AN APPROACH TO CLONE DETECTION IN BEHAVIORAL MODELS

by

ELIZABETH P. ANTONY

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
March 2014

Copyright © Elizabeth P. Antony, 2014
Abstract

In this thesis, we present an approach for identifying near-miss interaction clones in reverse-engineered UML behavioural models. Our goal is to identify patterns of interaction (“conversations”) that can be used to characterize and abstract the run-time behaviour of web applications and other interactive systems. In order to leverage robust near-miss code clone technology, our approach is text-based, working on the level of XMI, the standard interchange serialization for UML. Behavioural model clone detection presents several challenges - first, it is not clear how to break a continuous stream of interaction between lifelines (lifelines represent the objects or actors in the system) into meaningful conversational units. Second, unlike programming languages, the XMI text representation for UML is highly non-local, using attributes to reference information in the model file remotely. In this work we use a set of contextualizing source transformations on the XMI text representation to reveal the hidden hierarchical structure of the model and granularize behavioural interactions into conversational units. Then we adapt NiCad, a near-miss code clone detection tool, to help us identify conversational clones in reverse-engineered behavioural models. These conversational clones are then analysed to find worrisome patterns of security access violations.
Dedication

Dedicated to my loving mother Rose Antony. I miss you dearly, thank you for all you have done for me.
Acknowledgements

This work is supported in part by NSERC, as part of the NECSIS Automotive Partnership, and by the Ontario Research Fund through a Research Excellence grant.

Above all, I would like to give my praises to my Lord Jesus, for all the blessings I have received in my life.

I would also like to express my sincere gratitude to my supervisors Dr. James R. Cordy and Dr. Manar H. Alalfi for their exceptional advice, support and patience throughout the process of completing this work. It has been my privilege to be your student.

Special thanks to Doug Martin and Dr. Scott Grant for sharing their knowledge and expertise. And a big thank you to Doug Martin and Eric Rapos for proofreading my chapters meticulously and providing such valuable feedback.

I would also like to express my thanks to my brilliant defence committee examiners - Dr. Thomas Dean, Dr. Juergen Dingel, Dr. Kai Salomaa - and all my professors for enriching me with so much knowledge at Queens. And to all members of the STL lab Amal, Dr. Aseel, Gehan Selim, Karolina, Paul, Charu, Matthew, Andrew, Das and Mark it’s been a good and memorable 1.5 years.

In addition, to my husband Jimmy and loving son Allen, for their extreme support, patience and selfless love which helped me make it this far. And to my dad Antony and my mom Rose, who sacrificed so much in their lives for their children to be well educated.
Statement of Originality

Parts of this thesis, specifically parts of Chapter 3 and Chapter 4, have been published in an Early Research Achievements (ERA) track at the Working Conference on Reverse Engineering (WCRE 2013) [5]. The paper was co-authored and research was conducted under the supervision of James R. Cordy and Manar H. Alalfi.
Contents

Abstract i
Dedication ii
Acknowledgements iii
Statement of Originality v
Contents vi
List of Tables ix
List of Figures x

Chapter 1: Introduction 1
1.1 Motivation ................................................. 1
1.2 Contributions ........................................... 3
1.3 Organization of Thesis ................................. 3

Chapter 2: Background 5
2.1 UML Modelling Diagrams ............................... 5
2.2 Sequence Diagrams (SDs) ............................... 7
    2.2.1 SD XMI representation ............................ 8
2.3 Source Transformation .................................. 13
    2.3.1 The TXL Source Transformation Language ...... 14
            TXL grammar ................................... 14
            TXL Transformation Rules ....................... 16
2.4 Clone Detection ....................................... 17
    2.4.1 NiCad ........................................... 19
2.5 What is a model clone? ............................... 20
    2.5.1 How are model clones introduced? ............... 21
    2.5.2 Why is it necessary to detect them? ............ 21
### Chapter 3: Overview

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Background</td>
<td>27</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Clones in Sequence Diagrams</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Approach to clone detection</td>
<td>29</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Creating the Contextualised Conversations.</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>Identify</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>Consolidate</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>Contextualize</td>
<td>33</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Extraction and Normalization</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Summary</td>
<td>35</td>
</tr>
</tbody>
</table>

### Chapter 4: Contextualization And Extraction

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Defining a level of granularity</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>Creating the BES Unit and Consolidation</td>
<td>39</td>
</tr>
<tr>
<td>4.3</td>
<td>Contextualization</td>
<td>41</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Filtering</td>
<td>45</td>
</tr>
<tr>
<td>4.4</td>
<td>Extraction and Clone comparison</td>
<td>46</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Specializing the grammar</td>
<td>47</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Clone Comparison</td>
<td>48</td>
</tr>
<tr>
<td>4.5</td>
<td>Summary</td>
<td>48</td>
</tr>
</tbody>
</table>

### Chapter 5: Normalization

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Normalization</td>
<td>51</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Overriding the base grammar for SD XMI representation</td>
<td>52</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Blind renaming attributes</td>
<td>53</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Choosing the right Threshold</td>
<td>58</td>
</tr>
<tr>
<td>5.2</td>
<td>Summary</td>
<td>58</td>
</tr>
</tbody>
</table>

### Chapter 6: Analysis of Clone Detection Results

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Models used in the experiment</td>
<td>59</td>
</tr>
<tr>
<td>6.2</td>
<td>Results of Clone detection in SD models</td>
<td>61</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Results before normalization</td>
<td>61</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Results after normalization</td>
<td>64</td>
</tr>
<tr>
<td>6.3</td>
<td>Summary</td>
<td>66</td>
</tr>
</tbody>
</table>

### Chapter 7: Case Study

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Motivation</td>
<td>67</td>
</tr>
</tbody>
</table>
List of Tables

6.1 Early clone detection results at a difference threshold of 35% 62
6.2 Clone detection results after normalization at a difference threshold of 10% 65
7.1 Cross Clone Experiment Results 76
7.2 List of Actions with their ActionIds reported as violations 77
List of Figures

2.1 Parts of a basic sequence diagram ........................................... 8
2.2 Step I: XMI representation of a SD (The example is only a partial listing of
call elements of a SD) .......................................................... 10
2.3 General grammar for an xmi_element ........................................ 15
2.4 Redefine statement ............................................................... 15
2.5 Example of a TXL token definition (adapted from [8]) .................... 16
2.6 A TXL rule definition (adapted from [8]) ................................... 17
3.1 SD parts and clones ............................................................... 28
3.2 Steps of our approach ......................................................... 30
3.3 An example showing the various elements of a SD XMI representation .. 31
3.4 Step I: Identification of BES fragments in the XMI representation (The example is only a snapshot of the larger SD model) ............... 32
3.5 Example: Consolidation (The example is only a snapshot of the larger SD
model) ........................................................................ 33
3.6 Example: Contextualization (The example is only a snapshot of the larger
SD model) ........................................................................ 34
4.1 An example of SD showing the BES and OS elements with their corres-
ponding Ids from the XMI representation ..................................... 38
4.2 Step I: Identification of BES fragments in the XMI representation (The example is only a snapshot of the larger SD model) .......................... 39
4.3 A portion of a TXL rule to consolidate the conversation .................... 40
4.4 Consolidation [5] ........................................................................... 41
4.5 Elements referenced by elements in the XMI text of SD [5] .................... 42
4.6 Elements referenced by attributes in the XMI text of SD [5] .................... 43
4.7 Snapshot of a contextualized BES unit [5] .......................................... 45
4.8 Defining a bes_fragment ................................................................. 47
4.9 Example showing potential clones reported by NiCad (The example is only a snapshot of the larger SD model) ................................. 48
4.10 Visualisation of a clone pair reported by NiCad ................................. 49

5.1 General grammar for an [xmi_element] ........................................... 52
5.2 Grammar Override for [xmi_element] .............................................. 52
5.3 Example of blind renaming attributes in contextualized unit identified by eventTag (special case highlighted in pink) .............................. 55
5.4 Example of a Potential Clone 1 and Clone 2 to show the difference in lines 56
5.5 Example of a clone pair reported after blind renaming ....................... 57

6.1 An Example of a false positive. ....................................................... 63
6.2 A near miss clone pair reported with 66% similarity .......................... 64

7.1 Cross Clone Experiment ................................................................. 72
7.2 Example of identifying suspicious clone pairs in ForceAnon model ...... 75
7.3 Visualization of tree ....................................................................... 75

A.1 Action Id 152 ............................................................................... 89
A.2 Action Id 157 .............................................. 90
A.3 Action Id 226 .............................................. 90
A.4 Action Id 227 .............................................. 90
A.5 Action Id 228 .............................................. 91
A.6 Action Id 329 .............................................. 91
A.7 Action Id 336 .............................................. 92
A.8 Action Id 341 .............................................. 92
A.9 Action Id 393 .............................................. 93
A.10 Action Id 389 ............................................. 93
A.11 Action Id 390 ............................................. 94
A.12 Action Id 868 ............................................. 94
Chapter 1

Introduction

UML behavioural models, such as sequence diagrams, can be used to represent the complex dynamic interactions of interactive systems such as web applications. Using lifelines to represent concurrent processes such as the user, the browser, the server, the back-end database and various threads within them, sequence diagrams document behaviour as sequences of interactions between the lifelines using events, messages, and other communications. Sequence diagrams can be used in forward engineering to specify intended behaviour, or in reverse engineering to observe and document actual behaviour. In a previous work by Alalfi et al. [1, 2], the run-time behaviour of web applications was reverse engineered to UML Sequence Diagrams (SDs) that describe the entire history of interactions in a web application session. Using an automated test harness based on WATIR [31] to exercise the application in various different roles, behaviour of the application for users in those roles was documented and the behaviour was compared to the behaviour of other roles.

1.1 Motivation

Given the complexity of production interactive web applications, reverse engineered sequence diagrams are often very large, and hence difficult to analyze by hand. In particular,
the identification of repeated sequences of behaviour (conversations) between components is simply impractical to do manually.

In this thesis, we propose an automated approach to analysing such models to identify repeated patterns of similar interactions in the reverse engineered SD models using the near-miss code clone detector, NiCad [10]. NiCad requires a granularity to be specified, which is then used as the unit of comparison to detect clones.

In order to leverage robust near-miss code clone technology, our approach is text-based, working on the level of XMI, the standard interchange serialization for UML. Unlike programming languages, the XMI text representation for UML is highly non-local, using attributes to reference information in the model file remotely. In this work, we use a set of contextualizing source transformations on the XMI text representation to reveal the hidden hierarchical structure of the model and granularize behavioural interactions into conversational units. Clone detection is then applied to a contextualized text representation of the models that compares self-contained hierarchical text descriptions of interaction sequences using source transformations of the XMI interchange representation of the UML behavioural model.

Clone detection in behavioural models has many applications. For example, it can be used to identify similar behaviours in various models with the aim of re-factoring, or to identify locations so that bug updates or change propagation are applied consistently. In this thesis, we have also done a case study in reverse engineered behavioural models from web applications with cross-clone detection (to identify clones across models) to find worrisome patterns of access violations.
1.2 Contributions

This thesis makes the following three contributions:

1. First, we describe a new approach for identifying near miss clones in behavioral models with focus on basic SDs. Our approach is scalable and is the first approach to identify model clones in large reverse-engineered sequence diagrams.

2. Second, we automate the construction and identification of conversations as self-contained conversational units, which are then used as units of comparison for model clone detection. We evaluate our clone detection approach on a number of reverse engineered SD models of variant sizes.

3. Third, we perform a case study of cross-clone detection in behavioural models to identify access violations, by analyzing the conversational clones in reverse-engineered SD models representing interaction sequences of various user roles in an open source web application (phpBB).

1.3 Organization of Thesis

We begin with the background information for understanding our work in Chapter 2. Here we introduce the tools, the programming languages and the elements of a basic Sequence Diagram model in XML-based Metadata Interchange format (XMI) representation. We also introduce the terms that will be used throughout the thesis. Chapter 2 discusses existing work on clone detection in UML models and we introduce our approach to identifying conversational clones in basic SD models in Chapter 3. In Chapter 4, we provide the details of Contextualization and Extraction process with running examples. Chapter 5 describes the need for normalization and the steps taken for the same. Chapter 6 discusses
the results of clone detection in the reverse-engineered SD models, along with brief analysis of the results. A case study of the application of SD model cross-clone detection in identifying access violations is detailed in Chapter 7. Chapter 8 concludes and outlines possible future work.
Chapter 2

Background

In this chapter we provide the background information for understanding our work. We begin by discussing UML modelling diagrams followed by an overview of Sequence Diagrams (SD) and their XML Metadata Interchange representation (XMI). This is followed by an introduction to the terminology and parts of the XMI sequence diagram representation. Then we discuss the source transformation language (TXL) and the NiCad clone detector used in this research.

