the type of token *t2*. The ordinal of a token type is its fixed position within the enumeration of all token types.

Consider the lists of token types [*A*, *B*], [*B*] and [*B*, *A*], in this order. Assuming $A < B$, these lists are sorted lexicographically since the first element of the first list is smaller than that of both other lists and the second list contains fewer elements than the third while their first elements are equal to each other.

In the base case, i.e. when comparing non-nested model objects, the order only depends on the tokens extracted for that model object since such model objects do not contain any child elements. For example, when sorting vertices within a state object, vertices of type *Entry* are placed before *Exit* vertices. This is because the extracted tokens of type *ENTRY* are smaller than *EXIT* tokens due to the definition of the token type enum. In the recursive case, all of the reference lists are sorted in a fixed order before extracting tokens. Following this procedure ensures that the extracted tokens do not depend on the order of the references, nor on the order of model objects within each reference list.

Figure 4.1 depicts an example for a region model object assumed to be part of a statechart. The region comprises *Entry*, *Exit* and *State* vertices. Within the state, two regions *r1* and *r2* are contained in this order, with the first region containing two transitions and the second one containing a single transition. The ordinal *4* of the associated *TRANSITION*

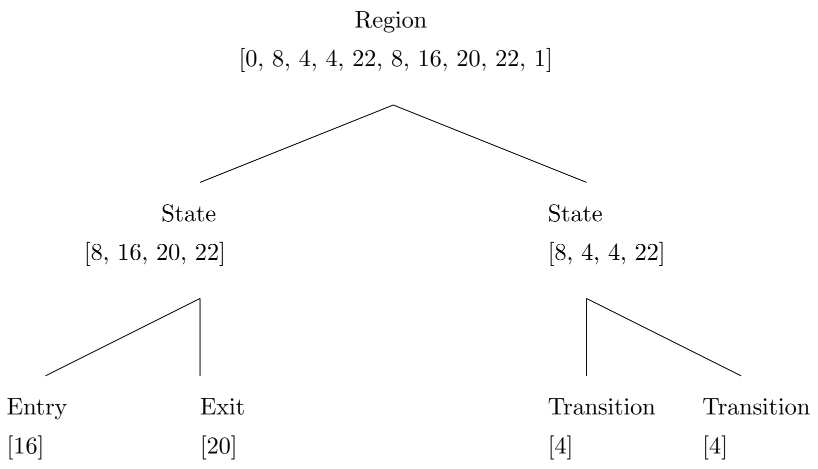


Figure 4.2.: A region model object with two states

token type is displayed beneath the transition model objects located at the bottom of the diagram. Below the transition model objects at the bottom of the diagram the ordinal *4* of the associated *TRANSITION* token type is shown. Underneath each sorted model object one can see the resulting token sequence that is obtained by visiting the model object itself followed by the sorted reference lists. For example, for region *r2* the extracted tokens are *REGION*, *TRANSITION* and *END_REGION* identified by their ordinals *0*, *4* and *1*. The ordinals for the *END_VERTEX* token is 22 which is why it is placed at the end of the token sequence for the state model object.

Figure 4.1a shows how the model object looks like after sorting all elements recursively by comparing the list of extracted token types. In particular, *r2* is placed before *r1* in the *regions* reference list because its token type sequence is smaller than that of *r1*. Starting from the leaves, on each level of the tree the smallest model object is placed on the left while the largest is placed on the right. One can obtain the final token sequence by appending the token lists from bottom to top, left to right of the diagram; end tokens separate nested model objects.

4.3.3. Simple Sorting Algorithm

While the proposed recursive sorting procedure achieves resistance against a specific kind of attack, sorting the model objects in this way has drawbacks. The fundamental problem is that when applying this procedure, deleting or inserting model objects now has a non-local effect on the token sequence.

This is demonstrated in Figure 4.2, which depicts one region with two states. After sorting, the resulting token sequence for this region is *[0,8,4,4,22,8,16,20,22,1]* (since the token sequence for the state on the right is smaller than that for the left one). Inserting a single new transition into the second state results in a token sequence of *[8,4,4,4,22]* for the second state. This causes the token sequences of both states to swap their relative order since the second state is now placed after the first when sorting them lexicographically. As a result, the token sequence for the whole region is *[0,8,16,20,22,8,4,4,4,22]* which is significantly different from the original token sequence.

A similar behavior can be observed when a model object is deleted. Deleting the *Exit* element from the first state changes the token sequence from *[0,8,4,4,22,8,16,20,22,1]* to *[0,8,16,22,8,4,4,22,1]*.

The algorithm described in the last section sorts model objects based on the complete token sequence that results from recursively iterating over all child elements. In contrast, the proposed simple algorithm only inspects the token type extracted for the root element. This way, the token order will not change when a child element is modified making it more resistant to deletion or insertion attacks.

5. Token-Based Plagiarism Detection for Statecharts

There are a number of steps required to implement a token-based system for detecting plagiarized statecharts. In our work, we choose to extend the open-source Java-based framework JPlag, described in detail in subsection 2.2.1. Since statecharts can be stored in a number of different formats, an implementation needs to first read in or parse the statecharts, ideally by using existing frameworks (section 5.2). Section 5.3 details how individual elements in the statechart are turned into a list of tokens. Two different strategies are proposed to demonstrate the effect of extracting tokens with varying degrees granularity.

A sorting step is discussed that ensures immunity against a certain type of attack. Next, a way to visualize matching sections of statecharts in JPlag is described. Finally, limitations of applying a token-based plagiarism detection system to statecharts are discussed.

5.1. Architecture

Both language modules are structured similarly. JPlag's *Language* interface defines common methods that must be implemented for all language modules. These methods include the *suffixes*, *viewFileSuffix*, *minimumTokenMatch* and *parse* methods.

The *suffixes* method returns the possible file extensions of the statechart files that can be loaded with the language module. The extensions are { ".scxml" } for the SCXML module and { ".sct", ".ysc" } for the Create module. The extensions for view files used for visualization are chosen as *.scxmlview* and *.createview*, respectively. The default minimum token match is set to 10 in both cases based on the evaluation results (see subsection 6.3.1 for a discussion on the choice of this parameter).

The result of the *parse* method is a list of tokens that JPlag then uses as an input into its greedy string tiling algorithm (see subsection 2.2.1 for details).

5.2. Parsing

A parser class is proposed that is responsible for extracting relevant properties from a statechart into a predefined class structure. This structure makes the statechart easier to traverse than the raw input document. Further, it delegates to an instance of the selected token strategy to turn this intermediate representation into a list of tokens.

5.2.1. Parsing SCXML files

In the case of the SCXML language module, the parser maps raw XML elements and their attributes to Java objects. For example, a transition like *<transition event="user.press_button" target="Blinking">* (see Figure 2.2) is mapped to an instance of a *Transition* class that has two attributes *event* and *target*.

Parsing takes place in two steps:

1. Parsing of the XML file: statecharts are parsed using a custom SCXML parser built on top of the XML parsing capabilities provided by *Javax* [1]. We use the Javax DOM (Document Object Model) parser instead of a SAX (Simple API for XML) parser that is also part of Javax. The DOM parser is better suited for our use case as it allows random access to any element of the XML document and has a very simple API.

2. Construction of the statechart: Having obtained the document root node, all nodes are visited recursively in two passes: In the first pass, only *<initial>* elements are visited and the target attributes of their outgoing transitions are collected. An *<initial>* element indicates the default state of a compound state to enter [31] (W3C SCXML Specification, section 3.6). Initial elements can be defined prior to the target state they are referencing which explains the necessity of this first pass. This referenced target state is now considered an *initial* state. Omitting it would result in different token sequences depending on the relative order of an <initial> element and its target state within the document.

In the second pass, the whole document is visited again and a *Statechart* object is constructed based on the encountered XML elements and their attributes. Figure 5.2 shows the classes that are part of a statechart. The classes are named after the XML elements in the SCXML specification while the attributes are parsed from the XML attributes.

Some classes additionally contain methods. These include *isTimed* and *isGuarded* for transitions, *isInitial* and *isRegion* for states. A transition is guarded if its *condition* attribute is not null. A state is a region if it has at least one substate.

Since timed transitions are not part of the SCXML specification, more effort has to be made to identify them. To determine whether a transition is timed, the *Entry*, *Transition* and *Exit* as well as their child elements *Send* (for *Entry* and *Transition*) and *Cancel* (for *Exit*) are examined. If all three elements contain matching IDs, the transition is marked as timed. An example of a timed transition is shown in Figure 5.1.

The SCXML parser performs some basic assertions as the statechart elements are visited. For example, each state element in the XML file must contain an *id* attribute which is necessary to complete the first pass. If an assertion fails, a *ParsingException* is thrown and the parsing is aborted.

5.2.2. Parsing Create files

Create uses .ysc files to store statecharts. This file is continuously modified as the user updates the diagram using the graphical editor. It contains all kinds of information about a statechart including model elements such as states or transitions, but also their appearance

[1] `https://docs.oracle.com/javase/8/docs/api/index.html?javax/xml/parsers/package-summary.html`

```
<onentry>
   <send event="Start_t_1_timeEvent_0" delay="1s"/>
</onentry>
<onexit>
   <cancel sendid="Start_t_1_timeEvent_0" />
</onexit>
<transition event="Start_t_1_timeEvent_0" target="Start">
</transition>
```

Figure 5.1.: How a timed transition can be modelled in SCXML

and position that is part of the visual representation of the state graph. Since these visual attributes are not relevant for understanding the semantics of a model, it is discarded while the statechart is loaded.

The Create language module utilizes EMF's resource loading capabilities to load the .ysc files. First, the supported file extensions are registered. Then, the statechart object is obtained from the loaded resource.

5.3. Token Extraction

The next step is to perform the actual token extraction. For each language module, two separate token extraction strategies are proposed, simple and handcrafted. The simple strategy focuses on the general structure of statecharts while the handcrafted strategy is more specialized. Tokens are extracted based on the model elements of statecharts.

To extract tokens, the statechart is traversed starting from the root element, followed by the respective child elements in a fixed order. Apart from the token type, each token contains information about the file and the *StatechartElement* / *EObject* associated with it. As it is appended to the list of extracted tokens, each token is enhanced with additional information required for visualization such as line and column numbers of the view file (see section 5.4).

For both strategies, we follow these general rules:

- R1: extract a distinct token for each class within the containment tree
- R2: for a nested model object, extract a token for the root object followed by tokens for each model object in its references lists
- R3: after extracting tokens for a nested model object, append a special end token

These three rules shall ensure that the structure of the input statechart is at least partially captured while mapping the model objects to a token stream.

