Noname manuscript No.
(will be inserted by the editor)

An Approach to Clone Detection in Sequence Diagrams and Its Application to Security Analysis

Manar H. Alalfi · Elizabeth P. Antony · James R. Cordy

Received: date / Accepted: date

Abstract Duplication in software systems is an important issue in software quality assurance. While many methods for software clone detection in source code and structural models have been described in the literature, little has been done on similarity in the dynamic behaviour of interactive systems. In this paper we present an approach to identifying near-miss interaction clones in reverse-engineered UML sequence diagrams. 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 existing robust near-miss code clone technology, our approach is text-based, working on the level of XMI, the standard interchange serialization for UML. Clone detection in UML behavioural models, such as sequence diagrams, presents a number of challenges - first, it is not clear how to break a continuous stream of interaction between lifelines (representing 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 related elements in the model file remotely. In this work we use a set of contextualizing source transformations on the XMI text representation to localize related elements, exposing the hidden hierarchical structure of the model and allowing us to granularize behavioural interactions into conversational units. Then we adapt NICAD, a robust near-miss code clone detection tool, to help us identify conversational clones in reverse-engineered behavioural models. These conversational clones are then analyzed to find worrisome interactions that may indicate security access violations.

Keywords Model Clone detection, Model based security analysis

M. H. Alalfi, E. P. Antony, J. R. Cordy
School of Computing, Queen's University, Kingston, Ontario, Canada
E-mail: {alalfi, antony, cordy}@cs.queensu.ca

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 the 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 our previous work [2, 3], 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 [32] 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. 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.

This paper is an extended version of an early results short paper presented at the International Working Conference on Reverse Engineering, WCRE 2013 [7], in which we proposed an automated approach to analyzing UML sequence diagrams to identify repeated patterns of similar interactions using the NICAD near-miss code clone detector [10]. 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 repeated similar behaviours with the aim of model re-factoring, or to identify instances of similar conversations so that bug fixes, updates and changes can be applied consistently, and thus enhancing the quality of the resulting software systems. In this paper, we leveraged cross-clone detection (identifying similar behaviours across different models) in a case study using clone detection to find worrisome conversational patterns that may indicate security access violations.

Code clone detection has been used to identify malware in software systems, as in the work of Karademir et al. to find embedded Javascript malware in Acrobat files [14] , and Farhadi at al.'s system to find malware assembly code clones in disassembled application binaries [12] . However, as code obfuscation methods become more and more sophisticated, it is increasingly difficult to detect security issues in mobile and other interactive systems using code analysis techniques alone.

On the other hand, however the code is obfuscated, in order to achieve their goals the behaviour of these malware variants necessarily remains similar, and thus by studying similarity of behaviour we can uncover threats that may not be able to be detected using code analysis. More generally, similarity of behaviour can often expose the relationship between different code implementations of any process aimed at achieving the same or similar results. Thus the ability to detect similar behaviour patterns, rather than simply similar code patterns, is increasingly important.

This paper makes the following three contributions:

1. A detailed description of a new approach for identifying near miss clones in behavioural models with a focus on UML sequence diagrams (SDs). Our method is is the first scalable approach to identifying model clones in large reverse-engineered sequence diagrams.
2. A precise definition of model clones in sequence diagrams, based on the concept of encapsulated interaction sequences (“conversations”).
3. An automated process for the identification and encapsulation of interaction sequences as self-contained conversational units, which are then used as the units of comparison for model clone detection. We evaluate our clone detection approach on a number of reverse engineered SD models of various sizes.
4. A case study demonstrating the utility of behavioural clone detection, by using cross-clone detection between reverse-engineered SD models representing interaction sequences of different user roles o identify potential security access violations in an open source web application (phpBB).