2.1 UML Modelling Diagrams

Over the past two decades, Model Driven Development (MDD) has gained importance. Software models are being used as main artifacts in the development of a system. UML modelling diagrams are one of the widely used representations for visualising the development of software systems. They can be used to explain business needs, to elaborate user requirements, to assist in the early stages of requirements elicitation, and to facilitate the analysis of the design during later stages of the design and implementation phase. Models can also be used to comprehend a system during reverse engineering tasks. This visual representation makes it easier to understand larger systems. Detecting duplications in the
model can be beneficial in software maintenance activities, re-factoring the duplications or even for reuse purposes.

UML diagrams represent two views of a system model: [32]

Static (or structural) view: emphasizes the static structure of the system using objects, attributes, operations and relationships. The structural view includes class diagrams, object diagrams, component diagrams, composite structure diagrams etc.

Dynamic (or behavioral) view: emphasizes the dynamic behaviour of the system by showing collaborations among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams, use case diagrams, state machine diagrams etc.

The most well-known UML modeling tools are IBM Rational Rose, ArgoUML, BOUML, Dia, Enterprise Architect, MagicDraw UML, Modelio, PowerDesigner, Rational Rhapsody, Rational Software Architect, StarUML and Umbrello. Some development tools such as Eclipse, NetBeans, and Visual Studio also offer UML modelling tools [32]. UML models can be exchanged among the various UML tools by using the standard XML Metadata Interchange (XMI) format [19].

Interaction diagrams are a kind of UML diagram representing the behavioral view. Sequence Diagrams (SD) are a form of an interaction diagram. It is the most common kind of interaction diagram, which focuses on the message interchange between lifelines (objects). Other forms of interaction diagrams include Interaction Overview Diagrams, and Communication Diagrams and other optional diagrams including Timing Diagrams and Interaction Tables. Part II of the UML Superstructure specification [18] defines the Interactions package where different interactions are specified depending on their purpose.
It is clear from the specification that the various elements that represent the basic sequence diagrams are spread across multiple packages. Chapter 14 of the UML Superstructure specification [18] explains the various associations and the packages that the interaction elements of a sequence diagram belong to. This dependency is also reflected in the XMI serialization where the various elements that make up the sequence diagram are grouped accordingly and thus not localized. In the XMI representation of the SDs, this association is also evident in the attribute values of each XMI element. In the next section, we detail some of the elements of a basic sequence diagram related to our work.

### 2.2 Sequence Diagrams (SDs)

UML sequence diagrams (SDs) are 2-dimensional graphical models used to represent the interaction between various objects or actors in a system, encoding the order in which events and message interactions between the actors occur. The diagram communicates what messages and events occur between the lifelines (objects) and more importantly the order in which it occurs. They are mainly used to model the behaviour of web applications and other interactive applications where the sequencing of interactions over time needs to be specified.

UML Sequence diagrams can be used in forward engineering to specify intended behaviour, or in reverse engineering to observe and document actual behaviour. The candidate SD models used in this thesis are obtained from a previous work in web application security analysis, using role-based access control by Alalfi et al. [1, 2]. The run-time behaviour of web applications was reverse engineered to UML Sequence Diagrams (SDs) that describe the entire history of interactions in a web application session. Using an automated test harness based on WATIR [31] to exercise the application in various different roles, behaviour
of the application for the different users in those roles was documented in these models.

The XMI representation of the models is used in our experiments, hence we provide more details of the various elements of the SD model in this representation in the following section. Figure 2.1 shows a small example highlighting the main elements of an XMI representation in a basic sequence diagram. These include Lifelines, Messages, Behavior Execution Specification (BES), Message Occurrence Specification (MOS), and Execution Occurrence Specifications (EOS). Other diagram elements such as Events, Classes, Properties etc. are not shown in the figure for readability.

![Figure 2.1: Parts of a basic sequence diagram](image)

### 2.2.1 SD XMI representation

Sequence diagrams can be represented in text using XMI representation, the XML-based standard interchange format used for exchanging models between various modelling tools [19].

Below we define some of the elements that make up the interaction in a sequence diagram from its XMI representation. Figure 2.2 is a snapshot of the SD shown in Figure
2.1. Interaction units or elements in a sequence diagram are represented in XMI format as “fragments” (highlighted in pink and green). Every element of a sequence diagram has a unique XMI identifier (“xmi:id”), and a set of attributes that shows the relationship of this element with the others. The attribute values have been renamed to aid comprehension.

The definitions of the elements are standard definitions from the UML Superstructure Specification. Each element of a SD is represented by a tag with the attribute xmi:type specifying the kind of UML element it represents.

**Lifelines** represent the object, or actors that participate in the interactions in the system being modelled. This is referred to as the lifeline element and represented as:

```
<lifeline xmi:type="uml:Lifeline" xmi:id="LFLNl2Id" name="l2" represents="PROPl2Id" coveredBy="MOS1Id"/>
```

A particular instance of a class that is participating in the interaction is represented as an `<ownedAttribute>` element of the type `uml:Property`.

```
<ownedAttribute xmi:type="uml:Property" xmi:id="PROPl2Id" name="l2" type="CLSSL2Id"/>
```

The lifeline element and the instances of the lifeline element (Figure 2.2) are highlighted in purple.

**Messages** define a particular communication between the Lifelines of an interaction. A message is associated with two Occurrence Specifications: a sending Occurrence Specification and a receiving Occurrence Specification. This is referenced by the `sendEvent` and `receiveEvent` attributes of the message element as shown below.

```
<message xmi:type="uml:Message" xmi:id="MSG1Id" name="Msg1" messageSort="asynchCall" receiveEvent="MOS2Id" sendEvent="MOS1Id" connector="OC1Id"/>
```

In the XMI representation in (Figure 2.2), the message elements are highlighted in ash
2.2. SEQUENCE DIAGRAMS (SDS)

Figure 2.2: Step I: XMI representation of a SD (The example is only a partial listing of all elements of a SD)
2.2. SEQUENCE DIAGRAMS (SDS)

grey. The messageSort attribute of the message element can take any of the below values:

1. synchCall - The message was generated by a synchronous call to an operation
2. asynchCall - The message was generated by an asynchronous call to an operation
3. asynchSignal - The message was generated by an asynchronous send action.
4. createMessage - The message designating the creation of another lifeline object.
5. deleteMessage - The message designating the termination of another lifeline

For this project, we have come across only one value of messageSort values namely synch-Call.

**Execution Specification (ES)** is a specification of the execution of a unit of behavior or action within the Lifeline. The duration of an Execution Specification is represented by two attributes start and the finish which indicate the starting and the ending element for an Execution Specification. An ES can be either a Action Execution Specification or a Behavior Execution Specification (BES).

**Behaviour Execution Specification (BES)** is a type of ES that indicates the execution of a behavior within the Lifeline. This element has two attributes start and finish indicating the beginning and end of a behavior. We have defined the sequence of OSs that occur between the elements referenced by the start and the finish attributes of a BES in the XMI representation of a SD to form a “conversation.” The BES elements are highlighted in light green in Figure 2.2.

```xml
<fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BES1Id" covered= "LFLNl2Id" start="MOS2Id" finish="EOS3Id"/>
```
Occurrence Specifications (OS) represents occurrence of various events or the moment in time when such events occur along a lifeline. There is a general ordering among the OSs in a sequence diagram. All OSs are ordered along a lifeline. This order is preserved in the XMI representation. There are two kinds of Occurrence Specification we have come across in our work.

Message Occurrence Specification (MOS) specifies the occurrence of events, such as sending and receiving of signals or invoking or receiving of operation or method calls. A message occurrence specification is a kind of message start or end points (Figure 2.1).

The OSs in the XMI representation (Figure 2.2) are highlighted in pink.

Execution Occurrence Specification (EOS) represents moment in time at which actions or behavior execution specifications finish. In the reverse engineered models, EOS elements have a xmi:id value which is referenced by the finish attribute of the particular BES.

The events in a SD represent the run time semantics of an action or behavior.

ExecutionEvent represents the finish of an execution of an action or behaviour.

MessageEvent specifies the receipt by an object of either a call or a signal. We look at two types of Message Events. Both types have an operation attribute that marks the operation that is associated with this event. The Message Events are highlighted in darker pink in Figure 2.2.
2.3. SOURCE TRANSFORMATION

SendOperationEvent (SOE) models the invocation of an operation call. It has an association which is the operation associated with the event. A send operation event specifies the sending of a request to invoke a specific operation on an object.

\(<\text{packagedElement xmi:type=}&\text{"uml:SendOperationEvent" xmi:id}=\text{"SOE3Id" name}=\text{"SendOperationEvent3" operation}=\text{"CLSSL3OO1Id"} />\)

ReceiveOperationEvent (ROE) specifies the event of receiving an operation invocation for a particular operation by the target entity. A receive operation event occurs when an operation invocation is received by the target object. Every SOE has a corresponding ROE and it always occurs before the ROE.

\(<\text{packagedElement xmi:type=}&\text{"uml:ReceiveOperationEvent" xmi:id}=\text{"ROE3Id" name}=\text{"ReceiveOperationEvent3" operation}=\text{"CLSSL3OO1Id"} />\)

2.3 Source Transformation

Source transformation has been used in many software engineering tasks. It involves transforming the original source text to another desirable form for the task at hand. Some examples of its application include translating from one source code language to another, design recovery, and transforming legacy code to a new structured code design. A number of source transformation tools exist including TXL. Various examples of TXL’s application in source transformation tasks are provided in [9] and on the TXL web page [7].

In our case, due to the flat structure of the XMI sequence diagram representation, it was necessary to transform or restructure it, and gather all elements of a conversation into a conversational unit to bring context. This transformation of the input XMI text to a desired form was done using TXL. We briefly describe TXL and some examples of its capabilities in the remainder of this section.
2.3. SOURCE TRANSFORMATION

2.3.1 The TXL Source Transformation Language

TXL [6] is a hybrid functional/rule-based programming language designed to support software analysis and source transformation tasks. The TXL processor works in three distinct phases:

1. **Parse Phase**: In this phase the input is tokenized, and a parse tree generated based on the TXL grammar defined for the input.

2. **Transform Phase**: In this phase, TXL takes the parse tree and produces the desired output based on the rules and functions.

3. **Unparse Phase**: During this phase, it unparses the transformed output and generates the output text based on the formatting specified in the grammar.

Every TXL program has two components [6]: a grammar describing the language(s) used, and a set of transformation rule to specify how to change the input.

**TXL grammar**

A TXL grammar is a specification for the language of the input source text, specified as a directly interpreted Backaus Naur Form grammar, in context-free ambiguous form. Non-terminal and terminals make up the TXL grammar. TXL grammar begins with a program definition - a special nonterminal that defines structure of the whole input source. Nonterminals define the units with which the source input is constructed. TXL defines a number of built-in nonterminals such as [id], [stringlit],[charlit], and [number], which represent identifiers, strings, characters, and unsigned integer and real numbers respectively.

A nonterminal X is specified in a define X... end define block. In our case, we have defined our input program to be a sequence of zero or more of [xmi_element] type as shown
below in Figure 2.3.

```
define xmi_element
    [general_xmi_element]
end define

define general_xmi_element
    < [opt xmi_colon] [id] [repeat tag_attribute] />
| < [opt xmi_colon] [id] [repeat tag_attribute] >
    [xmi_element*]
| < / [opt xmi_colon] [id] >
end define
```

Figure 2.3: General grammar for an xmi_element

Alternate forms of a nonterminal definition is specified using bar symbol ("|") Usually, the alternate forms are used when there is more than one of way representing a nonterminal. Nonterminal can be preceded by a modifier such as repeat, opt, list, push, pop. For example, [repeat xmi_element] means zero or more repetitions of the [xmi_element] will be matched; whereas opt X says nonterminal X is optional.

Nonterminal can be redefined to recognize other forms by providing the new definition within the redefine X ... end redefine block. In such a case, the new definition overrides the original definition.

```
redefine X
    [Y]
    | ...
end redefine
```

Figure 2.4: Redefine statement

In Figure 2.4, the ellipsis “...” means, for the overridden nonterminal X, first the definition Y takes precedence, then the previously defined definition of X is matched.

The tokens statement can be used to add new user-defined input item types as well as to extend or modify the pre-defined nonterminal types with the help of regular expressions. For example, in Figure 2.5, the nonterminal [hexnumber] is defined so that it recognizes
sequences of strings beginning with a 0 followed by upper or lower case X, and a sequence of at least one or more digits or upper and lower case letters A through F.

```
tokens
  hexnumber "0[Xx][dABCDEFabcdef]+"
end tokens
```

Figure 2.5: Example of a TXL token definition (adapted from [8])

Pretty printing in TXL allows the output to be explicitly formatted with the help of built-in formatting nonterminal. These formatting nonterminal provides the capability to automatically format the output based on the formatting specifiers described in the grammar such as spacing, indentation, adding newline, indentation and extending spaces etc.