The second and third rule make it possible to reconstruct the containment tree from the token stream where each model object is terminated by an end token. Some model objects of a certain type can contain references to other model objects of the same type. Appending an end token ensures that the tokens for each model object are delimited from

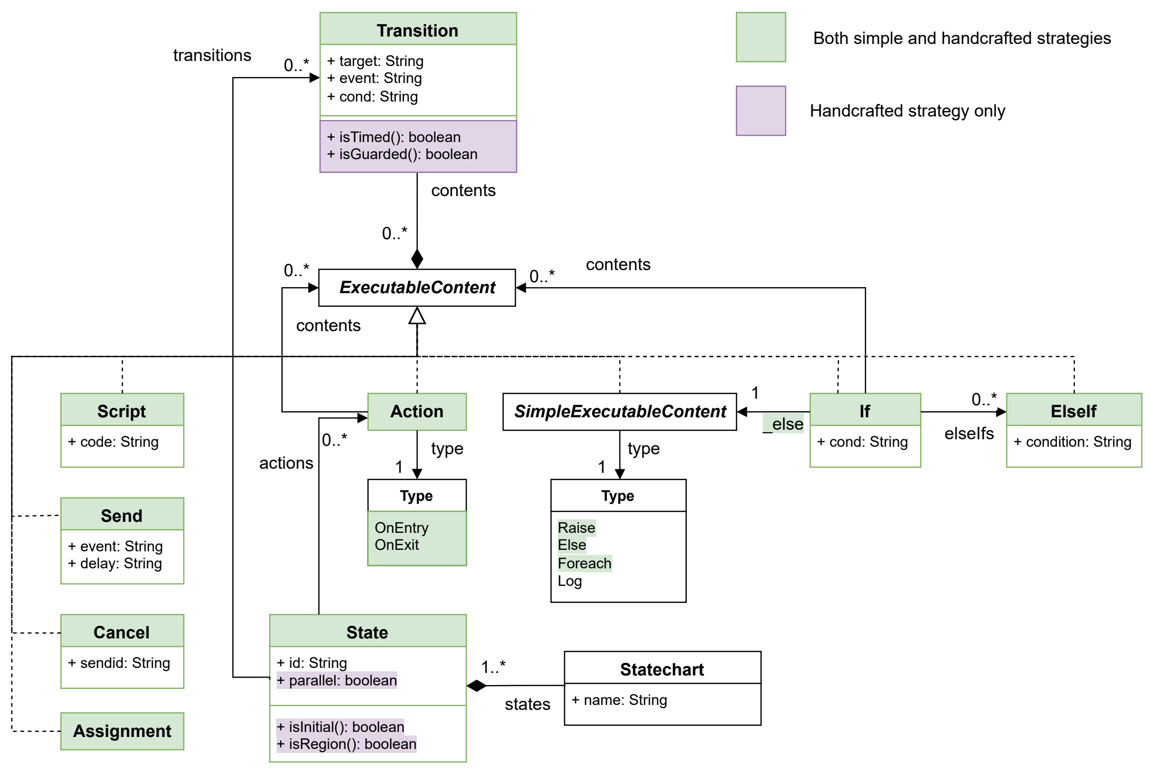


Figure 5.2.: The classes in the *model* package of the SCXML language module, showing which information is extracted as tokens for the two strategies

each other. In our case, omitting the end token would introduce ambiguities (meaning that it is possible to generate the same token stream from distinct model objects).

For example, an instance of the *Region* class contains instances of the *State* class within the *vertices* reference list. At the same time, a *State* instance can contain other regions. Without the end token, two adjacent *Region* tokens can either be the result of a state containing a region or the result of multiple regions within the same reference list.

5.3.1. Simple Token Extraction

The task of the *TokenGenerator* class is to visit all model elements recursively while extracting tokens. Figure 5.2 and Figure 5.3 shows models of statecharts for both language modules. Simultaneously, these models are at the same time class diagrams constructed by the parser which are subsequently used during token extraction. The simple token extraction strategy extracts a distinct token for each class in the models, which are highlighted in green. The following subsection describes specifics of the SCXML language module.

SCXML language module

All classes that represent parts of a statechart inherit from a common interface *StatechartElement* (Figure 5.2). The reason why tokens are also extracted for the *SimpleExe-*

cutableContent.Type enum is that *SimpleExecutableContents* can be seen as just another type of *ExecutableContent*, therefore using separate classes to represent them is not necessary.

If a class has references to other objects (e.g. a *Statechart* contains multiple *States*), an end token is extracted after extracting tokens for the child model objects.

Within one reference list, the child model objects are first sorted in a deterministic manner (subsection 4.3.1 which discusses sorting). Some model objects contain multiple references, e.g. *If* contains both *ExecutableContents* and *ElseIfs*. In that case, the tokens for the reference lists are extracted in a fixed order. For the *If* model object, first the *ExecutableContents* are visited, then the *ElseIfs*.

In section 4.2, an obfuscation attack is described whereby an attacker can change the number of *<onentry>* or *<onexit>* attributes without affecting the behavior of the statechart. The parser accounts for this by only extracting a single *ENTRY* or *EXIT* token even if there are several *actions* of a given type. All *executableContents* within all *actions* are collected into a single list and tokens are extracted in the order of appearance in the source file. The executable contents may not be reordered as that would change the behavior.

5.3.2. Handcrafted Token Extraction

The previously described simple strategy only relies on the class of the model object to extract tokens. However, there is other semantic information about a model object such as specific attributes. To address this, the following *handcrafted* token extraction strategy is proposed. It uses a larger token set than the simple strategy. The hope of introducing this extended strategy is that the false positive rate is lowered by introducing more detail into the token stream.

The additional tokens are extracted based on the values of Boolean attributes and values of enums. These types of attributes can be easily mapped to tokens. Other attributes such as names are not relevant for the purpose of detecting plagiarisms and thus there are no tokens extracted for them. Tokens that are extracted include all tokens of the simple strategy (shown in green) plus additional tokens (shown in purple).

Compared to the simple strategy, there are now more fine-grained tokens for *States* and *Transitions* as well as *Choice* and *Entry* model elements. For transitions, the extracted tokens for timed and guarded transitions are now different. For states, different tokens depending on the attributes of the states such as whether they are parallel states. A special characteristic of the Create language module is that tokens for *Choice* elements are extracted based on their kind (dynamic or static).

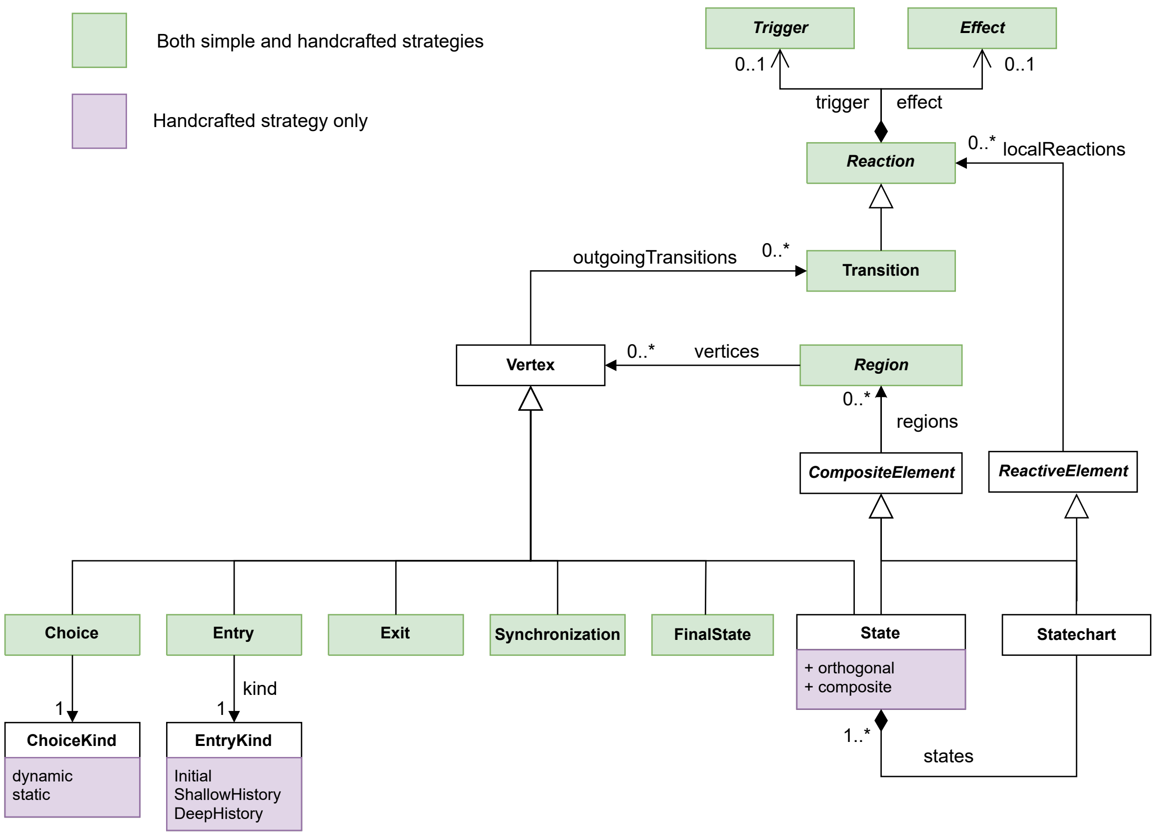


Figure 5.3.: An extract of the metamodel used for the Create language module, showing which information is extracted as tokens for the two strategies

5.3.3. Implementation

```java
public class SimpleScxmlTokenGenerator extends AbstractScxmlVisitor {

   protected ScxmlParserAdapter adapter;
   protected Sorter sorter;

   public AbstractStatechartVisitor(ScxmlParserAdapter adapter) {
      this.adapter = adapter;
      this.sorter = new SimpleSorter(this);
   }

   public List<Integer> peekTokens(StatechartElement element) {
      ScxmlParserAdapter prevAdapter = this.adapter;
      PeekAdapter peekAdapter = new PeekAdapter();
      // Switch out the main adapter for the peek adapter
      // so that the main token stream is not affected.
      this.adapter = peekAdapter;
      visit(element);
      this.adapter = prevAdapter;
      return peekAdapter.getTokenTypes();
   }

   @Override
   public void visitStatechart(Statechart statechart) {
      for (State state : sorter.sort(statechart.states()) ){
         visitState(state);
      }
   }

   // ...
}
```

Listing 5.1: Extract of the *AbstractStatechartVisitor* class showcasing the use of the *Sorter* class as statechart elements are visited

Listing 5.1 shows a part of the *SimpleScxmlTokenGenerator* class that represents the simple token extraction strategy for the SCXML module. To alter the active sorting strategy, the sorter variable must be set to a different instance of *Sorter*.