The rest of this paper is organized as follows. We begin with background information in Section 2, where we introduce the elements of basic UML sequence diagram models in the XML-based metadata interchange format (XMI) representation. We also introduce terminology that is used throughout the paper. In Section 3 we introduce our approach to identifying conversational clones in basic SD models. Using a running example, we provide details of the identification, contextualization and extraction process that yields self-contained conversational units, and motivate the need for normalization to remove irrelevant differences before comparison. In Section 4 we discuss the results of clone detection on the extracted conversations of reverse-engineered SD models, along with a brief analysis of the results. Then, we presents an example of the application of SD model cross-clone detection to identifying security access violations in reverse-engineered SDs of web applications run in various roles. Finally, Section 6 concludes and outlines opportunities for future work.

2 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.

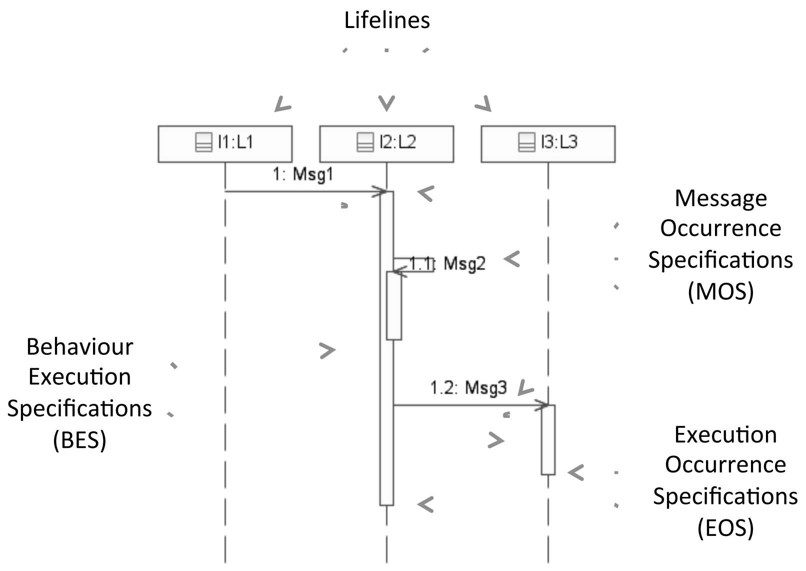

Lifelines
I1:L1
I2:L2
I3:L3
1: Msg1
1.1: Msg2
1.2: Msg3
Message Occurrence Specifications (MOS)
Behaviour Execution Specifications (BES)
Execution Occurrence Specifications (EOS)

Fig. 1 Elements of a Basic Sequence Diagram

Figure 1 shows an example highlighting the main elements in a basic sequence diagram. 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 patterns of these diagrams. Figure 2 shows an example of an SD model clone, in this case a repeated conversation between two lifelines.

2.1 Clones in Sequence Diagrams

We define clones in SDs to mean repeated patterns of similar or identical interaction elements that form complete conversations. A conversation is defined as a sequence of message interactions between two or more lifelines over a specific period of time (i.e., in the span of a BES). In this paper, we are primarily interested in identifying repeated conversations, and we define SD clones from this perspective. Code clones [22] and model clones [6] have been classified into types 1, 2, and 3, according to the level of similarity they exhibit. For SD clones, we extend these definitions as follows:

1. *Type 1 (exact)*: Conversations with identical interaction elements except for variations in visual presentation, layout and formatting. For instance, sequence diagram "message" elements with identical "name", "receiveEvent", "sendEvent" and "messageSort" attributes, but possibly different presentation fonts, sizes, physical coordinates, or colors.
2. *Type 2 (renamed)*: Conversations that may differ in the names of elements attributes values , as well as variations in visual presentation, layout and format-