**TXL Transformation Rules**

The second component of a TXL program, uses rules and functions to perform the desired source transformation. A TXL rule specifies the type of nonterminal in the parse tree, for which it searches for the pattern specified to be replaced with the desired replacement. TXL parses the input as described by the grammar definition and applies the rules recursively to the input until it fails and produces the transformed output. That is, every match of the type in the parse tree is matched for the pattern and transformed with replacement. For a function, only the first occurrence of the pattern in replaced. The output generated is formatted as defined in the grammar.

Rules are specified within the block `rule ruleName .... end rule`. A function is defined in a similar way with the keyword `rule` replaced by keyword `function`.

In the Figure 2.6 the type of the pattern and the replacement must be the same. Rules also use the deconstruct X to break up the patterns into sub-patterns or the deep construct
2.4. CLONE DETECTION

![TXL rule definition](adapted from [8])

\[
\begin{align*}
\text{rule} & \quad \text{ruleName} \\
& \quad \text{replace} \quad [\text{type}] \\
& \quad \quad \text{[pattern]} \\
& \quad \text{by} \quad [\text{replacement}] \\
& \text{end rule}
\end{align*}
\]

Figure 2.6: A TXL rule definition (adapted from [8])

*deconstruct X* to search inside the *nonterminal X*, for a specific pattern.

Besides user defined rules and functions, TXL has a number of built-in functions, and operators, a complete list of which can be found in the TXL reference manual [8]. Sequence operator “.” allows concatenation of two *nonterminal*. For example, \(X[Y]\) results in sequence concatenation of \(X\) with \(Y\). The extract operator “^” extracts all *nonterminals* of a specific type \([T]\) in the input and creates a sequence from them. For example \(X[^Y]\) appends a sequence consisting of every subtree of type \([T]\) contained in \(X\) to the sequence \(Y\) of type \([\text{repeat } T]\).

2.4 Clone Detection

Duplicated fragments of code can appear in software systems by way of copy and paste, bad coding techniques lacking in design, or programmer limitations (such as time constraints, lack of knowledge of the large domain, etc). Clone detection involves identifying sections of code that are similar or identical. Identifying duplications in code has proved beneficial in maintenance activities, improving the code design by re-factoring etc. Code clone detection is an established area of research that involves identifying fragments of code which are exactly or nearly similar [23]. The similarity in the code is defined by a granularity level such as statements, functions, blocks, and classes. Granularity specifies a unit that should be compared during the clone detection process.
In comparison to code clone detection, model clone detection is a fairly young area of research. There are many different kinds of models used for visually representing a software system. These are usually represented as box and arrow type of diagram. With Model Driven Development, models are extensively used in designing software systems. For example, Matlab’s Simulink models are used in automotive modelling. Behavioral models are used to specify the intended behavior or communication process of a system. Thus defining a model clone is a more challenging task. Fortunately, visual modelling diagrams often have a textual representation of the model. Model clone detection may involve identifying similar fragments of modelling elements occurring in the system. It is difficult to come up with a general definition for model clones as it is dependent on the model being used. Alalfi et al. [4] defines 3 types of clones in Simulink models at the subsystem level: Type-1 (exact), Type-2 (renamed), Type-3 (near-miss) model clones. As in the case of code, a granularity level needs to be specified for finding duplication in models: in the case of Simulink models [4], this granularity was system, or subsystem levels, Statecharts and Stateflow charts can set granularity at the chart or state level, for class diagrams granularity can be set at the class level. In our case, for SDs we have used the granularity to mean similar patterns of conversation elements that are part of Behavioral Execution Specification Unit. A number of work on model clone detection exists: near miss clone detection in Simulink models [4], in class diagrams [21], sequence diagrams [15], UML models in general by Störrle [29, 30] to name a few.

We will look at some of the related UML model clone detection work in Section 2.6.
2.4. CLONE DETECTION

2.4.1 NiCad

NiCad [10] is a flexible TXL-based hybrid language-sensitive / text comparison software clone detection system developed by James R. Cordy and Chanchal K. Roy. NiCad was designed for the automated detection of near-miss intentional clones. It is an easy-to-use powerful command line tool. NiCad 3.5 is used in our experiments. NiCad has successfully been used in many projects for finding clones in languages such as C, Java, WSDL documents [16], as well as in graphical Simulink models [4]. NiCad takes, as input, the source (in text form or models represented in textual form) to be analyzed along with configuration files which specify the filtering, threshold and type of normalization to be applied on the input source.

NiCad extracts all potential clones specified by the granularity which are normalized (consistent renaming, blind renaming, filtering), if necessary, to eliminate any unwanted differences to make the comparison process more precise and accurate. Normalization improves the clone results by identifying near-miss clones which would otherwise be not reported. NiCad then compares the extracted potential clones line-by-line using an efficient implementation of the Longest Common Subsequence algorithm.

There are two sets of results generated by NiCad, each reported in both HTML and XML formats. First, the results of the comparison are reported as clone pairs that differ in number of lines upto the specified difference threshold. Second, the clones in the input source are grouped into clone classes. Each clone class contains all the clones in the input which are similar and differ in number of lines only upto the specified difference threshold. Both formats contain source of the clones, specifying the degree of similarity, start and end line numbers of the clones found and the size of the clones.

NiCad provides the ability to find clones at various granularities (classes, functions,
blocks, statements, etc.), with varying degrees of near-miss similarity (e.g., 70, 80, 90 or 100% similar). It can be used either to find clones within a system, or cross-clones between two different systems.

NiCad is a very scalable, and powerful clone detector and the results of the clone detection have high recall and precision.

2.5 What is a model clone?

Different kinds of models and model diagrams are used for different systems. Each model representation is used to model a particular view of the system, hence it is difficult to come up with a general description of what constitutes a model clone. UML diagrams, mentioned previously, serve different purposes in the modelling of the system. Their representation in the UML metamodel specification shows the various model elements that belong to a specific model. Clones can be defined at many levels. It could be at the lowest level such as a smallest diagram element, or even entire entity or interaction. First, we need to identify the purpose for which we are defining a clone. But in general a model clone can be defined as fragments of a model that are identical to each other [29]. Depending on the purpose for which duplicates are identified, granularity or the unit of comparison in a model is used as the basis of comparison.

In our work, we have defined a sequence diagram clone to mean repeated patterns of similar interaction elements in a complete conversation. A Behavior Execution Specification element of a SD defines the start and end of a conversation. Next we will take a look at the different ways model clones are introduced based on the existing literature.
2.5. WHAT IS A MODEL CLONE?

2.5.1 How are model clones introduced?

Clones in models can be introduced in many ways. Model clones are mainly introduced by copy/paste. As Harald Störrle [29] points out, there is often large and distributed teams working on independent copies of shared models. An incorrect merge operation can introduce clones. Poor design lacking in abstraction and time constraints that prevent common functionality from being re-factored also introduce duplicates in models. Similarly, deleting a model element from the diagram may only delete it from the diagram view but not the actual model representation itself. There are, however, some tools that provide the option to delete the element from the model. These are often missed or ignored thus introducing unwanted duplicates into the model. Clones can also be introduced by language loopholes as pointed out by Harald Störrle [30]. Requirements/restrictions on the system being modelled might require replicating the activity or functions in multiple places in the system. In such cases, duplications are not necessarily considered bad and cannot be avoided. Model elements that occur frequently but carry little information, like final activity nodes, generalizations of classifiers, and multiplicity elements with same value specification may appear as clones when they are actually not [30]. In behavioural models, it is important to compare both syntactic as well as the semantic specifications. This is usually harder than finding syntactic clones. Nevertheless, it is important to identify where these duplications in models exist.

2.5.2 Why is it necessary to detect them?

Not all model clones are harmful. There may be situations where duplications appear as part of the modelling language constraints, yet the benefits of model clone detection are big. Clones introduced due to poor design techniques must be detected and removed as these
only increase the size of the system and increase maintenance costs. Detecting clones can help in software maintenance activities where bug updates and changes in requirements to a system must be propagated synchronously to the various parts of the system where similar functionality is present. Examining clones may also possibly improve the design by factoring out common sequences for reuse in the form of library components. As Deissenboeck et al. [12] point out, in automotive manufacturing where software product lines modelling approaches exist, it is often useful to extract the common functionality into reusable library components. Liu et al. [15] mention in their work that if duplications are not detected and changes are not updated, they can result in inconsistencies or even security issues in the resulting systems. Software in embedded systems, such as automotive or washing machines, are installed on environments that place constraints on the resource. In such cases, identifying duplications and re-factoring the models can help reduce the amount of software that is shipped with the product [24, 25]. Rubin et al. [24, 25] identify duplications in individual products with the intent of re-factoring them into product lines with common and variable features.

Based on our work as detailed in Chapter 7, we see that clone detection in a Role Based Access Control (RBAC) model, representing interaction sequences of an anonymous user accessing administrator links, can be used to find instances of unintended behavior in the conversation units.

2.6 Related work on UML model clone detection

In this section, we provide some existing research on clone detection on UML models. Liu et al. [15] have used suffix trees to identify clones in sequence diagrams. Like us, they use BES interactions as the basic elements of comparison, however, they encode each
sequence diagram into an array and then concatenate all the arrays into a Long Array (LA). A suffix tree is constructed for this LA. Their algorithm looks for longest common prefix in the suffix tree to check for duplicates and also ensures that the duplications detected are extractable. Duplicate fragments were refactored if they were considered a bad smell.

Tree comparison has been used by Rattan et al. [21] for finding duplicates in class diagrams from the XMI representation using the DOM’s API and XML parsing.

Rubin et al. [24, 25] work with both structural and behavioural models, specifically class and statechart diagrams. They identify common, variable and optional parts of the input model with the intent of re-factoring input model into product lines.

Störrle [29, 30] talks about challenges and possibilities in clone detection in all types of UML domain models. His work is based on an earlier work on model matching and model querying [28]. He observes that UML models are loosely connected graphs of heavy nodes. He implements graph matching and represents model elements as facts and models as a set of facts in Prolog, then encode Prolog rules to find clones using various similarity heuristics of model elements. Clone detection algorithm and the evaluation of the heuristics, is implemented as MQ\textsubscript{clone} tool, a plugin in MagicDraw UML CASE tool which reports the clones to the user.

Nejati et al. [17] use a match function to compare the input models using both static (structural and textual attributes like element names) and behavioral (to identify element with similar dynamic behaviors) properties of the models to find correspondence between model elements in hierarchical Echarts (a statechart dialect). A merge operator is used to then merge the elements that are similar. For static matching, a combination of typographic, linguistic and depth heuristics are considered to find the similarity values between corresponding state names. For behavioral matching, their algorithm iteratively computes
the similarity degree for every pair of states \((s, t)\) of the input models by aggregating the similarity degrees between the immediate neighbours of \(s\) and those of \(t\). For this their algorithm also compares the transition labels between the states [17]. Overall similarity is obtained by taking the average of both the static and behavioral similarity values. A threshold value selected by the user is used to translate the similarity value into a binary relation. All the state pairs whose similarity values is greater than the threshold are included in the binary relation, and others are left out. The state charts are merged based on the binary relation value after a set of sanity checks. According to the authors, "[their] match and merge algorithms are scalable in terms of high computational efficiency and space." Tool support (TReMer+) is also provided, however, there are still practical limitations for visualizing larger models among others. Their approach requires a domain expert to go over the correspondence relation for more correctness before the merging.

Of these techniques, only Liu et al. [15] and Störrle [30] handle UML 2.0 sequence diagrams, and only Liu et al. [15] also targets conversations. Our work is based on identifying similar patterns in sequences of message interactions using BES in SDs. With contextualization and consolidation steps, the BES units created are complete sequences of interactions and the clones reported are thus extractable as entire conversations. Our work also differs from others in its goal of characterizing and identifying patterns of potential security violations in web applications. None of the other methods have been tested on large models, and only exact (Type 1) clones are handled. By contrast, our work uses a similar approach to the one developed by Alalfi et al. [4] to detect near-miss clones in Simulink models in order to find near-miss (Type 3) clones in SDs. The additional distinction in this work is that UML models in general, and behavioural models specifically, require consolidation and contextualization to localize the representation for comparison. While there is
a number of work done in model comparison and model versioning, these are not included here. Stephan et al. [27] provides a survey of model comparison approaches.

2.7 Related work on clone detection for identifying security vulnerabilities in access control applications

Gauthier et al. [13] use clone detection for identifying clusters of security sensitive code in open source PHP web applications. With the assumption that syntactically similar clones should have similar access control privileges. They hypothesize that clones that do not follow this assumption violate security privileges and report them as security discordant clones.