The *Sorter* interface contains a single method that receives a list of *StatechartElements* and returns a sorted list. The strategy pattern is used to represent the sorting method. There are three implementations of *Sorter*: *NoOpSorter*, *SimpleSorter* and *RecursiveSorter*. While *NoOpSorter* returns the input list unaltered, *SimpleSorter* and *RecursiveSorter* apply the corresponding algorithm as described in subsection 4.3.1. The *sort* method is invoked from within *visit* methods for each nested model object, such as the individual states of a *Statechart*, as demonstrated in the example.

The *sort* method relies on *peekTokens* to obtain a list of token type ordinals associated with the current model object. The token ordinals are then compared using the *com-*

```
main region: Region {
 Entry {
  Transition (-> Start)
 }
 Start: State {
  Transition (-> Blinking)
 }
 Blinking: Orthogonal & composite state {
  blinking1: Region {
   Shallow history {
    Transition (-> Light)
   }
   Dark: State {
    Transition (-> Start)
    Transition (-> Light)
   }
   Light: State {
    Transition (-> Dark)
   }
  }
  blinking2: Region {
   Dark: State {
    Transition (-> Start)
    Transition (-> Light)
   }
  }
  Transition (-> Start)
 }
}
```

(a) Visualization for the Create language module

```
main_region: Region {
 Start: State {
  Transition (-> Blinking) {
  }
 }
 Blinking: Region {
  OnEntry {
   Assignment
  }
  Transition (-> Start) {
  }
  Dark: State {
   Guarded transition (-> Start) {
   }
   Timed transition (-> Light) {
   }
  }
  Light: State {
   OnEntry {
    Assignment
   }
   Timed transition (-> Dark) {
   }
  }
 }
}
```

(b) Visualization for the SCXML language module

Figure 5.4.: Visualization of the example statechart

pareTokenTypeLists comparison function of *PeekAdapter* that lexicographically compares two token ordinals. Inside *peekTokens*, the instance of the *ScxmlParserAdapter* class is temporarily replaced with an instance of *PeekAdapter*. Then the *visit* method for the current *StatechartElement* can be called without affecting the original token stream. Inside the *PeekAdapter* class, the ordinals of the appended tokens are collected into a list which is then returned by *peekTokens*.

5.4. Visualization

View files are used by the online JPlag report viewer available at `https://jplag.github.io/JPlag/`. It enables the comparison of matching sections within a submission pair.

A hierarchical text-based representation is proposed. The format is inspired by the textual format Emfatic that is employed to represent models [14]. Each extracted token is mapped to a line in the view file. A line contains the description of the corresponding token type and optionally the name of the associated *EObject* / *Statechart* if applicable. For transitions, the name of the target state is denoted in brackets to provide more context to the viewer. To denote a nested model object, an opening curly brace *{* is appended to emphasize the nested structure. Correspondingly, a closing curly brace denotes an end token. Model objects that are children of the same nested model object are vertically aligned, shown in the same column.

Figure 5.4 shows an example view file for the Create language module on the left and for the SCXML module on the right. The view files are created from the example statechart shown in Figure 2.2 in chapter 2. This example highlights the different focus of the two language modules.

6. Evaluation

6.1. QGM Plan

G1 Analyze the ability of the language modules to detect plagiarized statecharts

Q1.1 How does the minimum token match parameter influence the average similarity values?

Q1.2 How does the choice of the token extraction strategy influence the average similarity values?

Q1.3 How does sorting of model elements using different algorithms influence the average similarity values for different kinds of obfuscation attacks?

Metric Compute the average similarities of original and plagiarized submission pairs. Compare the distance between both similarity distributions.

6.2. Methodology

To evaluate the performance of the plagiarism detection system and fulfill the goals outlined in the GQM plan, a dataset of statechart files is required. In our case, we were generously provided with two sets of statechart assignments created by students at the University of Antwerp. They were made in the context of the Master's course *Modelling of Software-Intensive Systems* in the years 2020 and 2021, respectively.

For the 2020 assignment, students are asked to create a dashboard that simulates the behavior of employees working in a factory [30]. The task for the 2021 course is to model the behavior of a personal rapid transport trolley that drives on a circular track [32]. For the 2020 dataset, we note that the average model sizes are smaller than for the 2021 dataset with a median number of model objects of 177 (min: 91, max: 248, avg: 177.38) vs. 151 (min: 133, max: 177, avg: 154.47). Table 6.1 shows the percentage of each of the six types of model objects that occur on average in each dataset.

The assignments were created using Create and are available as .ysc files. The dataset is unlabeled in the sense that it is not clear how many cases of plagiarism are present or to what degree any pair of statecharts is plagiarized.

This dataset is used to evaluate the system by applying a set of common obfuscation attacks on all input statecharts. Before evaluating the SCXML language module, all .ysc files are first converted to the expected input format (.scxml) by using the conversion feature of Create.

To perform the evaluation, we develop a program that can apply obfuscation attacks to a given input statechart file. Each statechart is loaded using EMF, yielding a *Statechart* object.

Model object	2020 (%)	2021 (%)
Transition	51.35	45.00
Exit	0.22	0.47
State	27.94	27.64
Entry	9.48	11.81
Choice	1.41	3.50
Region	9.61	11.39
FinalState	-	0.19

Table 6.1.: Percentage of model elements per dataset

Then, a selected obfuscation attack is applied to the model and the modified statechart is exported.

As mentioned, Create performs several checks to validate a statechart. These checks are also performed before a statechart is exported. If any of them fails, the generation is aborted. This issue is critical since the same obfuscated statecharts need to be used in order to evaluate the SCXML language module.

To address this issue, we propose a modified set of attacks that generate only valid statecharts. These statecharts can be exported without generating any errors in Create. In the following description of the obfuscation attacks, some relevant invariants are explained that need to hold for a statechart to be valid.

The *swap* attack is not implemented as two *move* operations have the same effect as one *swap* operation. Note that for each attack listed, the term *random reference list* indicates that a reference list is selected from a random model object anywhere in the containment tree.

After running JPlag on pairs of original input files and their plagiarized counterparts, as well as on unrelated pairs of original statecharts, the similarity distributions are compared.

The first kind of submission pairs should yield much higher similarity values compared to pairs that are unrelated assuming the dataset contains no plagiarized submissions.

Element Deletion (*delete_n*):

In this attack, *n* random model objects are removed from non-empty reference lists. The attack is confined to only delete model objects of type *Transition* or *ReactionProperty*, both of which are leaves (i.e., model objects that either do not contain any reference lists or all of their reference lists are empty). The restriction of only deleting leaves is put in place to avoid deleting large parts of the model with a single deletion operation.

If a transition is selected, it is first ensured that the transition can be safely deleted from the model. A state's incoming transition is only deleted if that state has other incoming transitions originating from a different state. Otherwise, an unreachable state would be produced. Moreover, a singular transition from an *Entry* model object is not deleted since each *Entry* must contain at least one transition.

Other leaf model objects are *Entry* and *Exit*. For *Entry* model objects, each *Region* model object must include an *Entry* element (the variable for the error message Create uses is called *TRANSITION_UNBOUND_DEFAULT_ENTRY_POINT*), so it cannot be simply deleted. Similarly, it must be ensured that for every state, each region within it contains a named *Exit* model object (*TRANSITION_NOT_EXISTING_NAMED_EXIT_POINT*). Deleting any of them would thus involve non-trivial checks to exclude the generation of an invalid statechart. This fact leads to the decision not to implement them for the evaluation of this work.

Element Insertion (*insert_n*):

This attack involves inserting *n* model objects to one or multiple references lists of the model. We restrict our attack to only insert model objects where the underlying EObject is either of type *State* or *ReactionProperty*.

If a *State* EObject is selected, a new *State* EObject is first created using the *SGraphFactory* instance (which is generated by EMF) and assigned a random name. The created state must be connected to the existing model to be reachable, so a transition that starts at a random existing vertex and ends at the newly created state is created and inserted into the model instance. The source vertex must neither be of type *Entry* nor of type *Exit*. An additional condition for this transition is that its source and target vertices are not orthogonal to each other. Therefore, a check that verifies this condition is added and the source vertex is reselected if the check fails. Once the source and target vertices are selected, a guard condition is added to the transition by setting the specification attribute to be equal to "true". This achieves that Create no longer shows an error when trying to convert the obfuscated statechart. Omitting this guard condition would produce a transition that cannot be executed. If instead a *ReactionProperty* EObject is selected, a new *ReactionProperty* EObject is created using the SGraphFactory instance, assigned a random name and added to the containing transition. Finally, the created model object is inserted into a compatible reference list at a random index.

Element Moving (*move_n*):

This attack involves moving model objects within a reference list. A random model object is selected from a random reference list and moved to a different random position. It is possible that consecutive executions cancel each other out.

Element Renaming (*rename_n*):

The *rename_n* attack chooses *n* random model elements where the corresponding EObjects are of type *NamedElement*, such as *State* or *Region*. Then, the name attribute of each model element is set to a random new name using the *setName* method. The new name is randomly generated from a combination of ten letters and digits. Only EObjects with a non-null and non-empty name are considered, since that might break the semantics (e.g. transitions within an *Entry* model object must always be null).

Following this approach does not always work: for instance, it breaks code generation for a total of six statechart files in the 2020 dataset since the attack does not rename model objects consistently. More specifically, some *specification* attributes of some EObjects of type *SpecificationElement* reference other *NamedElements* by their name which would have to be updated to result in valid statecharts.

In the experiments in the next section that use the *rename* attack, these six statecharts are removed from the input files. This reduces the number of input statecharts for the 2020 dataset from 21 to 15.

6.3. Results and Discussion

This section presents the results of a set of experiments with each having different goals: The initial experiment delves into the effects of the minimum token match parameter t and the general distribution of the dataset ($d = 2020$ or $d = 2021$) for both our custom modules and the existing module for instances of EMF models. In the second experiment we discuss the impact the token extraction strategy (s is either *simple* or *handcrafted*) has on the distribution of the dataset and the modules' ability in plagiarism detection. Lastly, the impact of model object sorting on obfuscation attack resistance is examined in the third experiment.

6.3.1. Impact of Minimum Token Match

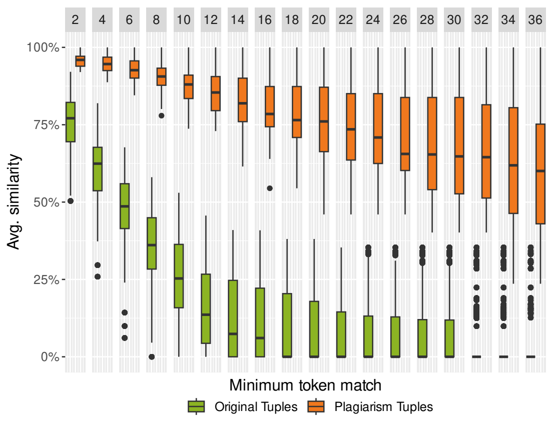
In this first experiment, we evaluate the impact that the minimum token match has on the similarity distribution for each language module. After varying the token extraction strategy (simple vs. handcrafted, section 5.3) for both the SCXML and Create language modules, we discuss first differences between the resulting similarity distributions.