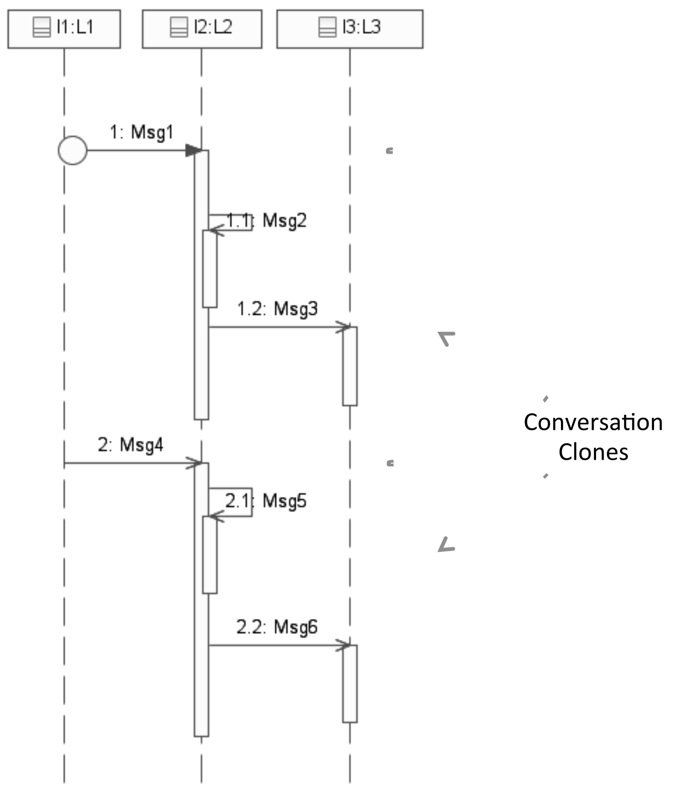


Fig. 2 Example of SD conversation clones

ting. For instance, two lifeline elements covered by a similar set of conversation messages but possibly different "name" or "xmi:id" attributes as well as different presentation fonts, sizes, physical coordinates, or colors. Such lifelines are considered Type2 clones as long as those lifelines are covered by similar set of messages in a specific conversation.

3. *Type 3 (near miss)*: Conversations that have small differences such as additions, deletions, or modifications of interaction elements, in addition to differences in the names of elements attribute values, and variations in visual presentation, layout and formatting. For instance, two conversations are considered Type 3 clones if the order or number of messages in the conversation is slightly different, in addition to the differences allowed by Types 1 and 2. The amout of difference allowed can be varied according to a configurable threshold.

3 Approach to Clone Detection

Behavioural model clone detection presents a number of challenges - first, it is not clear how to break a continuous stream of interaction between lifelines (represent-