Radio Frequency Identifier (RFID) tags are small wireless devices that are used to tag peoples and objects. RFID tags can authenticate and identify the tagged object. It is widely used in supply chains to track objects and as a product authentication mechanism, to grant access to highways, ski resorts, buildings, tag drugs in pharmacies, etc. Hence security in these devices is a main concern. Hackers clone RFID tags. Low cost RFIDs cannot implement security mechanisms such as cryptographic keys or the like to maintain its low cost. Cloned RFID tags are a major problem for the companies as it brings with it a huge financial loss and low reputation. Lehtonen et al. [14] investigates an approach to secure these low cost RFIDs from impersonation and cloning attacks by the detection of cloned tags. In our case, we have used cross clone detection to identify patterns of conversations in the administrator model that contain actions (SQL accesses) of administrator level in anonymous user models given access to administrator links (with forced browsing) as detailed in Chapter 7.
2.8 Summary

To summarize this chapter, we began by looking at different views of UML modelling diagrams describing briefly their intended purpose. Then, we described the elements of a basic SD in its XMI representation (the representation that is used in this work). Then, we discussed briefly source transformation, and the TXL language that is used for source analysis and transformation tasks. Clone detection is vastly studied subject and large number of research has been done in this area. We briefly talked about clone detection and text-based clone detector, NiCad. In the later sections of the chapter, we discussed model cloned detection, how they are introduced and why it is beneficial to detect them in most cases. The chapter concludes by highlighting related work on UML model clone detection and work related to the use of clone detection to detect access control vulnerabilities. The overall approach in identifying duplications in basic SDs will be detailed in the next Chapter 3.
Chapter 3

Overview

In the previous chapter, we provided the background information for our work. We discussed clone detection, basic Sequence Diagrams (SD) and their XMI representation. In this chapter, we introduce our approach briefly to identifying conversational clones in SDs. Our approach uses the powerful text based clone detector NiCad and the source transformation language TXL.

3.1 Background

UML sequence diagrams (SDs) are 2-dimensional graphical models used to represent the interaction between various objects or actors in a system, encoding the order in which events and message interactions between the actors occur. They are mainly used to model the behaviour of web applications and other interactive applications where the sequencing of interactions over time needs to be specified. Figure 3.1(a), shows an example highlighting the main elements in a basic sequence diagram. Chapter 2 provides more details on each of these elements. Sequence diagram model-clone detection entails discovering similar or identical sequences of behavioural interaction ("conversations"). Unlike source
code, which is represented as linear text, models are typically represented visually, as box- and-arrow diagrams. Model clones can thus be thought of as similar sub graphs of these diagrams. Figure 3.1(b) shows an example of a potential SD model clone. Here we compare two conversations or sub conversations occurring between the lifelines.

![Elements of a basic sequence diagram](image1)
(a) Elements of a basic sequence diagram

![Example of potential clones](image2)
(b) Example of potential clones

Figure 3.1: Sequence diagram parts and potential clones

### 3.1.1 Clones in Sequence Diagrams

We define clones in SDs to mean repeated patterns of similar or identical interaction elements in complete conversations. A conversation is defined as a sequence of message interactions between various actors during a specific period of time (i.e, within a BES). In our thesis we are interested in identifying repeated conversations, and we define clones in SD from this perspective. Code clones [23] and model clones [4] have been classified according to the types they can identify. Using a similar format we define clones in SD according to the types they can identify as follows.
1. *Type 1 (exact)*: clones have identical interaction elements in a conversation except for variations in visual presentation, layout and formatting.

2. *Type 2 (renamed)*: clones are Type 1 clones that are only renamed (i.e., *Type 1* clones that additionally differ only in identifiers and constant values).

3. *Type 3 (near miss)*: clones that have small differences such as additions, deletions, or modifications in the interaction elements that are part of a conversations. *Type 3-1* clones are identical *Type 3* clones without renaming. *Type 3-2* clones are *Type 3* clones with differences and renaming.

For example, a *Type 3-1* clones are identical conversations in which the same number of messages with the same operation name occurs between same actors. *Type 3-2* clones may contain similar sequences of message interactions between different actors or a clone that may differ for example, only an additional message interaction with an additional actor in the cloned conversation. Next, an overview of our approach to clone detection in basic sequence diagrams is provided.

### 3.2 Approach to clone detection

Our approach to clone detection consists of four main stages. As shown in Figure 3.2, it was necessary to transform the XMI representation into an intermediate contextualized representation to identify conversational units for comparison. The transformation and clone detection is done in four stages. The first stage transforms the XMI sequence diagram serialization into a contextualized form, in which we localize references to reveal the hidden structure of the textual SD representation. This transformation identifies self-contained
3.2. APPROACH TO CLONE DETECTION

units of behavioural interaction that we will use later as the fragments for clone comparison. In the second stage, each self-contained unit of interaction is extracted. The third stage normalizes the extracted contextualized units to remove irrelevant elements and rename irrelevant naming differences to make the process of clone identification more accurate. The final stage uses a standard code clone detector to automatically identify cloned behavioural interactions from the large set of contextualized and normalized interaction units.

In the flat structure of the XMI sequence diagram representation (Figure 3.3), there is little locality; fragments and elements of sub-conversations are spread across the text. XML attributes and the order of elements are used to reference and implicitly group related elements. Behaviour Execution Specifications (BESs), for example, (in Figure 3.3 a BES element highlighted in green), reference the lifeline they are part of using the covered attribute, and the sequence of messages and events of the conversation using the start and
3.2. APPROACH TO CLONE DETECTION

Figure 3.3: An example showing the various elements of a SD XMI representation

finish attributes, and these elements in turn refer to their parts in similar fashion. In order to restructure this scattered representation for comparison purposes, we need to recursively gather the referenced and related elements of the BES conversations together and organize them explicitly into the structures they represent.
3.2. APPROACH TO CLONE DETECTION

3.2.1 Creating the Contextualised Conversations.

The restructuring or the transformation process consists of the following 3 steps: Identify, Consolidate and Contextualize as shown in Figure 3.2.

Identify

This step identifies all the Behaviour Execution Specification (BES) elements in the model representation and restructures them into BES units identified by <BES>...<BES> tags. In the simple example shown in Figure 3.3, there is only one BES element.

Consolidate

Once the tag container for the BES is created, we gather and nest all of the BES’s conversation elements into the container. Each BES element has a start and finish attribute, the ids of which specify the elements of the flat representation that begin and end the BES’s conversation. Due to the general ordering of message, behaviour and event occurrences in the XMI representation, this step primarily involves moving the adjacent elements of the conversation before (represents the message occurrence that start the conversation) and after the BES element until the element that represents the end of the conversation into the <BES>...</BES> unit to consolidate the whole conversation. An example is shown in

![XMI representation of BES elements](image-url)

Figure 3.4: Step I: Identification of BES fragments in the XMI representation (*The example is only a snapshot of the larger SD model*)
3.2. APPROACH TO CLONE DETECTION

Figure 3.5: Example: Consolidation *(The example is only a snapshot of the larger SD model)*

shown in Figure 3.5.

**Contextualize**

Consolidated BES conversations may consist of embedded BESs, Message Occurrence Specifications (MOSs) and Execution Occurrence Specifications (EOSs) describing the conversation’s interactions with other lifelines. Similar to BESs themselves, these use their XML attributes to link to the elements such as messages, types and other lifelines that describe their meaning. The purpose of contextualization phase is to bring all the elements referenced by the attributes into the self contained unit. Contextualization proceeds recursively for these in-lined elements. In-lining the elements of each BES in this way creates a set of self-contained interaction units for comparison during clone detection. More details with examples is provided in Chapter 4. An example of a contextualized conversation is shown in Figure 3.6. We have shown only a snippet of the contextualization for the first MOS element from the consolidated unit. The Contextualization is similar to the work done by Martin et al. [16] in identifying contextual clones in WSDL documents.
3.2. APPROACH TO CLONE DETECTION

3.2.2 Extraction and Normalization

**Extraction** The above three steps represent the main idea of creating a granularity to compare the potential clones existing in basic SD models. An extractor module written for NiCad extracts all potential clones identified by the BES units from the contextualized input file given to NiCad. For this purpose, the general grammar for the SD representation is specialized to define the newly created BES units. Early clone detection at this stage resulted in no clones being extracted even at a 30% difference threshold. Lowering the threshold further to 35% did report clones; however, on analysing the reported clones,
3.3. SUMMARY

some were found to be undesired at the 35% threshold. They were reported as clones due to the similarity in the various elements in the contextualized unit at the bigger difference threshold. Thus it was deemed necessary to lower the threshold and perform some filtering and normalization.

Normalization In order to improve the precision and recall of our clone results, it was required to take a set of filtering steps to remove irrelevant elements from the comparison. Then, as an additional step for normalization, we used blind renaming.

Blind renaming involves renaming all attributes that are not relevant to comparison by a fixed value say “BR”. In NiCad a specialized grammar is used by the TXL renaming module during the parsing phase to distinguish the attributes that need to be renamed from those that should not be renamed. NiCad compares potential clones line by line. Thus by blind renaming attributes that make no difference, we improve the chances of identifying clones that otherwise would go undetected. Blind renaming the irrelevant attributes thus makes these lines similar and would report potential clone (pc) or BES units as a clone to some other similar BES unit.

With the normalization steps and lowering the difference threshold to 10% we were able to report meaningful clones. The exact set of normalization steps taken will be detailed later in Chapter 5.

3.3 Summary

In this chapter, we have given an overview of our work in clone detection in basic SD models. We discussed the non localized XMI representation of the SDs and the steps taken to identify conversational clones from this type of model representation. Currently,
the results we obtain from the clone detector are presented in NiCad’s default XML and HTML text formats. It is easy to trace the conversations to recover them as diagram from the start and finish attributes of the BES elements. For this thesis, we do this step manually however retrieving the original XMI representation from the contextualized representation can be automated with little effort. In the next chapter, we will describe the consolidation and contextualization phase in more detail with the help of an example.
Chapter 4

Contextualization And Extraction

In the previous chapter, we gave a quick overview of our approach to clone detection in Sequence Diagrams (SDs). We saw that the elements that make up a conversation are non localized, hence it is necessary to consolidate and contextualize the elements of the conversation into units which we call BES units. In this chapter, we will see these steps in more detail and provide details on the extraction phase and the early clone detection results.

4.1 Defining a level of granularity

Identifying cloned behavioural interactions in a large scale reverse-engineered SDs poses significant issues of scale. For that reason, we have adapted a highly scalable code clone detector, NiCad [10], to work on behavioural models. As a code clone detector, NiCad was previously used in detecting clones in programming languages such as C#, Java etc. NiCad was originally designed to find clones with well defined granularity or nested structure such as functions and blocks. For modelling languages such as Simulink, a new tool (Simone) was developed, based on NiCad, to identify near miss subsystem clones in Simulink models. Simone, adapted and specialized NiCad in different ways to enable the scalable and accurate manipulation of large scale Simulink models [4].
4.1. DEFINING A LEVEL OF GRANULARITY

Unlike Simulink models, the XMI serialization of UML sequence diagrams does not have an explicit nested structure. Thus one of the main challenges in adapting it to this task was understanding how to reverse engineer the hidden nested structural representation of interaction conversations from the flat representation of the original XMI SD serialization. The other challenge was the identification of the appropriate level of granularity for comparison. In SD conversations, granularity at the message level is very small and would lead to a huge number of clones that would not be useful and probably irrelevant for most applications. On the other hand, comparing the entire conversation of a whole lifeline would likely reveal very few clones and would miss many interesting sub similar conversations. Thus we decided to break conversations into smaller windows at the sub-conversation level. This led us to choose the SD Behavioural Execution Specification (BES) element as the best unit for comparison.

Figure 4.1: An example of SD showing the BES and OS elements with their corresponding Ids from the XMI representation
4.2 Creating the BES Unit and Consolidation

Figure 4.2: Step I: Identification of BES fragments in the XMI representation (The example is only a snapshot of the larger SD model)

In Figure 4.1 we see the various Occurrence Specification (OSs) and the Behavioral Execution Specification (BES) elements with their XMI Ids. The XMI Id is a unique identifier assigned to each element in the SD. Ids have been renamed to aid comprehension. In the Figure 4.1, there are three BES fragments. In the XMI representation (Figure 4.2), we see these three BES elements with start and finish attributes indicating the start and end of sub conversations happening on the corresponding lifelines identified by the attribute covered. It should also be noted that BES1ld and BES2ld are part of the main conversation identified by BES1ld. The XMI representation of this SD is used to explain the consolidation and contextualization process.

4.2 Creating the BES Unit and Consolidation

For each of the BES element identified, we create a container unit identified by <BES>...</BES> tags. We refer each such unit as a BES unit.

The consolidation step brings messages and execution occurrences that are part of the each BES into the corresponding, newly created, BES units. That is, we gather and nest all of the conversation elements of the BES into the container. Each BES element has a start and finish attribute, the ids or attribute values of these attributes specify the elements
4.2. CREATING THE BES UNIT AND CONSOLIDATION

Figure 4.3: A portion of a TXL rule to consolidate the conversation in the flat representation that begin and end the BES’s conversation. Because message, behaviour and event occurrences have a general ordering in the XMI representation, this step primarily involves moving the adjacent elements of the conversation before (representing the messages that start the conversation) and after the BES until the element that represents the end of the conversation into the <BES>...<BES> unit to consolidate the whole conversation (Figure 4.4).