The importance of the minimum token match parameter is emphasized in the foundations section (subsection 2.2.2). As explained, this parameter has a large effect on the number of false positives or false negatives that occur.

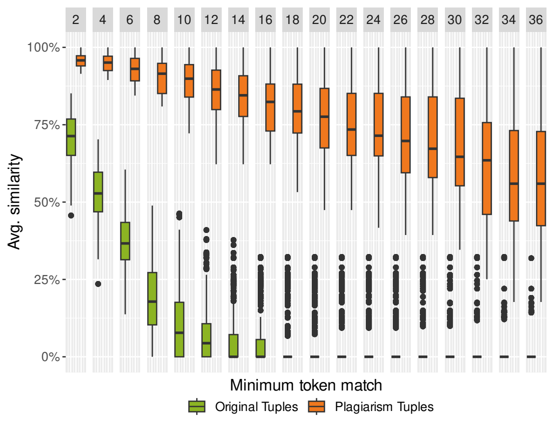
The properties of the parameter can also be observed in the case of our language modules. Figure 6.1 shows the distribution of average similarity values for the SCXML language module, varying the minimum token match. In the subdiagram on the left-hand side, the simple token extraction strategy is used, while the similarity distribution for the handcrafted one is displayed on the right.

In all figures shown for this experiment, plagiarism tuples only comprise statecharts obfuscated with the *insert10* and *delete5* attacks. To improve readability, plagiarisms created using the remaining attacks *move100* and *rename100* have been omitted from the diagram. This is because plagiarisms created with those attacks are 100% similar to the original statecharts and would thus result in a maximum value of 100% for each boxplot (see subsection 6.3.2).

First, we look at the results for the SCXML language module (Figure 6.1). As expected, the average similarity for both the original tuples and the plagiarism tuples steadily decreases as the minimum token match is increased. For a value of around 10 for the minimum token match parameter, the median average similarity is sufficiently low (25.31% for the simple strategy vs. 11.22% for the handcrafted one with $t = 10$) to expect few false positives. Further, the range of the similarity distribution for the plagiarism tuples is also not too high to risk an overlap with the distribution of the original tuples. Overlapping



(a) Simple strategy



(b) Handcrafted strategy

Figure 6.1.: Minimum token match vs. average similarity for the SCXML language module (2020 dataset)

distributions should be avoided because a reviewer can no longer clearly distinguish all potential instances of plagiarism from assignments that are not plagiarized.

The Create language module shows significant overlap of the two similarity distributions. This suggests that it performs worse than the SCXML module for insertion and deletion attacks. The module for EMF model instances shows low median similarity values for the original tuples for a reasonable choice of t such as $t = 10$. However, as for the Create module, there is an overlap of the similarity distributions.

This also holds for the 2021 dataset, see subsection A.1.1 in the appendix. Thus, we propose to use a fixed value of $t = 10$ for later experiments.

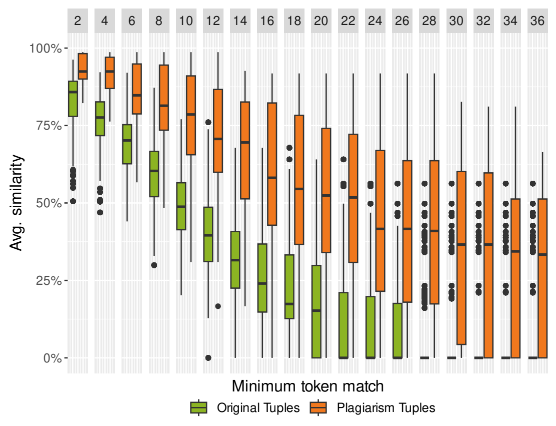
6.3.2. Impact of the Token Extraction Strategy

In this section, we look at the impact of the token extraction strategy. For this experiment, we run the SCXML and Create language modules on our original dataset and plot the average similarity distributions, varying only the active token extraction strategy. The EMF language module is also evaluated on the same dataset and shown in the diagram.

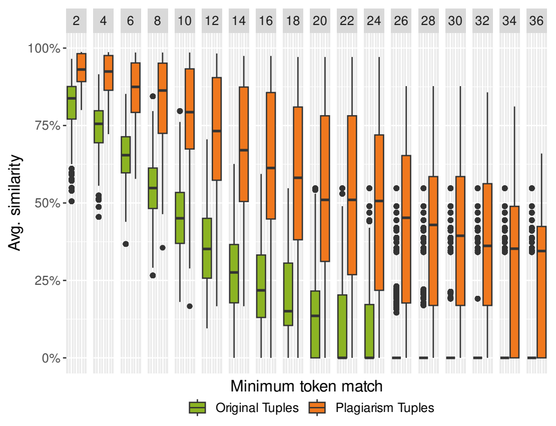
Figure 6.4 shows the similarity distribution for the 2020 dataset for each obfuscation attack. The minimum matching token length is fixed at t=*10* as before and the recursive sorting strategy is used.

First, it is noted that the median similarity values are consistently lower when using the simple token extraction strategy as compared to the handcrafted strategy when considering the original tuples. This could already be seen in the first experiment where it holds for other values of t as well.

One possible reason for this is that the handcrafted strategy is more fine-grained than the simple strategy, i.e. the token set is larger and some tokens are only extracted in the handcrafted token set. This lowers the median average similarity since in this case some sections in the token sequence that match when using the simple strategy do not match any longer.



(a) Simple strategy



(b) Handcrafted strategy

Figure 6.2.: Minimum token match vs. average similarity for the Create language module (2020 dataset)

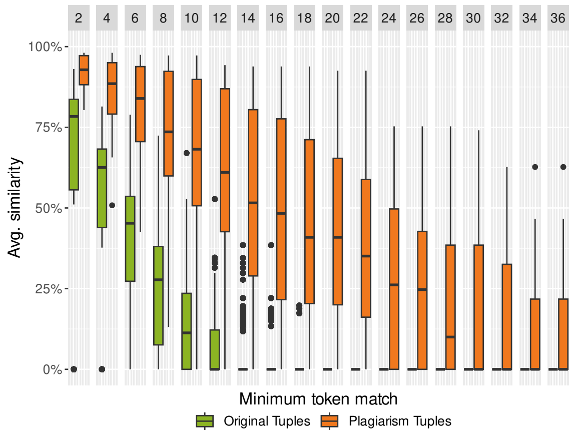
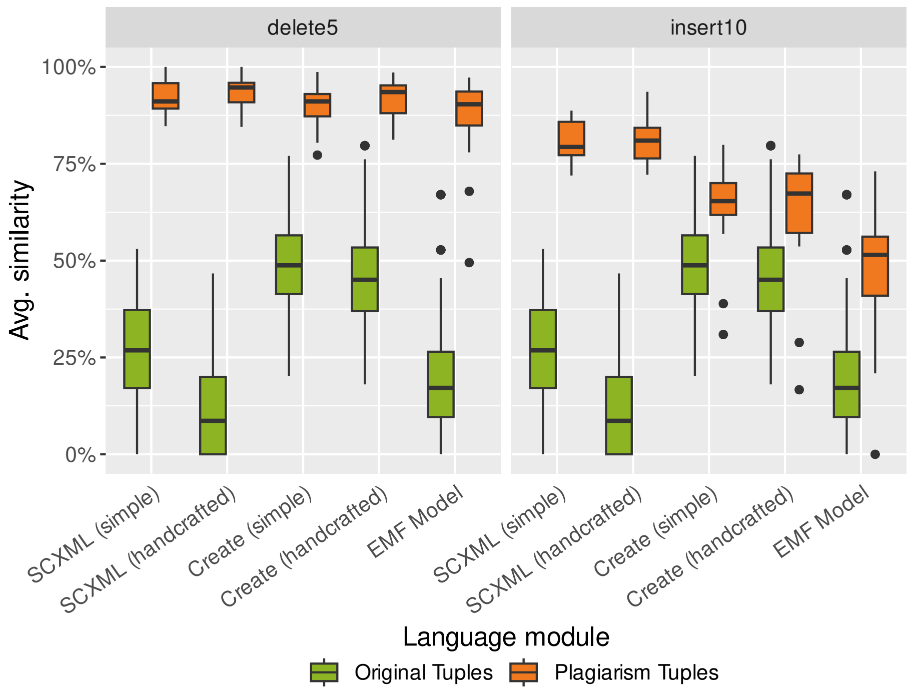
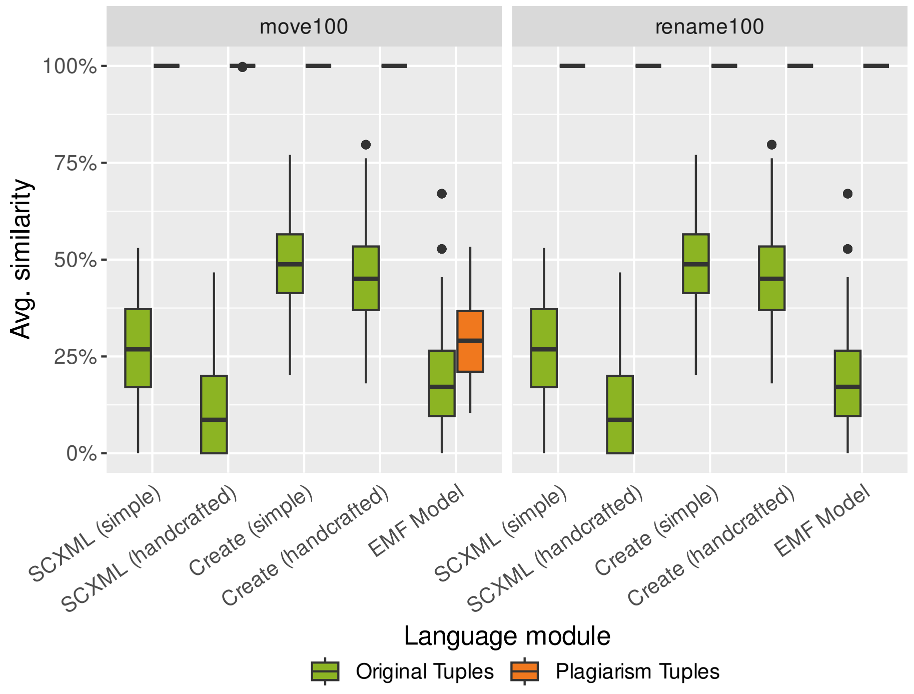


Figure 6.3.: Minimum token match vs. average similarity for the EMF Model language module (2020 dataset)



(a) Similarity distribution for the insert and delete attacks



(b) Similarity distribution for the move and rename attacks

Figure 6.4.: Impact of the token extraction strategy on average similarity for the 2020 dataset (t = 10, recursive sorting enabled)

For example, in the case of the SCXML language module, if the first assignment contained a transition that was not timed and the other assignment contained a transition that was timed, the extracted token would be of type *TRANSITION* both times. For the handcrafted strategy, for the second assignment a *TIMED_TRANSITION* is extracted instead and the two tokens are no longer the same which breaks the token sequence.