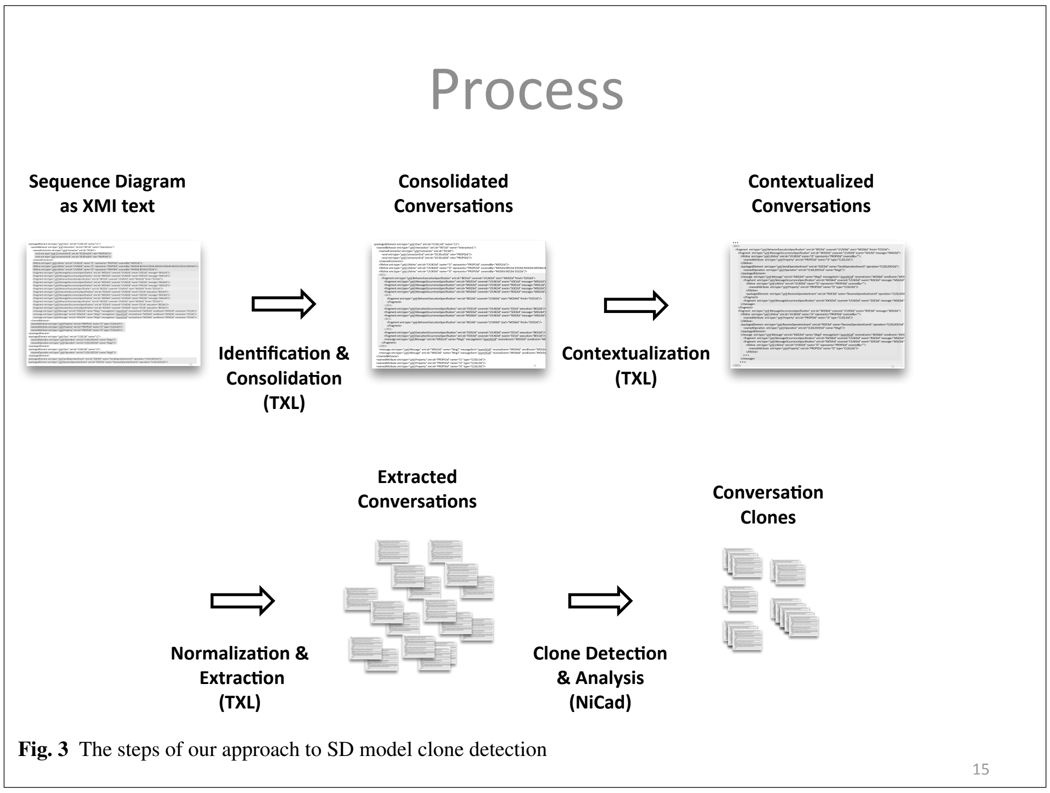

Process
Sequence Diagram as XMI text
Identification & Consolidation (TXL)
Consolidated Conversations
Contextualization (TXL)
Contextualized Conversations
Normalization & Extraction (TXL)
Extracted Conversations
Clone Detection & Analysis (NiCad)
Conversation Clones
15

Fig. 3 The steps of our approach to SD model clone detection

ing 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 related elements in the model file remotely. In this work we use a set of contextualizing source transformations on the XMI text representation to localize related elements, exposing the hidden hierarchical structure of the model and allowing us to granularize behavioural interactions into conversational units. Then we adapt NICAD, a robust near-miss code clone detection tool, to help us identify conversational clones in reverse-engineered behavioural models. These conversational clones are then analyzed to find worrisome interactions that may indicate security access violations.

To address the above challenges, our approach to SD model clone detection consists of four main stages (Figure 3). In the first stage, Identification and Consolidation, sequences of behavioural interactions in the XMI sequence diagram serialization are identified and consolidated into conversations, revealing the hierarchical conversation structure of the model in the textual SD representation.

In the second stage, Contextualization, the consolidated conversations are made independent of their context, by replacing XMI references to other parts of the model by inlining of the parts referred to. Following this transformation, the consolidated conversations in the XMI textual representation are self-contained, including all of the interaction elements that form the conversation.

In the third stage, Normalization and Extraction, the self-contained XMI representations of the conversations are extracted for comparison, normalized to remove irrelevant formatting and layout elements, and renamed to remove irrelevant naming differences in the XMI textual representation to make the process of clone identification more accurate.

```
<packagedElement xmi:type="uml:Collaboration" xmi:id="_fhwvcGGTEeO5r4_cb_qIFw" name="Collaboration1">
   <ownedBehavior xmi:type="uml:Interaction" xmi:id="INT1Id" name="Interaction1">
     <ownedConnector xmi:type="uml:Connector" xmi:id="OC1Id">
       <end xmi:type="uml:ConnectorEnd" xmi:id="OCE1Id" role="PROPL2Id"/>
       <end xmi:type="uml:ConnectorEnd" xmi:id="OCE2Id" role="PROPL1Id"/>
     </ownedConnector>
     < lifeline  xmi:type="uml:Lifeline"  xmi:id="L1Id" name="l1" represents="PROPL1Id" coveredBy="MOS1Id MOS4Id"/>
     < lifeline  xmi:type="uml:Lifeline"  xmi:id="L2Id" name="l2" represents="PROPL2Id"
            coveredBy="MOS2Id BES1Id MOS3Id"/>
     <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS1Id" covered="L1Id" event="SOE1Id"
            message="Msg1Id"/>
     <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS2Id" covered="L2Id" event="ROE1Id"
            message="Msg1Id"/>
     <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BES1Id" covered="L2Id" start="MOS2Id"
            finish="MOS3Id"/>
     <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS3Id" covered="L2Id" event="SOE1Id"
            message="Msg1ReplyId"/>
     <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS4Id" covered="L1Id" event="ROE1Id"
            message="Msg1ReplyId"/>
     <message xmi:type="uml:Message" xmi:id="Msg1Id" name="Msg1" receiveEvent="MOS2Id" sendEvent="MOS1Id"
            connector="OC1Id"/>
     <message xmi:type="uml:Message" xmi:id="Msg1ReplyId" name="Msg1" messageSort="reply" receiveEvent="MOS4Id"
            sendEvent="MOS3Id" connector="OC1Id"/>
   </ownedBehavior>
   <ownedAttribute xmi:type="uml:Property" xmi:id="PROPL1Id" name="l1" type="CLSSL1Id"/>
   <ownedAttribute xmi:type="uml:Property" xmi:id="PROPL2Id" name="l2" type="CLSSL12d"/>
 </packagedElement>
 <packagedElement xmi:type="uml:Class" xmi:id="CLSSL1Id" name="L1"/>
 <packagedElement xmi:type="uml:Class" xmi:id="CLSSL12d" name="L2">
   <ownedOperation xmi:type="uml:Operation" xmi:id="OOCLSS2Id" name="Msg1"/>
 </packagedElement>
 <packagedElement xmi:type="uml:SendOperationEvent" xmi:id="SOE1Id" name="SendOperationEvent1"
        operation="OOCLSS2Id"/>
 <packagedElement xmi:type="uml:ReceiveOperationEvent" xmi:id="ROE1Id" name="ReceiveOperationEvent1"
        operation="OOCLSS2Id"/>
```