We have used a number of TXL [6] source transformations to implement this restructuring. TXL is a structural transformation system, thus the first step was to define a general grammar for parsing the original XMI representation of the sequence diagrams. Next, a set of source transformation rules were created to identify, consolidate and contextualize BES specifications into self-contained hierarchical units representing the sub-conversations to
4.3 Contextualization

Figure 4.4: Consolidation [5]

compare. Figure 4.3 shows a TXL rule to consolidate the conversation elements for each BES up to the element identified by its finish attribute’s value, FinishId, in the XMI serialization. A consolidated conversation is shown in Figure 4.4.

4.3 Contextualization

As we see from Figure 4.4, consolidated BES conversations consist of embedded BESs, Message Occurrence Specifications (MOSs) and Execution Occurrence Specifications (EOSs) describing the conversation’s interactions with other lifelines. Similar to BESs themselves, these use their XML attributes to link to the elements such as messages, types and other
4.3. CONTEXTUALIZATION

Figure 4.5: Elements referenced by elements in the XMI text of SD [5] lifelines that describe their meaning. Figure 4.5 shows how the attribute for each of the MOS and the EOS in the BES unit references other element in the XMI text. The attributes of these elements in turn reference other elements in the text as seen in Figure 4.6.

Contextualization brings context to the elements that are part of the conversation. That is it brings all the elements –the lifelines, its properties, events and messages represented by the Occurrence Specifications– into the contextualized unit. This helps in finding interaction units that are similar by taking into account all of the properties. This will also produce complete conversation clone pairs when the clone detector does the comparison.
To contextualize a consolidated BES, each MOS and EOS fragment in the BES is converted to a container tag, and the elements referred to by the attributes of the fragment are then in-lined into the container. In this way, the BES becomes an independent self-contained unit with no dependence on its context. For example, MOS fragments have covered, event and message attributes, as shown in Figure 4.6. These attributes represent the covered lifeline, event and message of this particular Message Occurrence Specification. Contextualization proceeds recursively. Thus to in-line the covered attribute of the second MOS in the third embedded BES example of Figure 4.5, the <lifeline> element with id LFN12Id, referred to by the attribute, is located and copied into the container tag.
of the MOS. From the in-lined `<lifeline>` element, the `<ownedAttribute>` element with `xmi:id="PROPl2Id"`, referred to by the lifeline’s `represents` attribute, is then in-lined to include the property/object of the class that the lifeline covers. Similarly, the elements referenced by the `event` attribute is in-lined recursively till there are no more elements left. For the `event` attribute of the MOS, which specifies whether the message occurrence is a send or receive event, we in-line the corresponding `<packagedElement>` with `xmi:id="SOE3Id"` and, from its `operation` attribute, the corresponding `<ownedOperation>` element is then in-lined. Finally, the `message` attribute of the MOS references the corresponding `<message>` element. The `<message>` element, in turn in-lines the sending and receiving MOS of the message with the `sendEvent receiveEvent` attributes, and the `<ownedConnector>` element referenced by the `connector` attribute, which represents the end points of the message where they connect to the lifelines.

Similarly each MOS and EOS fragment in the consolidated BES is converted to a container tag and contextualized by recursively inlining the elements referred to by its attributes in a similar fashion, yielding the completely contextualized BES shown in Figure 4.7.

In-lining the elements of each BES in this way creates a set of self-contained interaction units for comparison in clone detection. Our process of contextualization is very similar to the work of Martin et al. [16] in identifying contextual clones in WSDL documents; where the `<operation>` elements of a WSDL document are contextualized by in-lining each operation description element into Web Service Cells, or WSCells. Similar to WSDL documents the contextualization stage was necessary to consolidate all the conversation elements into a self-contained unit from the XMI representation of SDs.
4.3. CONTEXTUALIZATION

4.3.1 Filtering

In order to tackle the scalability issues and redundant information, it was necessary to do a few filtering steps. The first step taken was to anonymize the `coveredBy` attribute of the `<lifeline>`. This attribute value is a very large concatenated string representing each of the MOS, BES, or EOS occurring on a `<lifeline>` and in the order in which it occurs on it. However, this information does not provide any meaning to the conversation comparison and is thus removed. For extremely large SD models, this value is extremely large and only increases the conversation size. Then during the contextualization phase, we in-line all the elements referred by the attributes except the following attributes:

Figure 4.7: Snapshot of a contextualized BES unit [5]
1. **type:** The *type* attribute of the `<ownedAttribute>` element is not expanded as it would bring in the element’s class and all of its operations which is not necessary. We are only interested in the class and the corresponding `<operation>` elements in the conversation. The `<packagedElement>` referenced by the *event* attribute of MOS has an *operation* attribute that brings in the operation involved in the conversation.

2. **connector:** Similarly, the connector attribute of the `<message>` is not contextualized. The *end* attribute of the `<ownedAttribute>` element inline from a `<lifeline>` element brings the end point of the particular MOS in our case.

3. **execution:** The *execution* attribute of an Execution Occurrence Specification (EOS) refers to the BES element that it part of. Thus expanding this attribute will simply duplicate the information.

### 4.4 Extraction and Clone comparison

**Extraction phase for the NiCad clone detector**  
NiCad clone detector parses and extracts the potential clones to be compared. It comes packaged with number of extractor modules for C#, Java, Python, C, WSDL languages at different granularities such as functions, blocks etc., representing the units to be compared. The extractors are uniformly named in this format: `language-extract-granularity.txl`. The contextualized files are given the extension `.sd`. Thus, the extractor module is named `sd-extract-bes.txl` and is placed in the `/txl` directory. The contextualized models are then input to the NiCad, which uses the extractor module to extract the units specified by the granularity and compares them. This requires redefining the general grammar of a XMI SD element `[xmi_element]` to recognize newly created BES units in the model.
4.4. EXTRACTION AND CLONE COMPARISON

% defining a bes_fragment type.

\begin{verbatim}
define bes_fragment
  [srcfilename] [srclinenumber]
  <'BES [id] = [attvalue] [id] = [attvalue]>
  <fragment xmi:type = "uml:BehaviorExecutionSpecification" [repeat tag_attribute] >
  [xmi_element*]
  </fragment>
  </'fragment>
  [srcfilename][srclinenumber]
</'BES>
\end{verbatim}

end define

Figure 4.8: Defining a bes_fragment

4.4.1 Specializing the grammar

The extractor module requires the grammar to be defined for the newly created BES units as shown in Figure 4.8. The definition for the new contextualized representation of \textit{bes \_fragment} is enclosed in \texttt{define bes \_fragment ....end define} block. The original definition for a \texttt{xmi \_element} is modified to recognize \texttt{bes \_fragment} and other forms.

\textbf{Extraction} The script written for NiCad is placed in the “scripts” folder and when NiCad is invoked with the command “\texttt{./nicad3 bes sd /directory}”, it will look for the extractor module of the specified granularity to extract \texttt{bes} units in \texttt{sd} models located in the specified directory.

An example of the extracted potential clones is shown in Figure 4.9. The TXL transformation rules for the extractor uses the new definition for bes unit in the grammar to extract the potential clones (pc). Each pc is surrounded with \texttt{<source>} tags with file name, startline(line number where the BES unit starts), endline( line number where the BES unit ends) as the attributes.
4.5. SUMMARY

In this chapter, we saw the contextualization phase in more detail. We talked about the extractor written for the contextualized SD representation. The necessary filtering steps to
Figure 4.10: Visualisation of a clone pair reported by NiCad

remove irrelevant elements to improve scalability were highlighted. In the next chapter, we will look at the normalization steps specific to our application to remove unwanted clone pairs.
Chapter 5

Normalization

In previous chapters, we discussed the steps taken to create conversational units (namely BES units) to identify near miss clones. In NiCad, after the extraction phase, the extracted fragments can be normalized if necessary and compared. Normalization is a necessary stage to improve the precision and recall of the clone detection results. In this chapter we highlight the normalization steps used in our method and their effect on enhancing results. The need and amount of normalisation is based on the context of different SD models on which we are performing clone detection. That is, the normalization steps may differ based on application of the clone detection in these models. Either way, normalization will improve recall and report more clones that would otherwise be missed by the clone detector.

As explained in this chapter, our normalization has been additionally refined to improve precision from the perspective of our application of clone detection to identify security risks or access violations as detailed in Chapter 7, with focus on SQL database access actions performed by users with different privilege levels. The normalization step can be tailored to the SD models; thus for other applications it may be different depending on the
information we are interested to compare.

5.1 Normalization

After contextualizing the candidate SD models (Chapter 4), we conducted experiments on these contextualized models at various difference thresholds (10%, 20%, 30%). No clones were reported at these values due to the differences in each line. Increasing the threshold to 35% did report a number of clones, however there were a few unwanted clone pairs. Thus it was considered necessary to do some normalization which included removing some redundant elements and blind renaming. In our case, for basic SD models, it was necessary to do “blind” renaming (e.g., replacing all attribute values of \textit{xmi:id} with “BR”). Because NiCad is a line by line text comparison clone detector using an efficient longest common subsequence algorithm, blind renaming of certain attributes helps to eliminate differences during comparison. Blind renaming involves selectively renaming irrelevant attribute values as recognized by the grammar using transformation rules.

In addition, we removed the redundant \textit{<operation>} elements referred to by the \textit{event} attribute value of \textit{<packagedElement> with xmi:type=“ReceiveOperationEvent”}. These referenced \textit{<operation>} elements are always the same as that referenced by the corresponding \textit{<packagedElement> with xmi:type=“SendOperationEvent”}. This normalization further improves the size of the normalized extracted potential clones and removes redundant elements from comparison.

The generic renaming algorithm that is packaged with NiCad could not be used for Sequence Diagram (SD) models as it renames all attributes with a constant value. Selectively blind renaming attribute values of specific elements of a BES unit for a SD model is not included. Hence it was necessary to write a custom renaming plug-in for basic SD models.
TXL’s agile parsing technique uses a custom variant (“override grammar”) of the base SD grammar designed to distinguish attribute values to be renamed from the ones that should not be renamed. This sort of renaming help us to find all of the near miss and exact clones at the specified difference threshold, thus improving precision and recall.

5.1.1 Overriding the base grammar for SD XMI representation

TXL grammar overrides allows us to override the general base grammar for an \([xmi\_element]\) shown in Figure 5.1 to recognize some elements separately.

```
define xmi_element
  [general_xmi_element]
end define

define general_xmi_element
  < [opt xmi_colon] [id] [repeat tag_attribute] />
| < [opt xmi_colon] [id] [repeat tag_attribute] >
  [xmi_element*]
| < / [opt xmi_colon] [id] >
end define
```

**Figure 5.1: General grammar for an \([xmi\_element]\)**

This is done using the `redefine ... end redefine` statement. Redefine statements also use extension “...” to refer to the overridden nonterminal before the redefinition. In Figure 5.2, the extension “...” is a pre-extension where the additional grammatical form takes precedence over the original form [11]. The overridden grammar is shown in Figure 5.2.

```
redefine xmi_element
  [bes_fragment]
  | [operation_element]
  | [owned_operation]
  | [owned_rule]
  | [specification_element]
  | [owned_parameter]
  | [defaultValue_element]
  | ...
end redefine
```

**Figure 5.2: Grammar Override for \([xmi\_element]\)**
This shows the nonterminal [xmi_element] has been redefined to recognize [bes_fragments], [operation_element], [owned_operation], [owned_Rule], [specification_element], [owned_parameter], [defaultValue_element] nonterminal definitions first. The newly defined non terminals represent the operation elements of XMI SD referenced by event attribute of message element. This overriding helps in choosing the specialized grammar definition to be used in source transformation tasks; specifically blind renaming in this case.

5.1.2 Blind renaming attributes

Every element in a XMI representation of SD model is identified by a unique identifier xmi:id followed by set of attributes specific to the element.

`<lifeline xmi:type="uml:Lifeline" xmi:id="00L" name= "Browser Session" represents="Prop00" coveredBy="44585SPage" />`

For example, in the lifeline element, xmi:type specifies the type of UML element it represents, followed by the unique identifier for the lifeline element represented by xmi:id, and a set of attributes specific to the lifeline element. These attributes are defined as [repeat tag_attribute] in the grammar. The name attribute of the lifeline indicates the class to which the lifeline belongs and so on. We keep the attribute values that provides meaning in the conversational unit. The generated identifiers for each XMI element such as xmi:id and other irrelevant attributes that reference these identifiers in the contextualized XMI representation are blind renamed except the values for the following attributes:

1. name: Every XMI element of a basic SD besides the Occurrence Specifications has a name attribute which is important. For example, name of a message, event, operation etc.