Further, in Figure 6.4b one can again observe that sorting the statecharts recursively establishes immunity against attacks of type *move*. Also, it is expected that all approaches are immune against renaming attacks since the name of a statechart element is not considered as the tokens are extracted.

6.3.3. Impact of Sorting

This experiment is designed to gain insight into the effects of the two sorting procedures described in detail in subsection 4.3.1.

As discussed in subsection 4.3.1, this extra step takes place during the token extraction. The goal of the recursive sorting algorithm is to ensure immunity against *move* attacks while the simple algorithm aims to perform better for deletion and insertion attacks.

Figure 6.5 depicts the average similarity distribution with different configurations for the sorting procedure using the 2020 dataset. Sorting is either turned off entirely or the simple or recursive algorithm is used. The EMF model language module does not perform any kind of sorting of elements within a reference list, which is why it is not displayed here. Similarly to previous experiments, the *rename100* attack is omitted since it has no effect on the extracted token sequence (as shall be demonstrated in the next subsection). Parameters t = *10* and s = *handcrafted* are selected as discussed.

Looking at the diagram, a couple of observations can be made: first, the median average similarity for the original tuples is lowest when sorting is disabled and highest when the recursive sorting algorithm is used. The effect is most noticeable for the Create language module. This can be explained by the fact that sorting increases the amount of possible matching sections. Without sorting, if there are parts of two statecharts that consist of the same types of model objects but in different order the token streams would not match completely. Sorting now allows these sections to match, leading to higher similarity values.

It can be seen that the choice of the sorting algorithm has a lower effect on the original tuples for the SCXML module compared to the Create module. This may be due to the fact that for the SCXML module, more token types are related to the data model (executable contents), while the Create module makes use of more structural token types. Executable contents cannot be reordered because that would change the behavior of the statechart so overall there seem to be fewer elements that can be reordered.

Issues with the recursive sorting algorithm are presumed decreased resistance against insertion and deletion attacks (subsection 4.3.2). While the effect appears not too strong in the diagrams, the average similarity for tuples created from the insertion or deletion attacks are actually lowest when using this algorithm.

As a further observation, it is noted that both systems manage to successfully achieve (near) immunity against *move* attacks when the recursive sorting algorithm is used. For both the Create and SCXML language modules, the similarity is at 100% for nearly all plagiarized tuples, which reinforces the earlier claims of immunity against this specific type

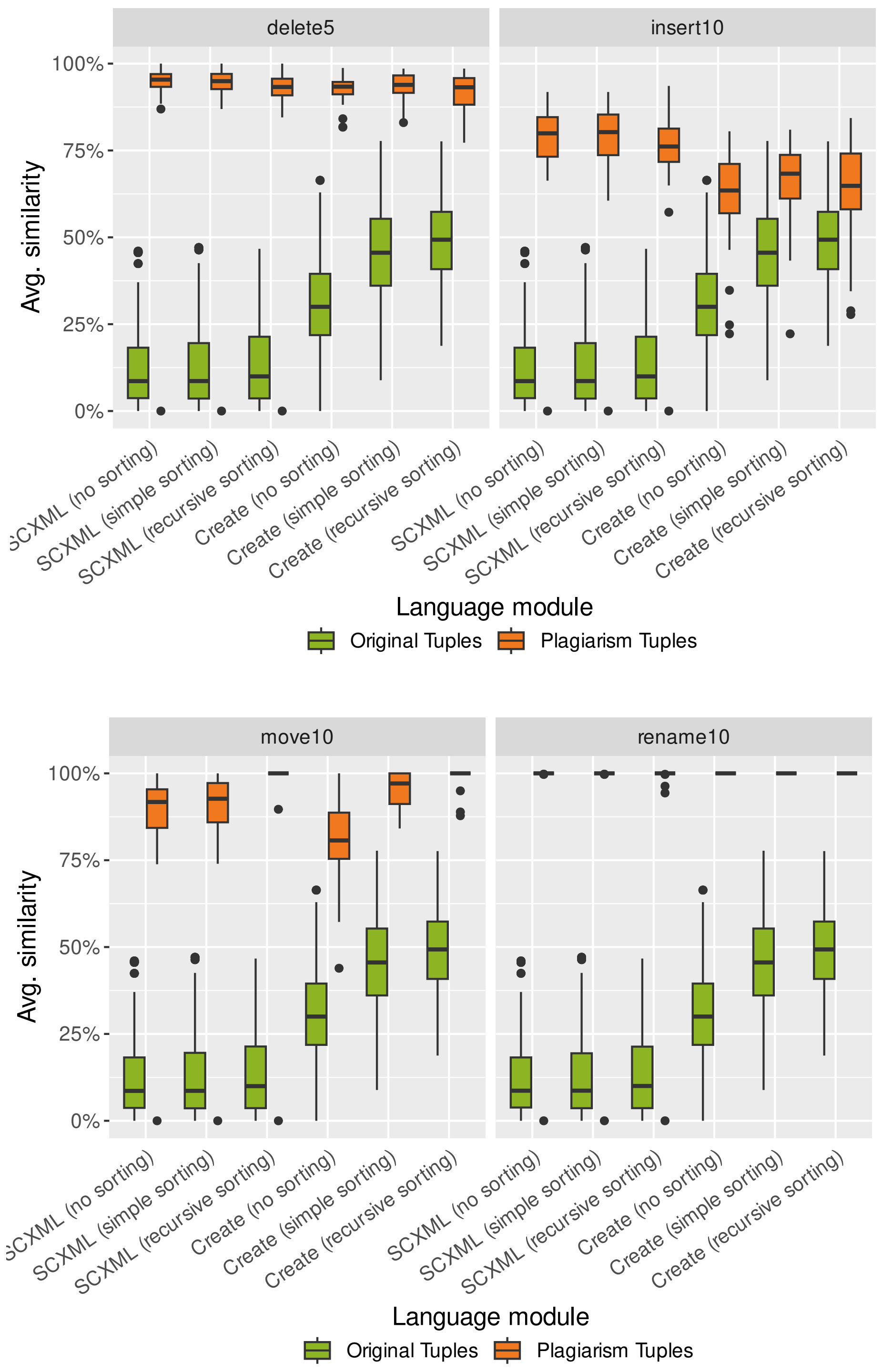


Figure 6.5.: Impact of sorting on average similarity to show the resistance against obfuscation attacks ($d = 2020$, $t = 10$, $s = handcrafted$)

of attack. It should be noted that there are several outliers in the diagrams. As of the writing of this thesis, it is not clear whether these are the result of a bug in the implementation of the parsers or the token extraction strategy, or even caused by JPlag's core code. Further possible reasons include bugs in the code used to generate the obfuscation attacks or in the code of Create that is used to generate the SCXML files. Due to organizational reasons, we choose not to investigate this further.

The above observations also hold for the 2021 dataset, see subsection A.1.1.

6.3.4. Comparison of the SCXML and Create language modules

We now look at aspects where the two modules differ and explore how these variations might account for the observed differences in performance.

1. **Granularity and size of the token sets:** Although the size of the token sets for both language modules is the same at 23 (for the handcrafted strategy), the token sets focus on different aspects. For example, the SCXML token set does not contain tokens that are related to the statechart language which is used by Create. In SCXML there are special XML elements for the data model such as the *<assign>* or *<script>* elements [31]. Special tokens for these are only extracted in the SCXML module, making it more fine-grained than the Create module in that regard.

2. **Number of tokens extracted per statechart:** The different granularities of the token sets can also be observed by looking at the number of tokens that are extracted per statechart. For the original statecharts in the 2020 dataset, the median number of tokens extracted per statechart is 144 for the Create module while it is at *272* for the SCXML module (an increase of nearly 90%). While not as significant, there is also an increase in the median number of number of tokens for the 2021 dataset with 129 and 185, respectively (a difference of around 43%).

   These values can explain why inserting or removing the same number of elements into a statecharts has a smaller effect on the Create language module since disrupting a shorter token sequence has a larger impact on the reduction of the similarity value.

3. **Execution time:**

   Execution time is also a factor that becomes more important as the number of submissions grows. While the execution time plays a subordinate role for a small number of input files, it becomes significant when considering the practical use of the language module in real-world applications, such as academia.

   Figure 6.6 shows the execution time depending on the number of input statecharts for both language modules. The experiment is run on a subset of the 2021 dataset. The diagram shows the average number of milliseconds a call to *JPlag.run* takes over the course of ten runs.

   It is immediately evident that the execution time for the Create language module is much larger compared to the SCXML module. For 21 input files, the execution times differ by around a factor of 16 (1767 ms vs. 109 ms). Although the curves for

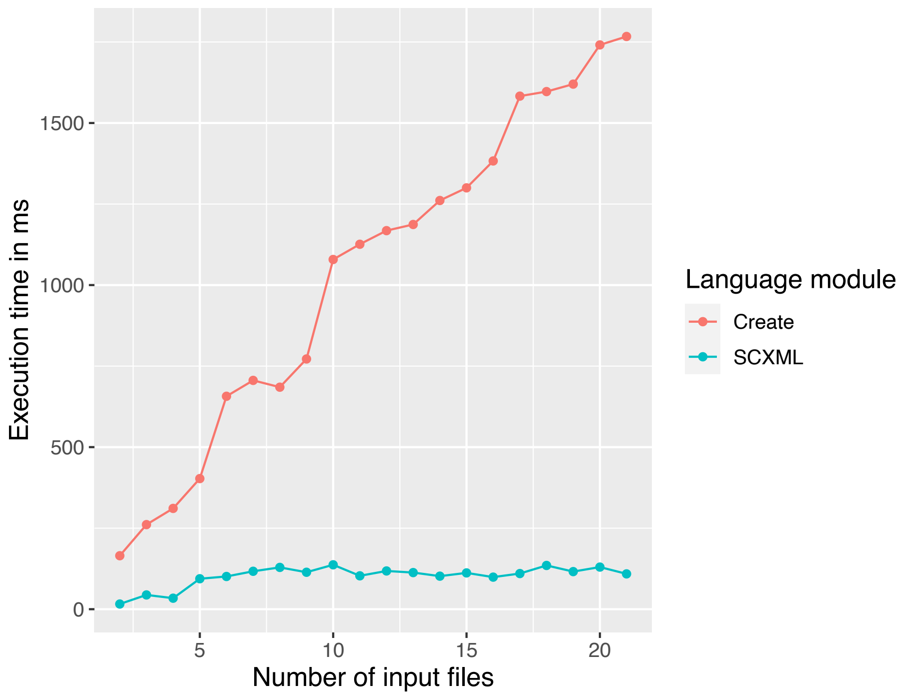


Figure 6.6.: Execution time per language module for the 2021 dataset

both language modules roughly follow a linear trend in this graph due to the low number of submissions in the datasets, it can be assumed that the execution time would approximate a quadratic curve if the number of input files were to increase further. This assumption is due to the fact that the number of comparisons JPlag performs is $\binom{n}{2} = \frac{n^2}{2} - \frac{n}{2}$ where n is the number of submissions.