Fig. 4 An example showing the various elements of the XMI representation of a SD
Conversations are represented by BehaviorExecutionSpecification elements (green), associated with lifelines (purple), and consisting of messages, events and operations (red). Relationships between elements in the XMI textual representation are represented both implicitly, by sequential adjacency, and explicitly, using attributes referencing other elements.

The first three stages use TXL [9] source transformations to transform the textual SD representation and extract self-contained conversational units. The final stage, Clone Detection and Analysis, uses NICAD [10], a standard code clone detector, to automatically identify cloned conversations in the large set of contextualized and normalized conversational units. In the following subsections, we elaborate each of these stages in more detail.

3.1 Identification and Consolidation

The flat structure of the XMI sequence diagram representation (Figure 4), offers little locality - fragments and elements of conversations are spread across the XMI text, using attributes and textual ordering to reference and implicitly group related elements.

Behaviour Execution Specifications (BESs), for example, (e.g., green highlight in Figure 4), reference the lifeline they are part of using the *covered* attribute, and to the sequence of messages and events comprising their associated conversation using the *start* and *finish* attributes. These attributes reference the first and last elements of the sequence that forms the conversation, implicitly including the elements textually between them, and these included elements in turn refer to their parts using attributes in similar fashion.

In order to make this scattered representation of conversations amenable to comparison, we need to recursively gather the referenced and related elements of BES conversations together and organize them into an explicit hierarchical representation of the interaction structures they represent. This restructuring or the transformation process consists of two main steps: identifying and consolidating conversational units into a hierarchy, and then contextualizing these units to be independent of their surroundings, as shown in Figure 3.

3.1.1 Identification: 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. NICAD has previously been used in detecting clones in programming languages such as C, C#, Java, Python and other programming languages, and more recently has been extended and specialized to finding model clones, as part of the Simone model clone detector [6].

NICAD is designed to find code clones of a given granularity, such as functions, blocks, or statements. It begins by enumerating all of the instances of the desired units in the code, and then comparing them pairwise for near-miss similarity within a defined difference threshold. In Simone, the units of model comparison are Simulink subsystems, which compare roughly to functions or classes in traditional programming languages.

Unlike Simulink models, the XMI serialization of UML sequence diagrams does not have an explicit nested structure. Rather, it is a flat sequence of individual elements linked by attributes as described above. Thus one of the main challenges in using NICAD to analyze sequence diagrams for clones is understanding how to reverse engineer the hidden nested structural representation of interaction conversations from the flat representation of the original XMI SD serialization.