2. type: This specifies the type of UML element, such as uml:Property, uml:Class,
5.1. NORMALIZATION

uml:Message etc. The tag name, in most cases reflect the UML element it belongs to, however there are cases where MOS, EOS, BES are all referred as fragment. Similarly, uml:SendOperationEvent, uml:ReceiveOperationEvent, uml:Class type elements are represented with <packagedElement>. The xmi:type attribute will distinguish the kind of UML element and for the same reason, we preserve these attribute values as well.

3. role: The role attribute represents the object/actor that is involved in the conversation.

All other tag attribute values that does not add any meaning during potential clone comparison are thus replaced by a constant value “BR”. The use of grammar overrides makes it easier to selectively “blind” rename attributes. The effect of blind renaming on a small section of the contextualized unit, namely the operation element, is shown in Figure 5.3.

Here all the tag attribute values except name and type have been renamed. There is also a special case (highlighted in pink) for one of the <ownedParameter> element where the attribute value for xmi:id follows the pattern “AcTS*”. For this special case, we blind rename the name attribute as well. This is highlighted in Figure 5.3. This value represents the time stamp information as to when the action identified by the <ownedParameter>, with xmi:id value of AcID* took place and will be different each time the action is executed, irrespective of the action itself. Hence, we want to eliminate this difference. The TXL tokens statement is used to modify the input item types, in this case the xmi:id attribute value representing the time stamp information. We use tokens statement to specify this pattern and then using grammar overrides we create a new definition for this tag attribute value and <ownedParameter> containing this tag attribute. Then, transformation rules that recognize this new definition for <ownedParameter> are used to blind rename the name attribute for
5.1. NORMALIZATION

(a) Before blind renaming

```
<eventTag event="SendOp1252">
  <packagedElement xmi:type="uml:SendOperationEvent" xmi:id="SendOp1252" name="SendOperationEvent1252" operation="ownedOp1252">
    <operationTag operation="ownedOp1252">
      <ownedOperation xmi:type="uml:Operation" xmi:id="ownedOp1252" name="Select(search,t . topic_id IN ($search_results) ...">
        <ownedRule xmi:type="uml:Constraint" xmi:id="52045" name="Select" constrainedElement="ownedOp1252">
          <specification xmi:type="uml:OpaqueExpression" xmi:id="Const52045" name="t . topic_id IN ($search_results) and t . topic_poster ...">
            <ownedParameter xmi:type="uml:Parameter" xmi:id="AcID52045" name="342"/>
            <ownedParameter xmi:type="uml:Parameter" xmi:id="PN52045" name="324"/>
            <ownedParameter xmi:type="uml:Parameter" xmi:id="AcTS52045" name="1262771841"/>
            <ownedParameter xmi:type="uml:Parameter" xmi:id="AcFromList52045" name="phpbb_topics t, phpbb_forums f, phpbb_users u, phpbb_posts p, phpbb_users u2"/>
            <ownedParameter xmi:type="uml:Parameter" xmi:id="AcRt52045" name="t.*, f.forum_id, f.forum_name, u.username, u.user_id, u2.username as user2, u2.user_id as id2, ..."/>
          </ownedRule>
        </ownedOperation>
      </operationTag>
    </packagedElement>
  </eventTag>
```

(b) After blind renaming

```
<eventTag event="BR">
  <packagedElement xmi:type="uml:SendOperationEvent" xmi:id="BR" name="SendOperationEvent1252" operation="BR">
    <operationTag operation="BR">
      <ownedOperation xmi:type="uml:Operation" xmi:id="BR" name="Select(search,t . topic_id IN ($search_results) ...">
        <ownedRule xmi:type="uml:Constraint" xmi:id="BR" name="Select" constrainedElement="BR">
          <specification xmi:type="uml:OpaqueExpression" xmi:id="BR" name="t . topic_id IN ($search_results) and t . topic_poster ...">
            <ownedParameter xmi:type="uml:Parameter" xmi:id="BR" name="342"/>
            <ownedParameter xmi:type="uml:Parameter" xmi:id="BR" name="324"/>
            <ownedParameter xmi:type="uml:Parameter" xmi:id="BR" name="BR"/>
            <ownedParameter xmi:type="uml:Parameter" xmi:id="BR" name="phpbb_topics t, phpbb_forums f, phpbb_users u, phpbb_posts p, phpbb_users u2"/>
            <ownedParameter xmi:type="uml:Parameter" xmi:id="BR" name="t.*, f.forum_id, f.forum_name, u.username, u.user_id, u2.username as user2, u2.user_id as id2, ..."/>
          </ownedRule>
        </ownedOperation>
      </operationTag>
    </packagedElement>
  </eventTag>
```

Figure 5.3: Example of blind renaming attributes in contextualized unit identified by event-Tag (special case highlighted in pink)

this particular <ownedParameter> element.

Figures (5.4(a),5.4(b)) show 2 partial snapshots of potential clone pairs, which are missed at difference thresholds 30% and lower due to the irrelevant differences in attribute values in each line as highlighted.
5.1. NORMALIZATION

Figure 5.4: Example of a Potential Clone 1 and Clone 2 to show the difference in lines
Figure 5.5: Example of a clone pair reported after blind renaming
5.2. SUMMARY

By eliminating these differences with blind renaming, it is clear from the snapshot of the textual representation in Figure 5.5 that these are similar potential clones and are reported as clone pairs by NiCad. NiCad adds an additional attribute \textit{pcid} to the \texttt{<source>} which is the identifier assigned to the potential clone.

5.1.3 Choosing the right Threshold

While a difference threshold of 35\% without normalization seems to be a good value with filtering, there were some undesired clone pairs reported. Normalization allows us to lower the threshold, but blind renaming at 20\% produces a large number of clone pairs and grouped them all into one class. For a clone size of 242 lines, a 10 \% difference threshold would mean that the clone detector will find clone pairs which are up to 24 lines different, and 48 lines at 20\% threshold. This may contain uninteresting clone pairs based on the action performed. We chose 10\% as the difference threshold value as it proved to enhance the clone detection results for precision and recall. The precision and recall calculated is based on the experiment that will be discussed in our case study Chapter 7.

5.2 Summary

In this chapter, we discussed the need for normalization and the various steps taken to achieve the same. We explained why we choose 10\% as similarity threshold. The extent of normalization will depend on the context of the SD models. Adapting it to various SD models may require specializing the grammar definition for the concerned element and using transformation rules. In the next chapter, we will discuss the results obtained from clone detection in the reverse engineered SD models.
In this chapter, we discuss the results obtained from clone detection on the reverse engineered SD models. We first provide results without normalization followed by a careful analysis of results after normalization. Most clone detectors require a similarity threshold to be specified during clone detection. NiCad clone detector also requires you to specify the percentage of lines that can be different in a clone pair. This threshold value is referred to as “difference threshold”. This chapter also demonstrates using a brief example the need for normalization and selection of the lower difference threshold.

6.1 Models used in the experiment

As explained in the previous chapters, our approach is different from other existing techniques in its ability to handle large reverse engineered SDs efficiently. The SD models we experiment with in this thesis were obtained from previous work by Alalfi et al. [1, 2] on recovering behavioural (SD) models from dynamic web applications. The models we used here are recovered from multiple execution scenarios of the popular web forum phpBB 2.0 for different user roles.

We worked with 7 models of various sizes, ranging from 752 to 469,356
XMI lines. For each user role, there are two sets of reverse engineered SD models. In each case, the models differ in the size showing the extent of coverage during the visits to the forum for each user role. First, for the anonymous user role we have two sets of reverse engineered SD models, AnonSD1 and AnonSD2. Both models represent the interactions of an anonymous user visiting the forum. They differ in size indicating the extent of coverage. Second, for the registered user role we again have two sets of reverse engineered SD models, RegSD1 and RegSD2. These models represent the interaction sequences of a registered user accessing the forum. Third, we have two sets of models representing execution traces of an anonymous user with forced browsing (that is, an anonymous user attempting to access links accessible to an administrator), ForcedAnonSD1 and ForcedAnonSD2. Finally, the last model represents an administrator user’s role, showing the interaction traces of an administrator accessing the web application forum (AdminSD).

Each model is preprocessed as detailed in Chapter 4 to a contextualized representation and the resulting file is given the extension .sd.

**Reports generated by NiCad** The contextualized model is then input to the NiCad clone detector and the results are reported in both HTML and XML formats along with a log. By default in NiCad, these formats have reports generated displaying the various clones pairs in the model along with start and end line numbers of the clones in the contextualized representation. NiCad reports clone pairs found with and without sources. Along with these another report is generated which shows the the various clones grouped into classes based on the similarity value. The clones grouped in classes are also reported with and without source of the clones (in contextualized XMI text) along with a class identifier (“classid”) assigned to each class. For our analysis we have used the XML report displaying the clone pairs both with and without source. The reader is referred to the work by Roy et al. [22] to
obtain further details about how clone pairs are created and grouped into classes, in general.

Early experiments of contextualization on larger models without any filtering resulted in several performance issues. The contextualization was expensive in terms of time and memory requirements. Thus it was necessary to filter some redundant elements and anonymize certain attributes as detailed in Chapter 4. We present below two sets of results, first before normalization with filtering and second after normalization.

6.2 Results of Clone detection in SD models

The results obtained from clone detection in the SD models is provided in the following sections.

For each set of results in the tables below, the first column indicates the different reverse engineered SD models used in the experiment. The second column indicates the size of the original XMI representation of the SD model (before contextualization) in number of XMI lines. The third column reports the number of Behavioral Execution Specifications extracted as conversational units with granularity specified as BES units in each model. The fourth column reports the number of clone pairs in the model. Two potential clones are reported as a clone pair if the clones in the clone pair differ in number of lines only upto the specified threshold. The details about the how NiCad generates the clone pairs are not detailed here. The fifth column reports the number of classes the clones were grouped into based on the similarity of the clones as specified by the threshold value.

6.2.1 Results before normalization

In our early experiments, we worked with a few reverse engineered SD Models without any filtering steps. This resulted in number of clones due to the similarity in the redundant
information mainly in the coveredBy attribute of the <lifeline> element and the connector attribute of the <message> element. Without the filtering steps such as anonymizing the coveredBy attribute and eliminating the redundant elements referred to by the connector and type attributes the size of the contextualized unit is extremely large and the process is very expensive in terms of memory and CPU time. We do not provide these results as this was of very low accuracy and we were unable to perform the contextualization on the larger models.

Table 6.1 shows the results of clone detection after filtering steps, at a difference threshold of 35%. The higher threshold was necessary as the clone detector did not report any clones at lower thresholds. At the higher thresholds, due to the larger allowable difference, the clone detector reported clones and grouped them into classes based on the similarity of the clones.

<table>
<thead>
<tr>
<th>Model Name</th>
<th># Lines</th>
<th>#BES</th>
<th>#Clone Pairs</th>
<th>#Clone Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnonSD1</td>
<td>752</td>
<td>30</td>
<td>18</td>
<td>7</td>
</tr>
<tr>
<td>RegSD1</td>
<td>5376</td>
<td>223</td>
<td>1311</td>
<td>14</td>
</tr>
<tr>
<td>Forced:AnonSD1</td>
<td>9504</td>
<td>142</td>
<td>407</td>
<td>27</td>
</tr>
<tr>
<td>Forced:AnonSD2</td>
<td>37914</td>
<td>513</td>
<td>9918</td>
<td>44</td>
</tr>
<tr>
<td>AnonSD2</td>
<td>53861</td>
<td>314</td>
<td>3330</td>
<td>27</td>
</tr>
<tr>
<td>RegSD2</td>
<td>455687</td>
<td>954</td>
<td>51334</td>
<td>42</td>
</tr>
<tr>
<td>AdminSD</td>
<td>469356</td>
<td>1232</td>
<td>75947</td>
<td>51</td>
</tr>
</tbody>
</table>

Table 6.1: Early clone detection results at a difference threshold of 35%

While most of the clones generated are meaningful clones, there were some undesired
clone pairs. For instance, we noticed that some “Select” operations were being paired with “Update” operations. For example, in ForcedAnonSD1 model, representing Anonymous User accessing administrator links, we note that BES unit with Potential Clone identifier (pcid) 136, containing the AcId 393, which is the “Select” operation gets paired with pcid 142, containing AcId 404 which is an “Update” operation. This is shown in Figure 6.1. While this is expected at the larger threshold, it is a false positive as we are interested in comparing clones based on the action performed from the perspective of the experiment in finding access violations, Chapter 7. As expected, all instances of these conversations were paired in this manner.

![Figure 6.1: An Example of a false positive.](image)

Another example of a near miss clone pair at 66% similarity is shown in Figure 6.2. Even though they are near miss clones, they differ in the action ids, one with AcId 374 and the other with AcId 375.

Thus it was deemed necessary to "blind" rename some attributes and filter irrelevant
6.2. RESULTS OF CLONE DETECTION IN SD MODELS

Figure 6.2: A near miss clone pair reported with 66% similarity

elements in the contextualized representation of BES unit.