The differences in execution time can be explained by the fact that the Create language module uses EMF to read in the statechart files which presumably is significantly slower than parsing the corresponding SCXML files with a custom parser.

6.4. Threats to Validity

In this section, various threats to the validity of our evaluation are discussed. These can be categorized into internal validity, external validity, construct validity, and threats to statistical validity.

Our evaluation faces two main threats to *internal validity*. First, the dataset used in our study is rather small, consisting of only 38 statecharts as part of two independent modeling assignments in an academic setting. The small number of statecharts may not be representative in other domains where statecharts are used such as in industrial applications [21]. Second, the exact number of plagiarisms in the dataset is not known, preventing us from showing the true-positive rate of our system. This limits our ability to accurately evaluate its performance. It is worth noting that obfuscation attacks are only applied to the statecharts in the .ysc format, rather than the SCXML files directly.

Consequently, the SCXML module was evaluated on the SCXML files generated from the obfuscated .ysc files. Since it is not clear how Create's SCXML code generator works, there is a possibility that some statecharts elements are rearranged during the conversion, which may influence the results shown for the *move* attack.

Another aspect is that for the second and third experiments, the number of attacks applied to a statechart varied within an experiment as well as across the experiments. For example, 100 move and 100 rename attacks were used for the former while it were ten for the latter. In the second experiment (subsection 6.3.2), the move and rename attacks were updated to apply the obfuscation attacks ten times each to improve the comparison within the experiment. Unfortunately, there was not enough time to rerun the evaluation for the third experiment (subsection 6.3.3) with an equal number of attacks per statechart. This reduces internal validity as it does not allow the comparison of results between experiments. However, the experiments could be rerun with matching number of attacks after generating the necessary obfuscated statecharts.

Regarding *external validity*, our approach focuses specifically on statechart models, which may limit the generalizability of our findings to other types of models or domains. However, we believe that some findings, such as the introduction of a sorting step, can be applied to other kinds of models with similar properties.

A threat to *construct validity* in our evaluation is the exclusive use of statecharts created with Create for testing the SCXML module. This approach does not consider the possibility that SCXML-based statecharts created using other (future) tools may utilize different features of the SCXML specification. Additionally, we might not have considered all features in the SCXML or Create specification.

Our evaluation encounters two threats to *statistical validity*. First, we use a program that randomly generates plagiarisms from the input statecharts to evaluate the effects of different obfuscation attacks on our language modules. This method may not accurately represent real-world obfuscation attacks, potentially limiting the relevance of the evaluation to real-world scenarios. Second, in a real-world setting, it is possible that an attacker develops more complex attacks that do not change the behavior of the statechart. One example of such an attack is described in section 4.2 whereby an attacker changes the number of *<onentry>* or *<onexit>* elements. While this specific attack is accounted for in the implementation, there may be other such as semantics-preserving attacks that an attacker can employ to change the token sequence. This limitation may lead to an overestimation of the system's performance in detecting real-world plagiarism.

Further, the described attacks are performed in such a way as to only generate valid statecharts. This is done to increase the realism of the attacks under the assumption that an attacker would only create valid statecharts that can actually be executed / simulated by the modeling software. Refer to the next section on limitations (section 6.5) for further details on this aspect.

6.5. Limitations

Statecharts have several attributes that make it harder to apply token-based plagiarism detection systems to them compared to code:

1. **Dependence on the representation format:** There are many different ways to store statecharts. Statecharts can be represented using various different notations including SCXML or the format used by Create depending on the application. This makes it harder to compare statecharts across multiple formats. Of course, one format can be converted into another. However, this is not always possible since some concepts are unique to a specific application such as Create.

2. **Difficulty to capture behavior:** Two statecharts may behave similarly even if they are structured differently. For example, states can be executed one after each other or in parallel (by using orthogonal states). A token-based approach extracts completely different tokens in each of these cases because it exclusively examines the constructs used in the definition of the statecharts while disregarding any similarities in behavior.

3. **Low resistance against specific kinds of attacks:** In our implementation, only rudimentary checks are performed during the parsing step (see subsection 5.2.1). However, when allowing invalid statecharts, it becomes very easy for an attacker to fool the plagiarism detection system. For example, an attacker can simply copy a random state and insert it into every region element of the statechart. If this state is unreachable (i.e. there exists no transition which contains this state as its target), the attack does not change the behavior of statechart. Nevertheless, it significantly changes the token sequence as a result of the inserted state elements.

7. Future Work

Our implementation shall be added to the JPlag repository as a new language module supporting plagiarism detection for statecharts. Based on the findings in the evaluation, we will include only the SCXML module with the handcrafted token extraction strategy, as this combination was found to perform the best. It remains to be seen how our system performs when using it with more and larger datasets. Potentially, the implementation for the token extraction has to be improved or extended to account for other SCXML elements or concepts defined in the specification.

This thesis introduced sorting algorithms that can be used to defend against certain obfuscation attacks. Each of them performs better for certain cases of obfuscation attacks than others. Ideally, it should be possible for the end user to toggle the sorting strategy to be used from the command line to control this behavior. Related to this is that the SCXML module currently uses Create-specific concepts such as timed transitions which do not have a direct equivalent in the SCXML specification. These should be made configurable by the user as well.

To evaluate the resilience against a wider range of obfuscation attacks, our system for automatically creating plagiarized statecharts should be expanded to support a wider range of attacks.

In the future, our token-based approach should be compared with other techniques including ones that are graph-based such as the work of Fauzan et al.

8. Conclusion

This thesis demonstrated the application of token-based plagiarism detection to statecharts by extending the plagiarism detector JPlag. We created two language modules for statecharts in different formats and evaluated them using a real-world dataset of statecharts.

The evaluation highlights several factors that influence the ability of a token-based plagiarism detection system to detect possible cases of plagiarism. These include the choice of the language module, the minimum token match parameter, the token extraction strategy and the choice of the sorting strategy.

Our results showed that the SCXML language module outperforms the Create module and a generic module for EMF models. Different token extraction strategies for both language modules were described and evaluated. The handcrafted token extraction strategy with a larger token set proved to be more effective than the simple strategy, indicating the importance of finer-grained types of tokens that capture a larger part of the structure and behavior of a statechart.

The choice of the sorting strategy represents a trade-off between the sensitivity to certain obfuscation attacks and the similarity distribution for original submission pairs. Our novel recursive sorting algorithm achieves immunity against the move attack but increases the effectiveness of an attacker to obfuscate cases of plagiarism by inserting or deleting elements from the statechart.

Despite these results, the evaluation also revealed some limitations of token-based plagiarism detectors when applied to statecharts. This suggests that further work is required to compare the token-based approach with other techniques.

Bibliography

[1] Object Management Group (OMG). *Unified Modeling Language Specification, Version 2.5.1.* 2017, p. 305. URL: https://www.omg.org/spec/UML/2.5.1/PDF.

[2] *A Review on Semantic Similarity Measures for Ontology - IOS Press.* URL: https://content.iospress.com/articles/journal-of-intelligent-and-fuzzy-systems/ifs18120 (visited on 03/27/2023).

[3] Itemis AG. *YAKINDU Statechart Tools Documentation: Quick Reference.* URL: https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/quick_ref (visited on 11/08/2022).

[4] itemis AG. *What is itemis CREATE?* URL: https://www.itemis.com/en/products/itemis-create/documentation/user-guide/overview_what_are_itemis_create_statechart_tools (visited on 03/30/2023).

[5] Alex Aiken. *A System for Detecting Software Similarity.* URL: http://theory.stanford.edu/~aiken/moss/ (visited on 11/22/2022).

[6] *Approximate Nearest Neighbors | Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing.* URL: https://dl.acm.org/doi/abs/10.1145/276698.276876 (visited on 11/20/2022).

[7] Önder Babur, Loek Cleophas, and Mark van den Brand. "Metamodel Clone Detection with SAMOS". In: *Journal of Computer Languages* 51 (Apr. 1, 2019), pp. 57–74. ISSN: 2590-1184. DOI: 10.1016/j.cola.2018.12.002. URL: https://www.sciencedirect.com/science/article/pii/S1045926X18301939 (visited on 11/22/2022).

[8] E. Benowitz, K. Clark, and G. Watney. "Auto-Coding UML Statecharts for Flight Software". In: *2nd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'06).* 2nd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'06). July 2006, 5 pp.–417. DOI: 10.1109/SMC-IT.2006.19.

[9] Sami Beydeda, Matthias Book, and Volker Gruhn, eds. *Model-Driven Software Development.* Berlin, Heidelberg: Springer, 2005. ISBN: 978-3-540-25613-7 978-3-540-28554-0. DOI: 10.1007/3-540-28554-7. URL: http://link.springer.com/10.1007/3-540-28554-7 (visited on 11/20/2022).

[10] Federico Ciccozzi et al. "How Do We Teach Modelling and Model-Driven Engineering? A Survey". In: *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings.* MODELS '18. New York, NY, USA: Association for Computing Machinery, Oct. 14, 2018, pp. 122–129. ISBN: 978-1-4503-5965-8. DOI: 10.1145/3270112.3270129. URL: https://doi.org/10.1145/3270112.3270129 (visited on 03/29/2023).

[11] Georgina Cosma and Mike Joy. "Towards a Definition of Source-Code Plagiarism". In: *IEEE Transactions on Education* 51.2 (May 2008), pp. 195–200. ISSN: 1557-9638. DOI: 10.1109/TE.2007.906776.