The second major challenge is simply the identification of an appropriate level of granularity for comparison. In SD conversations, individual messages are very small, and would yield a huge number of clones that would not be useful or relevant for most applications. On the other hand, comparing the interactions of entire lifelines would likely reveal very few clones, and would miss clones of many interesting shorter interactions. Thus we decided to break lifelines into grouped sequences of interactions with other lifelines, based on the SD Behavioural Execution Specification (BES) elements of the lifeline. We call these grouped sequences *conversations*, since they encapsulate complete sequences of related interactions initiated by one lifeline with others.

Figure 5 shows an example interaction, with Message Occurrence Specification (MOS) and Behavioral Execution Specification (BES) elements labelled with the XMI Ids of their XMI textual representation. The XMI Id is a unique identifier assigned to each element of the SD in its XMI textual form. (In the Figure, XMI Ids have been renamed to simpler identifiers to aid understanding of the example.) In the example of Figure 5, there are three labelled BES fragments, *BES1Id, BES2Id*

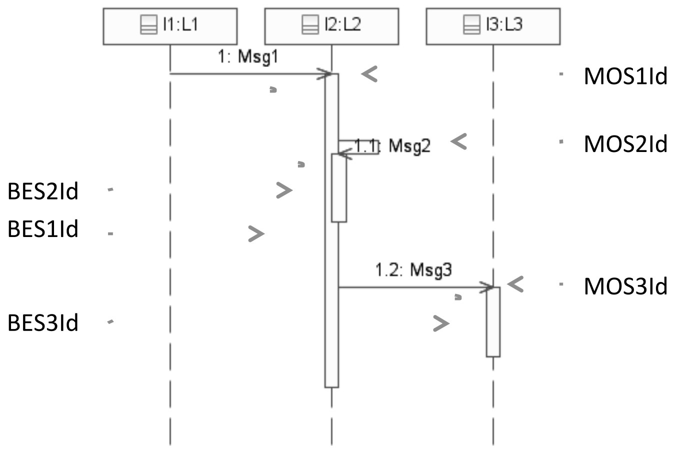


Fig. 5 An example SD fragment, showing the BES and MOS elements with their corresponding Ids from the XMI representation

and *BES3Id*, corresponding to the three BES elements of the XMI representation, shown in Figure 6. In the XMI representation, the *start* and *finish* attributes of the BES elements indicate by reference the beginning and ending elements of the BES's conversation in the flat XMI representation, and the *covered* attribute identifies the corresponding lifelines. It should also be noted that conversations are often nested - in the example of Figure 5, the conversations identified by BES2Id and BES3Id are part of the main conversation identified by BES1Id.

3.1.2 Consolidation - Creating a conversational unit

For each BES element identified in the XMI representation, we create a conversation container unit identified by <*BES*>...</*BES*> tags. The consolidation step draws the messages and execution occurrences that are part of the each BES into the corresponding BES conversation units. That is, we gather and nest all of the conversational elements of the BES into the container. The *start* and *finish* attributes of each BES specify the elements in the flat representation that begin and end the BES's conversation.