6.2.2 Results after normalization

The steps taken for normalization are detailed in Chapter 5. In brief, normalization involved eliminating the irrelevant differences by selectively blind renaming certain attribute values. With this NiCad was able to find clones which are upto 10% different in XML lines. Besides this, we also filtered out the operation elements for the element with xmi:type="uml:ReceiveOperationEvent". This is redundant as it is the same operation element as in the case of xmi:type="uml:SendOperationEvent". This would remove irrelevant
elements from the comparison process and make it more accurate. The results after selectively blind renaming specific attributes of elements in SD models at a difference threshold of 10% are shown in Table 6.2. The results shows that no clones were reported at 10% without blind renaming due to differences in each line. An example of potential clone pair found after blind renaming was given in Chapter 5 (Figure 5.5). Before normalization and at larger threshold of 35%, the clone report also included very small clone pairs (in terms of number of lines in the BES unit) which are not very meaningful or false positives. These were eliminated after normalization and at low threshold value of 10%. Normalization also eliminated all of the false positive clone pairs such as those shown in Figure 6.1. As expected at low similarity threshold of 10%, some near miss clone pairs reported above in Figure 6.2 are also not reported.

<table>
<thead>
<tr>
<th>Model Name</th>
<th># Lines</th>
<th>#BES</th>
<th>#Clone Pairs</th>
<th>#Clone Classes</th>
<th>#Clone Pairs</th>
<th>#Clone Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnonSD1</td>
<td>752</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>8</td>
</tr>
<tr>
<td>RegSD1</td>
<td>5376</td>
<td>223</td>
<td>0</td>
<td>0</td>
<td>1116</td>
<td>14</td>
</tr>
<tr>
<td>ForcedAnonSD1</td>
<td>9504</td>
<td>142</td>
<td>0</td>
<td>0</td>
<td>298</td>
<td>19</td>
</tr>
<tr>
<td>AdminSD</td>
<td>37914</td>
<td>513</td>
<td>0</td>
<td>0</td>
<td>9384</td>
<td>27</td>
</tr>
<tr>
<td>AnonSD2</td>
<td>53861</td>
<td>314</td>
<td>0</td>
<td>0</td>
<td>3156</td>
<td>24</td>
</tr>
<tr>
<td>RegSD2</td>
<td>455687</td>
<td>954</td>
<td>0</td>
<td>0</td>
<td>38770</td>
<td>29</td>
</tr>
<tr>
<td>ForcedAnonSD2</td>
<td>469865</td>
<td>1292</td>
<td>0</td>
<td>0</td>
<td>60455</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 6.2: Clone detection results after normalization at a difference threshold of 10%

On analyzing the results after normalization, we noticed all of the clone pairs were
paired based on the action performed with a similarity percentage of (96 - 100)%. The action performed is represented as part of the operation elements in the contextualized representation. This extensive blind renaming was necessary for getting more accurate results in finding access violations.

6.3 Summary

In this chapter, we presented results of clone detection on a number of SD models of varying sizes. First, we reported the results of clone detection after filtering some elements during contextualization at a difference threshold of 35%. After analyzing the results we found that clone detection with just the filtering steps at 35% difference threshold provided decent results, except for a few undesired clone pairs from the perspective of our case study to find access violations. In order to eliminate the differences and improve the results, we proceeded with normalization and chose a lower threshold of 10%. In our case we used an extensive set of normalization to eliminate all differences so that the clone detector reports clones with high accuracy based on the action performed to suit the need in identifying access violations as laid out in the next Chapter 7. Generally speaking, the amount of normalization and the value chosen for the difference threshold may vary based on the context of the SD models the application for which clone detection is applied.
Chapter 7

Case Study

In the previous chapter, we discussed the results of clone detection, with and without normalization. In this chapter, we will evaluate our approach to the problem of analyzing or identifying worrisome security patterns in a web application models with lower privilege levels. The experiment involved a reverse engineered sequence diagram model with forced browsing for the detection of access control violations in "conversations" occurring in the model.

The results of the experiment are promising, with 100% recall and 86% precision. The following sections will detail the experiment. We begin by briefly discussing the authentication and authorization mechanisms for web applications and one of the main security risks, namely forced browsing.

7.1 Motivation

Web applications are subject to many security risks. The Open Web Application Security Project (OWASP) lists access control, injection, authentication and session management attacks to be among the top 10 security risks [20]. Most web applications implement some
sort of authentication/authorization mechanism to access resources or to perform some functions. These are usually determined by access control policies set up by the administrator for the web application. Authentication will determine a user’s authenticity by verifying that he/she is who he/she claims to be. Some of the most common methods for authentication include password-based authentication and device-based authentication using devices, such as cards and keys. Other forms of authentication include biometric authentication such as fingerprint scanners, retina scanners, and facial recognizers. Once the user is authenticated, authorization will determine whether he/she can access the resources based on his role. However, there are cases where these security mechanisms fail.

Web applications mainly rely on password-based authentication, once the user is authenticated, access control mechanisms authorize the user to access resources of the application. Many web applications implement the access control by hiding links from the user, depending on the privilege level [26]. This kind of vulnerability is also known as forced browsing. However, this form of obscurity is highly vulnerable to security breaches because attackers bypass the access control mechanisms to obtain these links and access unauthorized pages.

7.1.1 Forced browsing

According to OWASP definition [20],

Forced browsing is an attack where the aim is to enumerate and access resources that are not referenced by the application, but are still accessible.

Attackers use forced browsing to retrieve pages or perform operations that would otherwise require authentication. A bad implementation of an authentication/authorization mechanism can make such attacks possible. Forced browsing is usually found to be common
problem in web applications with different user access control roles. For example, a web application may have different access control roles such as a logged in user, registered user, administrator, anonymous user etc. Access control mechanisms authorize the various users to access the appropriate resources based on their role.

Web application developers often make implicit assumptions of access control and often hide links to obscure access to certain pages. However, it is not a difficult task for a determined and skilled hacker to access these links. By manipulating the parameters in the URL of a web application, he/she can gain access to pages to which they are not otherwise authorized. No matter how much prevention measures are taken, attackers usually find some vulnerability in the application or system to bypass these security measures. The potential impact of forced browsing may include unauthorized access to all administration functions, access to server variables, viewing sensitive data (such as other user’s personal information), or performing dangerous operations. To avoid this problem, user access rights must be restricted to correct privilege level and not by pages available to the user based on the interface or by manipulating the URL in web applications.

7.2 Motivating Example

An example of such a web application with Role Based Access Control (RBAC) is the popular open source internet bulletin board system phpBB. This forum was chosen as the case study as it represents various access control permissions. Like most dynamic web applications, this forum also interacts with a database back-end extensively. Vast amounts of information such as session critical information, and access permissions are stored in the database. The web application developers restrict access to some privileged pages by obscuring the links to such pages. This makes it vulnerable to many forms of security attacks.
In a previous work, Alalfi et al. have reverse engineered the execution traces of various user roles of dynamic web application into the UML 2.1 sequence diagram XMI representation using PHP2XMI framework [1]. The UML-based security model is transformed into a Prolog model and this model is verified to see if the model conforms to specified access control security properties [3]. The authors also identified a list of unauthorized SQL access actions which represent an anonymous user attempting to access the administrators links.

In our work, we use the same set of models and apply cross clone detection to see if we are able to identify the same list of access violations based on the action performed. Each model represents execution traces of a different user with the appropriate control roles. In the cross cloning experiment detailed in the next section, we have used three design recovered sequence diagram models. The first model (AnonUser) represents the execution traces of an anonymous user, the second model (AdminUser) represents the execution sequences for an admin, and the third model (ForcedAnon) represents the execution sequences of an anonymous user, provided with links that only administrators can access while navigating the same forum.

In the next few sections, we will explain the application of NiCad’s cross-clone detector on the models and see how we used it to identify the various cases of access violation occurring in the interaction sequences or conversations in the model.

### 7.3 NiCad cross clone detector

NiCad cross clone (nicad3cross) detector is a command line extension that identifies clones across systems. It takes, as input, the directories to be checked for clones and a configuration file specifying the normalization and filtering to be done, and provides the output
results in both XML form for easy analysis and HTML form for convenient browsing [10]. The order in which the directories are specified determines the clone instances that it searches for. For example:

```
./nicad3cross CrssMdlT10/AdminAnon/Admin CrssMdlT10/AdminAnon/Anon bes sd sdbes-blindreport
```

With the above command, NiCad will find instances of conversation units (bes) that are present in the model placed in the folder `CrssMdlT10/AdminAnon/Admin` that are clones in the second model placed in folder `CrssMdlT10/AdminAnon/Anon`. The input file type is `sd`, which is the extension given for the contextualized sequence diagram model representation. The NiCad cross clone detector will use the configuration file `sdbesblindreport` to determine the difference threshold, the minimum and maximum number of lines in the clones, the type of renaming to be done, whether we wish to cluster the cross clone results into classes based on the similarity, etc.

For each model, the clone detector first normalizes and then extracts the potential clones which are BES units or the conversation units. Each potential clone (pc) extracted in this manner is given a potential clone Id (pcid). For the above case, the pc(s) are numbered incrementally. All pcs in the AdminUser (first model) are numbered in sequential order and then consecutive pcids are assigned to the pcs in AnonUser (second model). There are 1542 potential clones in Admin model numbered 1 to 1542. The Anon model has 314 potential clones numbered from 1543 to 1856.

### 7.4 Cross Clone Experiment

The cross clone detection step is shown in figure 7.1. In the experiment, we first run the cross clone detector on AdminUser model with ForcedAnon model, which will find clone
instances of Admin conversations that are present in the ForcedAnon model. Then, we run a second experiment in which the same Admin model is run with the Anon model. This will give us clone instances of Admin conversations that are present or allowed in the anonymous user model. We then take the difference in extracted clone pairs from both experiments based on the Admin’s Potential Clone Id (pcid). This differencing is done to make analysis easier by narrowing down the Admin conversations existing in the ForcedAnon which would not be allowed by a AnonUser. The resulting list of clone pairs is considered to be potential access violations. The result is then analyzed, to see what action (SQL statement) is executed in these conversations. The actions are validated against the results published by SecureUML model checking in [3]. We want to check if an anonymous user can access any unauthorized content by allowing him access to links that only the
Listing 7.1 is a snapshot of the clone pairs reported by the NiCad cross clone experiment with the source. Common conversations in both models (a clone pair) are enclosed within the <clone nlines=... similarity=...></clone> tags. Each clone in the pair is enclosed within <source></source> tags. The <source></source> tags specify the file location of the model along with the start and end line numbers of where this clone instance is located in the file and the potential clone Id (pcid).

For example, the cloned conversation in the AdminUser model is specified within

```
<source file="CrssMdl10Spt28/AdminSqAnonUsingUNF2/AdminSq/Admin.sd" startline="203106" endline="203351" pcid="1220"> ...
```

Similarly, the second tag provides the name of the second model and its corresponding locations for the clone and its pcid in the clone pair.

```
<source file="CrssMdl10Spt28/AdminSqForcedAnon/ForcedAnon/ForcedAnon.sd" startline="32572" endline="32817" pcid="1692"> ...
```

Besides providing the report with the source of the clone instances, NiCad generates an XML report of the clone pairs without the source. For our analysis, we first use this report to get a list of all the Potential Clone Ids from each of the input models from the cross clone experiment. This list is imported into Excel and each Admin pcid from the (Admin x ForcedAnon) clone pair is looked up to see if the particular Admin conversation, identified by the Admin’s pcid, is not present in the (Admin x Anon) clone pair. The Admin conversations that are not present in (Admin x Anon) are reported as potential access violations. Figure 7.2 shows an example of this step.