[12] Florian Deissenboeck et al. "Model Clone Detection in Practice". In: *Proceedings of the 4th International Workshop on Software Clones.* IWSC '10. New York, NY, USA: Association for Computing Machinery, May 8, 2010, pp. 57–64. ISBN: 978-1-60558-980-0. DOI: 10.1145/1808901.1808909. URL: https://dl.acm.org/doi/10.1145/1808901.1808909 (visited on 03/28/2023).

[13] *Editing State Charts | Qt Creator Manual.* URL: https://doc.qt.io/qtcreator/creator-scxml.html (visited on 03/29/2023).

[14] *Emfatic.* URL: https://www.eclipse.org/emfatic/ (visited on 03/27/2023).

[15] Reza Fauzan et al. "An Automated Statechart Diagram Assessment Using Semantic and Structural Similarities". In: *International Journal on Advanced Science, Engineering and Information Technology* 11.6 (6 2021). URL: https://repository.poliban.ac.id/id/eprint/910/ (visited on 03/27/2023).

[16] David Harel. "Statecharts: A Visual Formalism for Complex Systems". In: *Science of Computer Programming* 8.3 (June 1, 1987), pp. 231–274. ISSN: 0167-6423. DOI: 10.1016/0167-6423(87)90035-9. URL: https://www.sciencedirect.com/science/article/pii/0167642387900359 (visited on 11/16/2022).

[17] H. T. Jankowitz. "Detecting Plagiarism in Student Pascal Programs". In: *The Computer Journal* 31.1 (Jan. 1, 1988), pp. 1–8. ISSN: 0010-4620. DOI: 10.1093/comjnl/31.1.1. URL: https://doi.org/10.1093/comjnl/31.1.1 (visited on 11/20/2022).

[18] *JPlag: Token-Based Software Plagiarism Detection.* URL: https://github.com/jplag/JPlag (visited on 11/16/2022).

[19] Cynthia Kustanto and Inggriani Liem. "Automatic Source Code Plagiarism Detection". In: *2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing.* 2009, pp. 481–486. DOI: 10.1109/SNPD.2009.62.

[20] Salvador Martínez, Manuel Wimmer, and Jordi Cabot. "Efficient Plagiarism Detection for Software Modeling Assignments". In: *Computer Science Education* 30.2 (Apr. 2, 2020), pp. 187–215. ISSN: 0899-3408. DOI: 10.1080/08993408.2020.1711495. URL: https://doi.org/10.1080/08993408.2020.1711495 (visited on 11/20/2022).

[21] Raimundo Santos Moura and Luiz Affonso Guedes. "Simulation of Industrial Applications Using the Execution Environment SCXML". In: *2007 5th IEEE International Conference on Industrial Informatics.* 2007 5th IEEE International Conference on Industrial Informatics. Vol. 1. June 2007, pp. 255–260. DOI: 10.1109/INDIN.2007.4384765.

[22] Shiva Nejati et al. "Matching and Merging of Statecharts Specifications". In: *29th International Conference on Software Engineering (ICSE'07).* 29th International Conference on Software Engineering (ICSE'07). May 2007, pp. 54–64. DOI: 10.1109/ICSE.2007.50.

[23] Matija Novak, Mike Joy, and Dragutin Kermek. “Source-Code Similarity Detection and Detection Tools Used in Academia: A Systematic Review”. In: *ACM Transactions on Computing Education* 19.3 (May 2019). DOI: 10.1145/3313290. URL: https://doi.org/10.1145/3313290.

[24] A. Parker and J.O. Hamblen. “Computer Algorithms for Plagiarism Detection”. In: *IEEE Transactions on Education* 32.2 (1989), pp. 94–99. DOI: 10.1109/13.28038.

[25] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. “Finding Plagiarisms among a Set of Programs with JPlag”. In: *Journal of Universal Computer Science* 8.11 (2000), pp. 1016–1038.

[26] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. *JPlag: Finding Plagiarisms among a Set of Programs.* University of Karlsruhe, Department of Informatics, 2000.

[27] Timur Sağlam. *JPlag Language Module for EMF Model Instances.* URL: https://github.com/jplag/JPlag/tree/emf-model-new/languages/emf-model (visited on 04/01/2023).

[28] Timur Sağlam et al. “Token-Based Plagiarism Detection for Metamodels”. In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings.* MODELS ’22. New York, NY, USA: Association for Computing Machinery, Nov. 9, 2022, pp. 138–141. ISBN: 978-1-4503-9467-3. DOI: 10.1145/3550356.3556508. URL: https://doi.org/10.1145/3550356.3556508 (visited on 11/22/2022).

[29] Alberto Sanfeliu and King-Sun Fu. “A Distance Measure between Attributed Relational Graphs for Pattern Recognition”. In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.3 (May 1983), pp. 353–362. ISSN: 2168-2909. DOI: 10.1109/TSMC.1983.6313167.

[30] *Specification of Factory Dashboard Behaviour Using Statecharts.* URL: http://msdl.uantwerpen.be/people/hv/teaching/MoSIS/202021/assignments/Statecharts (visited on 03/25/2023).

[31] *State Chart XML (SCXML): State Machine Notation for Control Abstraction.* URL: https://www.w3.org/TR/scxml/ (visited on 11/08/2022).

[32] *Statecharts Assignment.* URL: http://msdl.uantwerpen.be/people/hv/teaching/MoSIS/202122/assignments/Statecharts (visited on 03/25/2023).

[33] Harald Störrle. “Effective and Efficient Model Clone Detection”. In: *Software, Services, and Systems: Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering.* Ed. by Rocco De Nicola and Rolf Hennicker. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 440–457. ISBN: 978-3-319-15545-6. DOI: 10.1007/978-3-319-15545-6_25. URL: https://doi.org/10.1007/978-3-319-15545-6_25 (visited on 03/19/2023).

[34] *Variability Mining of State Charts | Proceedings of the 7th International Workshop on Feature-Oriented Software Development.* URL: https://dl.acm.org/doi/abs/10.1145/3001867.3001875 (visited on 04/03/2023).

[35] *Winnowing | Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data.* URL: `https://dl.acm.org/doi/abs/10.1145/872757.872770` (visited on 11/22/2022).

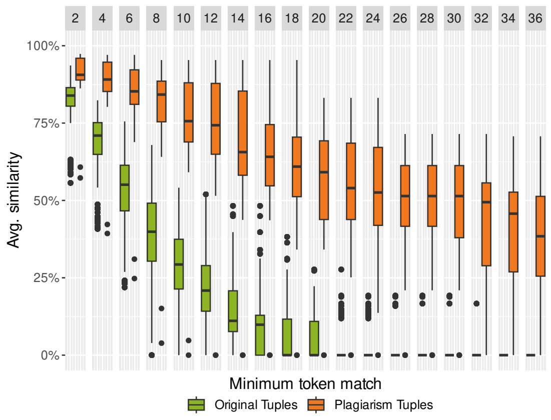
[36] Michael Wise. “String Similarity via Greedy String Tiling and Running Karp-Rabin Matching”. In: *Unpublished Basser Department of Computer Science Report* (Jan. 1, 1993).

[37] *YAKINDU Statechart Tools 3.x.* YAKINDU, Mar. 13, 2023. URL: `https://github.com/Yakindu/statecharts` (visited on 03/30/2023).

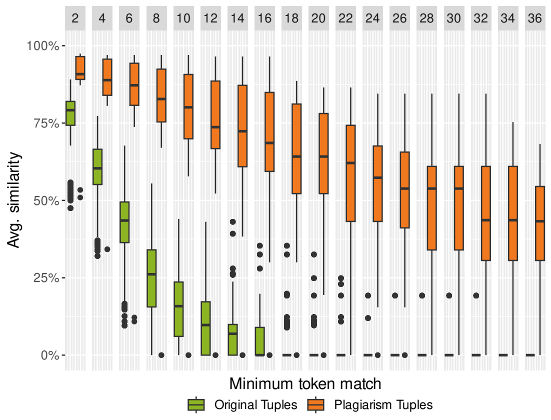
A. Appendix

A.1. Evaluation

A.1.1. Impact of Minimum Token Match

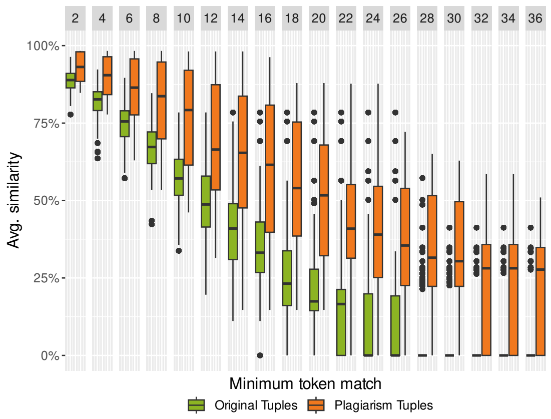


(a) Simple strategy

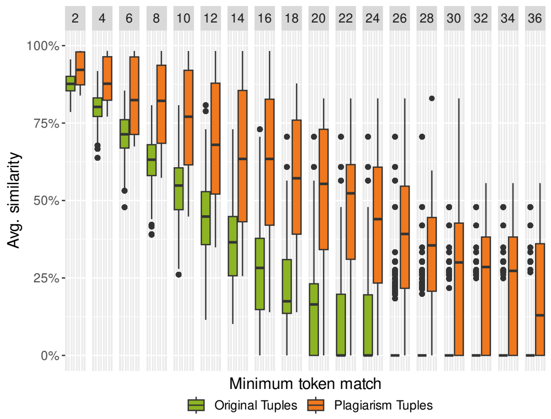


(b) Handcrafted strategy

Figure A.1.: Minimum token match vs. average similarity for the SCXML language module (2021 dataset)



(a) Simple strategy



(b) Handcrafted strategy

Figure A.2.: Minimum token match vs. average similarity for the Create language module (2021 dataset)