These suspicious clone pairs are identified in the NiCad report with source and are then checked to see the actions associated in each conversation. Each action represents a SQL
Listing 7.1: A snapshot of cross clone pair reported by NiCad

```xml
<clone nlines="231" similarity="91">
  <source file="CrssMdl10Spt28/AdmnSqAnonUsingUNF2/AdmnSq/Admin.sd" startline="203106" endline="203351" pcid="1220">
    <BES start="48550R" finish="EventOcc179839">
      <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BehivExec179839" covered="phpbb_forumsphpbb_postsphpbb_posts_textphpbb_topicsphpbb_users" start="48550R" finish="EventOcc179839">
        ...
      </fragment>
    </BES>
  </source>
</clone>

<clone nlines="231" similarity="91">
  <source file="CrssMdl10Spt28/AdmnSqAnonUsingUNF2/AnonUnF2/ForcedAnon.sd" startline="32572" endline="32817" pcid="1692">
    <BES start="52612R" finish="EventOcc9151">
      <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BehivExec9151" covered="phpbb_forumsphpbb_postsphpbb_posts_textphpbb_topicsphpbb_users" start="52612R" finish="EventOcc9151">
        ...
      </fragment>
    </BES>
  </source>
</clone>
```
7.4. CROSS CLONE EXPERIMENT

(a) (1,7) and (5,8) are potential suspicious clone pairs
(b) Identifying the AcIds 226 and 228 in the suspicious clone pairs

Figure 7.2

(a) Action Id 341
(b) Action Id 393

Figure 7.3: Visualization of the patterns involving worrisome access violations

Database (DB) access and is identified by an AcId. The AcId involved in the suspicious conversations are validated against the already published results by SecureUML Model Analysis. Figure 7.3 shows the SD patterns for conversations involving actions with AcId’s 341 and 393. A shows the other 10 patterns. In the next section, we present the results and evaluation.
7.4. CROSS CLONE EXPERIMENT

Table 7.1: Cross Clone Experiment Results

<table>
<thead>
<tr>
<th>Model</th>
<th># Clone Pairs</th>
<th>#Clone Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Admin x Forced Anon</td>
<td>1154</td>
<td>31</td>
</tr>
<tr>
<td>Admin x Anon</td>
<td>3030</td>
<td>19</td>
</tr>
<tr>
<td>Suspicious conversations</td>
<td>62</td>
<td></td>
</tr>
</tbody>
</table>

7.4.1 Evaluation

Table 7.1 shows the results of the cross cloning experiment done with a difference threshold of 10%. For the first experiment, involving Admin model and ForcedAnon model (Admin x ForcedAnon) the NiCad cross clone detector reports 1154 clone pairs. Based on the similarity of the clones, the reported cross clone pairs were grouped into 31 classes. Similarly, the cross clone experiment run with Admin model and Anon model (Admin x Anon) reported 3030 clone pairs and grouped into 19 classes. Out of the clone pairs reported in (Admin x ForcedAnon), 62 suspicious conversations were identified based on the differencing experiment as detailed in Section 7.3. These conversations contain the ActionId’s listed in Table 7.2. This list, identified by cross clone detection, is presented in a similar format as was reported by SecureUML model analysis. SecureUML model analysis reported 12 of these actions to be access violations in ForcedAnon model. Our experiment reported 14 actions to be access violations in ForcedAnon model, 12 of which are the same as reported by SecureUML analysis, but the other 2 were found to be false positives. For calculating precision and recall, we validate the results against the result obtained from SecureUML model analysis and use the following definition.

\[
Precision = \frac{\text{No. of ActionIds reported as violation by SecureUML Analysis}}{\text{No. of Action Ids reported as violations by cross clone detection}}
\] (7.1)
Table 7.2: List of Actions with their ActionIds reported as violations

Recall = \frac{\text{No. of ActionIds that are actually access violations by cross clone detection approach}}{\text{No. of Actions Ids that are violations by SecureUML Analysis}} \quad (7.2)

While the recall is 100%, the precision for the experiment is 12/14 or 86%. This leads to the conclusion that slight optimization may be required for obtaining better precision.

7.5 Discussion

The goal of our experiment was to validate the results we got with the results obtained from SecureUML Model Analysis. Besides identifying potential violations, using a near miss
clone detection approach helped us in identifying multiple instances of such conversations occurring from different pages in both models, as shown in Table 7.2. In this table we only report the number of instances of the cloned suspicious conversations occurring in both the Admin model as well as the Forced Anon model.

Reverse engineered UML SD model clones extracted in this manner can also aid in program comprehension and maintenance. The BES tags \(<BES \text{ start} = "48550R" \text{ finish} = "EventOcc179839"/> \ldots </BES>\), as shown in Listing 7.1, have start and finish attributes which identify the start and end of the Occurrence Specifications (OS) for the conversation. We can consolidate the OS fragments from these attributes. Once this is done the elements represented by the event, message, covered attributes can be used to recover the visual representation of the pattern present in the BES unit or conversation. At present, we manually recover the XMI representation of the pattern and use IBM Rational Software Architect to visualize the pattern as a SD. This process can however be automated.

NiCad [10] has been successfully used in many projects for finding clones in source code in languages such as C, C#, Java, Python and WSDL. Alalfi et al. [4] developed SIMONE, an extension to NiCad clone detector to find near miss system and subsystem level clones in graphical Simulink Models by transforming the graph based models to the normalized textual form. Our experiment comes as a result of our curiosity to see if NiCad can be used to find clones in UML behavioral models and the subsequent application of the same to find access violations in these model sets. This technique of detecting access violations in web applications is robust. We have worked with models of varying sizes and with proper filtering, our clone detection approach is found to be scalable to extremely large models in contextualized representation. From our experiment, we found the precision and recall also
to be high. Near miss cross clone detection in this manner on the reverse engineered behavioral models, will help identify a number of instances of interaction sequences of Admin or any other user model with higher privilege level in models that have a lower privilege level. It is also the first security analysis technique that uses near miss clone detection on reverse engineered UML models. This method of cross cloning also helps in model comprehension, analysis of conversations in reverse engineered SD representation in web application for security risks or violations. Besides this, the complete conversation units can be recovered to sequence diagram to visualize the security access pattern violation.

7.6 Summary

In this chapter, we have briefly discussed web application authentication and authorization mechanisms, followed by a brief explanation of forced browsing and the security risks associated with it. We then detail our cross cloning experiment using NiCad cross clone detector to identify instances of Admin access patterns occurring in the Anon model provided links that an administrator can access. We then validated our results with the results obtained from SecureUML Analysis. Chapter 8 concludes and outlines possible future work.
Chapter 8

Conclusion and Future Work

8.1 Summary

The NiCad clone detector [10] has been successfully used in finding clones in many source code languages. It is a text based clone detector which requires a specified granularity (a unit of comparison) which occurs naturally in most source code languages. Examples of granularity in source code languages include functions, blocks (ex. an if then block), statements, or even a class level granularity for Object Oriented languages. For modelling languages such as Simulink, a new tool (Simone) was developed, based on NiCad, to identify near miss subsystem clones in Simulink models. Simone adapted and specialized NiCad in different ways to enable the scalable and accurate manipulation of large scale Simulink models [4].

The work in this thesis, comes out of our interest in finding how NiCad can be applied to find clones in UML behavioral models, called Sequence Diagrams (SD). We work with reverse engineered SD models in their XMI representation. The models represent the execution traces of different users in various access control roles, navigating the popular open source phpBB forum. The XMI representation of the SD models has a flat structure with
no immediate granularity visible to be used as the unit of comparison for clone detection. Thus it became necessary to identify a level of granularity for the SDs. Upon analyzing this representation, we found that all the Occurrence Specifications for the messages and the Behavioral Execution Specification are ordered in the sequence diagram model. Since the duration for the execution of a behavior is represented by the Behavioral Execution Specification element, it led us to propose and automate the creation of a Behavioral Execution Specification unit (BES unit) to be the granularity that is required for clone comparison. We interchangeably refer to this unit as a conversation.

The BES unit is created by a set of transformations for consolidating the conversation elements that are part of this Behavioral Execution Specification and through contextualization, in-line all the elements that are referenced by the attribute of these elements into the BES unit. A set of filtering steps were necessary to remove irrelevant elements and attributes during contextualization. This was necessary to reduce the size of the contextualized representation and thus tackle the memory and time constraints for the very large SD models. This contextualized representation of the models is then input to NiCad, which uses the extractor module written for SD, to extract the potential clones (which are BES units), and does the clone comparison to generate the clone report. NiCad requires a difference threshold to be specified which indicates the maximum number of lines that can be different in the clones. The contextualization is very similar to the work done by Martin et al. [16].

Besides identifying conversational clones, we have done a case study to identify access violations in an anonymous user SD model indicating forced browsing (ForcedAnon model) by analyzing cross-model conversation clones existing between an Admin and
8.1. SUMMARY

Forced Anon (Admin x ForcedAnon) and verifying it against cross-model conversation clones in Admin and Anonymous User Model (Admin x Anon). The ForcedAnon SD model represents the execution sequences of an anonymous user provided with links that only an administrator (admin) can access while navigating the same forum.

In the first experiment, we run the cross clone detector (plug-in available with NiCad) to find the instances of admin conversations that is present in the anonymous user model with forced browsing. In a similar way, we run a second experiment to find cross-model conversations common in Admin and Anonymous User model. The difference list of clone pairs that indicates the suspicious conversations. This list is then checked for the action or operation being executed in each case and are validated against the results published by Alalfi et al. [3] with SecureUML Model Analysis. Each action represents a SQL DB access which are identified by an AcId in the contextualized representation. Since we are interested in comparing exact actions or operations, it was necessary to selectively blind rename all irrelevant attributes and remove certain redundant elements from the comparison process to make it more accurate. A blind renaming module was also written for this task. Blind renaming allows us to set the difference threshold to lower value, as is in this case of 10%. The results of this experiment are very promising with 100% recall and 86% recall.

We automate the process of contextualizing the elements that are part of a conversation into a Behavioral Execution Specification unit (BES unit). Also an extractor module was written to extract potential clones (BES units) for clone comparison for NiCad. NiCad also has the advantage of setting the minimum and maximum size for the extracted potential clones for comparison. We evaluate our approach on a number of reverse engineered SD models. NiCad being a near miss clone detector, we are able to leverage this capability to identifying near miss clones in basic SDs. The extracted potential clones represent each
complete sequence of interaction as it appears in the reverse engineered models. A list of access violations were identified and was validated against the published results by Alalfi et al. [3].

8.2 Limitations

While we find the precision and recall of our experiments to be high, we admit that more variety of SD models may be necessary to include all possible variations that may occur in a SD model. However this limitation can be easily overcome by adapting the SD XMI grammar to include this new element. Similarly if necessary, the transformation rules can also be changed to include any new requirements. We chose the difference threshold value and normalization steps to find access violations in the candidate reverse engineered models. These can also be varied if necessary depending on the task at hand. The clone pairs reported by NiCad are still in the XML and HTML format and the process of visualizing the clones is manual. Our approach is great method for identifying access violations in reverse engineered SD models, however it may not be applicable for a real-time detection of these violations. We have worked with only basic Sequence Diagram models.

8.3 Future Work

Our future work comes about from the limitations mentioned in the previous section. We see a number of possibilities for future work.

- Work with other UML behavioral models such as State Charts.
- Test our approach on more variety of SD models, in particular advanced SDs.
- Automate the process of visualising the clones reported in the clone pair.
• Test the approach on SD models designed during forward engineering with the aim of re-factoring to improve the design and thus improve maintenance activities.

In conclusion, we leveraged the capabilities of NiCad, a very scalable and robust clone detector to automate the process of finding repeated sequences of interactions in the reverse engineered SD models. We have worked with models of varying sizes and with proper filtering of irrelevant attributes and removing redundant elements, our clone detection approach is found to be scalable to extremely large models in contextualized representation. It was also found that due to the differences in each line a higher threshold of 35% was necessary for NiCad. Normalization in the form of blind renaming irrelevant attributes allows us to lower this threshold. However, normalization is optional, and the selection of difference threshold and it will depend on the task at hand. From our experiment, we found the precision and recall also to be high. Near miss cross clone detection in this manner on the reverse engineered behavioral models, will help identify number of instances of interaction sequences of Admin or any other user model with higher privilege level in models that have lower privilege level. It is also the first security analysis technique that uses near miss clone detection on reverse engineered UML SD models. We have also manually recovered the interactions or conversation for each of the access violation list identified.
Bibliography


[16] Douglas Martin and James R. Cordy. Towards web services tagging by similarity

[17] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela
Zave. Matching and merging of statecharts specifications. In In 29th International

[18] OMG. UML superstructure specification, http://www.omg.org/spec/UML/2.4.1/Super-
structure/PDF, last access February 2014.


access November 2013.


[22] Chanchal Kumar Roy and James R Cordy. Nicad: Accurate detection of near-miss
intentional clones using flexible pretty-printing and code normalization. In ICPC

[23] Chanchal Kumar Roy, James R. Cordy, and Rainer Koschke. Comparison and eval-
uation of code clone detection techniques and tools: A qualitative approach. Sci.

[24] Julia Rubin and Marsha Chechik. From products to product lines using model match-


Appendix A

Unauthorized Database Access Conversations

The following conversations were flagged as potential security violations in the ForcedAnon behavioural model using our NiCad cross-clone detection on the Anon, Admin and ForcedAnon behavioural models of user interaction with phpBB 2.0. Each conversation is associated with an unauthorized access to a SQL database resource.

Figure A.1: Action Id 152
Figure A.2: Action Id 157

1: Select(modcp,forum_id = $forum_id)

1.1: Result(modcp,forum_name, forum_topics)

---

Figure A.3: Action Id 226

1: Select(posting,forum_id = $forum_id)

1.1: Result(posting,*)

---

Figure A.4: Action Id 227

1: Select(posting,t.topic_id = $topic_id and f.forum_id = t.forum_id)

1.1: Result(posting,f.*, t.topic_status, t.topic_title, t.topic_type)
Figure A.5: Action Id 228

Figure A.6: Action Id 329
Figure A.7: Action Id 336

Figure A.8: Action Id 341
Figure A.9: Action Id 393

Figure A.10: Action Id 389
Figure A.11: Action Id 390

Figure A.12: Action Id 868