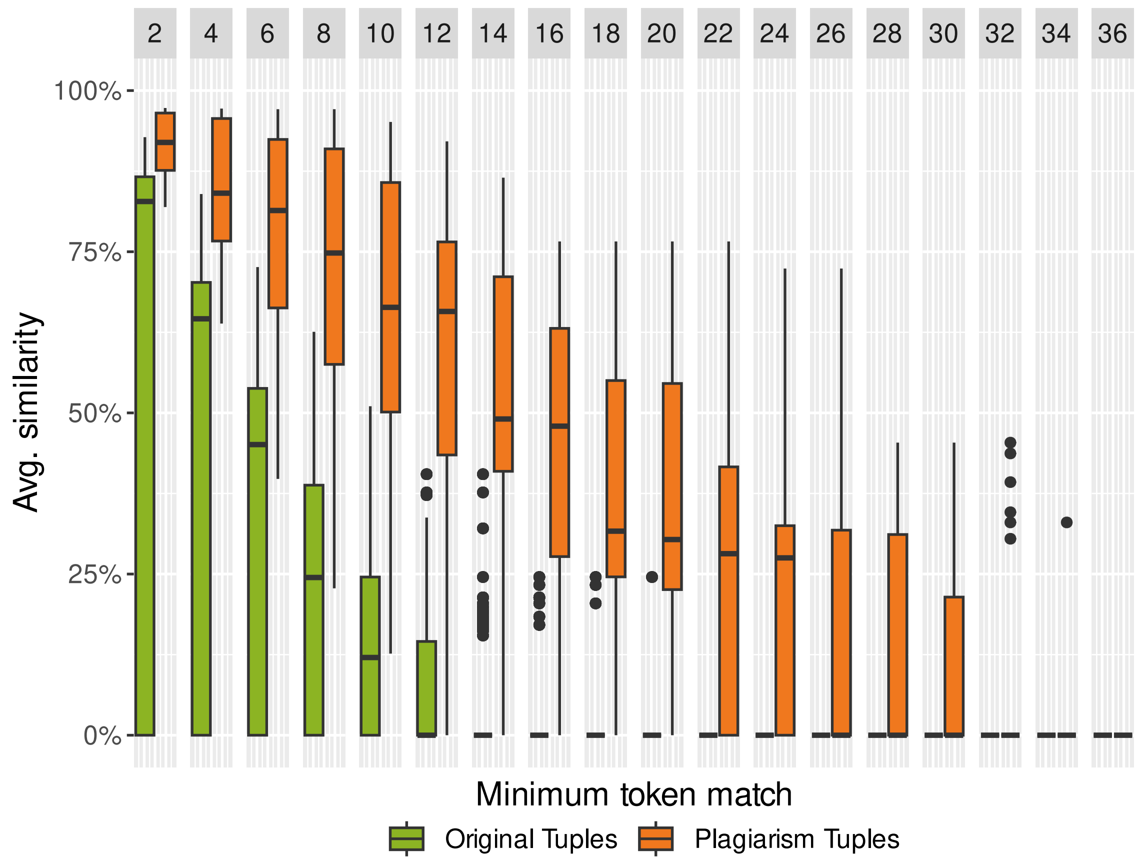
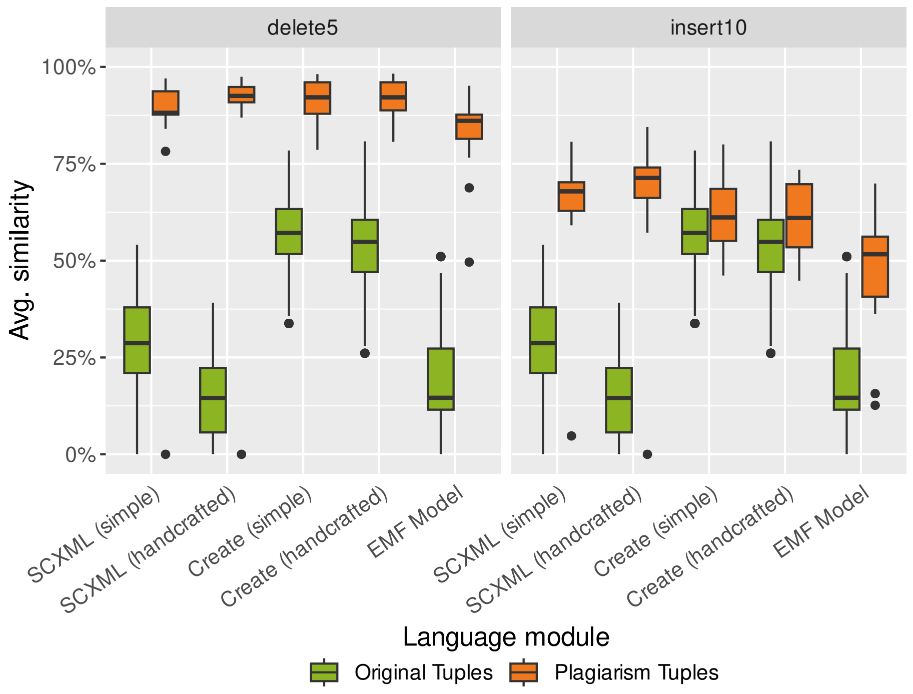
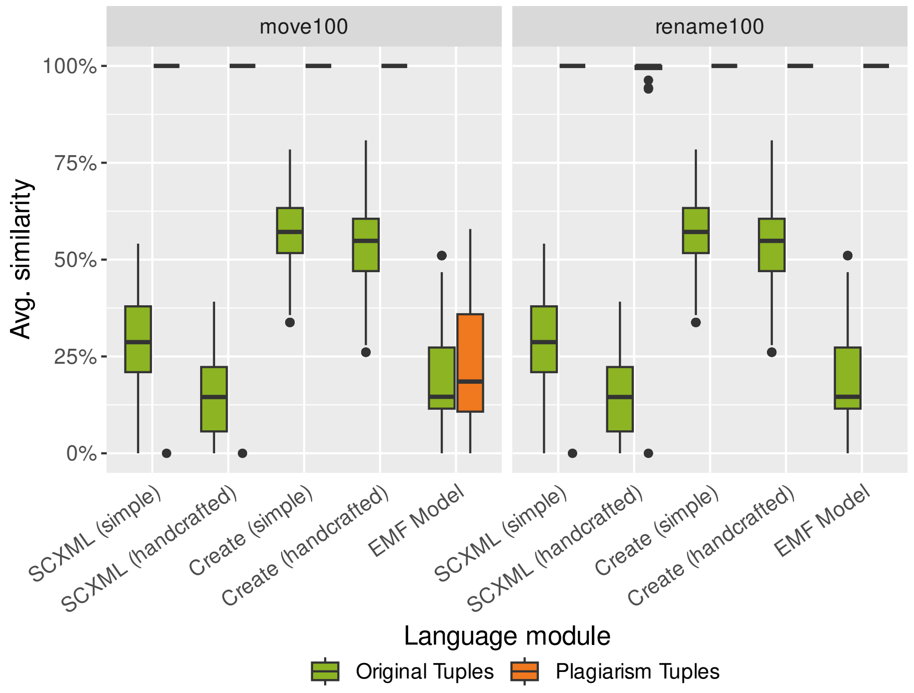


Figure A.3.: Minimum token match vs. average similarity for the *EMF Model* language module (2021 dataset)



(a) Similarity distribution for the insert and delete attacks



(b) Similarity distribution for the move and rename attacks

Figure A.4.: Impact of the token extraction strategy on average similarity for the 2021 dataset (t = 10, recursive sorting enabled)

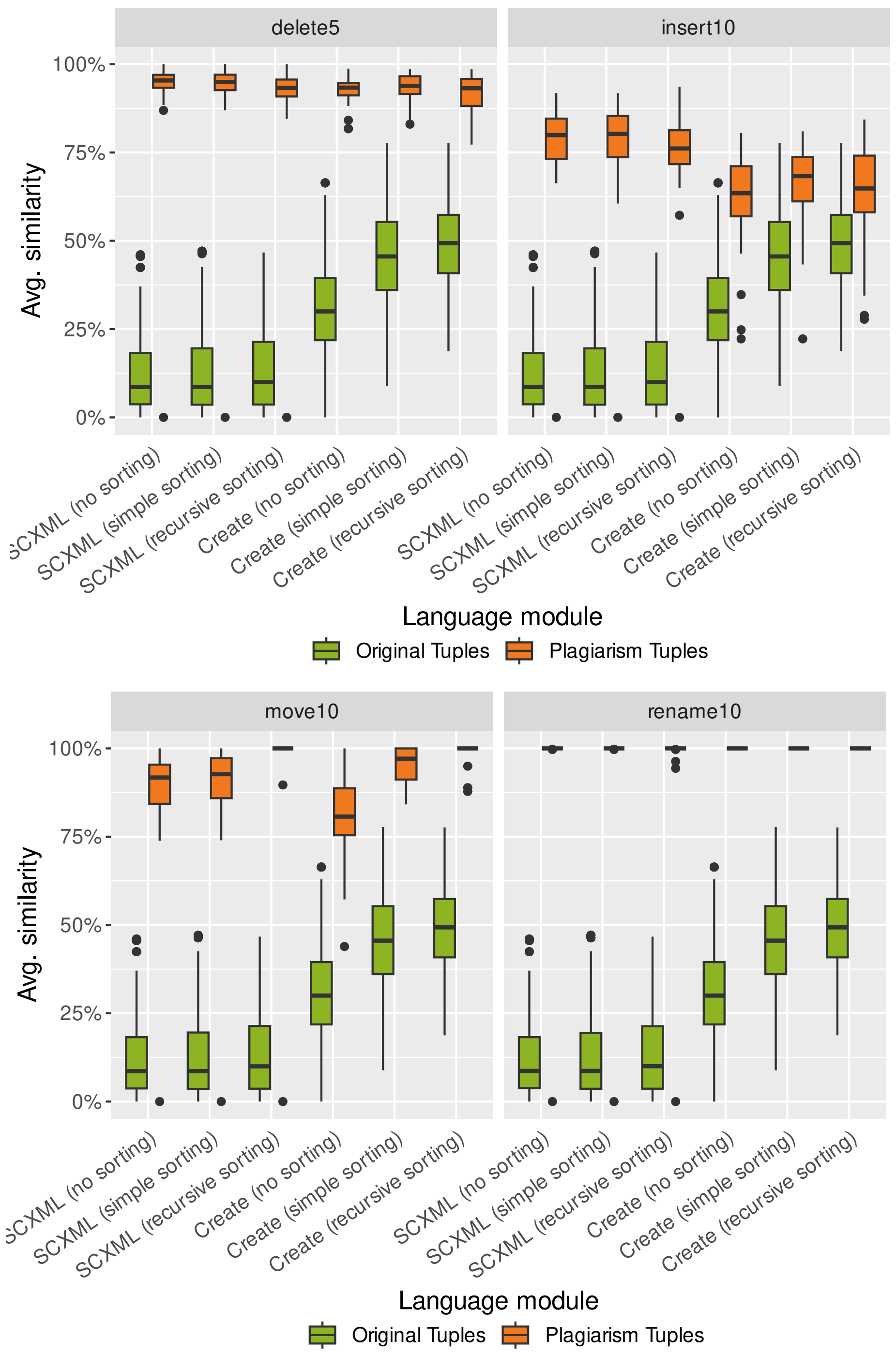


Figure A.5.: Impact of sorting on average similarity to show the resistance against obfuscation attacks ($d = 2021$, $t = 10$, $s = handcrafted$)