Because message, behaviour and event occurrences have a general sequential ordering in the XMI representation, this step primarily involves moving the elements adjacent to the BES element inside the new <*BES* >...</*BES* > container. Elements immediately before the BES element, beginning with the one referenced by its *start* element, represent the message(s) that initiate the conversation. Elements following the BES element, beginning with the one immediately adjacent and ending with the one specified by its *finish* attribute, represent the messages, executions and subconversations that are part of the conversation. Recursively consolidating BES conversation units yields an explicit hierarchy of conversations such as the one shown in Figure 8.

```
 . . .
<packagedElement xmi:type="uml:Collaboration" xmi:id="Collaboration1Id" name="Collaboration1">
  <ownedBehavior xmi:type="uml:Interaction" xmi:id="INT1Id" name="Interaction1">
    <ownedConnector xmi:type="uml:Connector" xmi:id="OC3Id">
      <end xmi:type="uml:ConnectorEnd" xmi:id="OC3End1Id" role="PROPl2Id"/>
      <end xmi:type="uml:ConnectorEnd" xmi:id="OC3End2Id" role="PROPl3Id"/>
    </ownedConnector>
    < lifeline   xmi:type="uml:Lifeline"  xmi:id="LFLNl1Id" name="l1" represents="PROPl1Id" coveredBy="MOS1Id"/>
    < lifeline   xmi:type="uml:Lifeline"  xmi:id="LFLNl2Id" name="l2" represents="PROPl2Id" coveredBy="MOS2Id BES1Id
             EOS3Id MOS3Id MOS4Id BES2Id EOS1Id MOS5Id"/>
    < lifeline   xmi:type="uml:Lifeline"  xmi:id="LFLNl3Id" name="l3" represents="PROPl3Id" coveredBy="MOS6Id BES3Id
             EOS2Id"/>
    <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS1Id" covered="LFLNl1Id" event="SOE1Id"
             message="MSG1Id"/>
    <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS2Id" covered="LFLNl2Id" event="ROE1Id"
             message="MSG1Id"/>
    <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BES1Id" covered="LFLNl2Id" start= "MOS2Id"
             finish="EOS3Id"/>
    <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS3Id" covered="LFLNl2Id" event="SOE2Id"
             message="MSG2Id"/>
    <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS4Id" covered="LFLNl2Id" event="ROE2Id"
             message="MSG2Id"/>
    <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BES2Id" covered="LFLNl2Id" start="MOS4Id"
             finish="EOS1Id"/>
    <fragment xmi:type="uml:ExecutionOccurrenceSpecification" xmi:id="EOS1Id" covered="LFLNl2Id" event="EE1Id"
             execution="BES2Id"/>
    <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS5Id" covered="LFLNl2Id" event="SOE3Id"
             message="MSG3Id"/>
    <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS6Id" covered="LFLNl3Id" event="ROE3Id"
             message="MSG3Id"/>
    <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BES3Id" covered="LFLNl3Id" start="MOS6Id"
             finish="EOS2Id"/>
    <fragment xmi:type="uml:ExecutionOccurrenceSpecification" xmi:id="EOS2Id" covered="LFLNl3Id" event="EE1Id"
             execution="BES3Id"/>
    <fragment xmi:type="uml:ExecutionOccurrenceSpecification" xmi:id="EOS3Id" covered="LFLNl2Id" event="EE1Id"
             execution="BES1Id"/>
    <message xmi:type="uml:Message" xmi:id="MSG1Id" name="Msg1" messageSort="asynchCall" receiveEvent="MOS2Id"
             sendEvent="MOS1Id" connector="OC1Id"/>
    <message xmi:type="uml:Message" xmi:id="MSG2Id" name="Msg2" messageSort="asynchCall" receiveEvent="MOS4Id"
             sendEvent="MOS3Id" connector="OC2Id"/>
    <message xmi:type="uml:Message" xmi:id="MSG3Id" name="Msg3" messageSort="asynchCall" receiveEvent="MOS6Id"
             sendEvent="MOS5Id" connector="OC3Id"/>
  </ownedBehavior>
  <ownedAttribute xmi:type="uml:Property" xmi:id="PROPl2Id" name="l2" type="CLSSL2Id"/>
  <ownedAttribute xmi:type="uml:Property" xmi:id="PROPl3Id" name="l3" type="CLSSL3Id"/>
</packagedElement>
 . . .
```

Fig. 6 Step I: Identification of BES fragments in the XMI representation

Identified BES fragments are shown in green. The beginning and ending elements referred to by the first BES conversation are shown in red, and its invoking MOS is shown in purple.

Identification, consolidation and contextualization are implemented as structural transformations of the XMI textual representation using the TXL [9] source transformation system. Beginning with a general grammar for parsing the XMI representation of UML sequence diagrams, a set of structural transformation rules is created to identify, consolidate and contextualize BES specifications into self-contained hierarchical units. Figure 7 shows an example TXL rule to consolidate the conversational elements of each BES up to the element identified by its *finish* attribute's value, *FinishId*, in the XMI serialization. The result of consolidating the three example BES conversations identified in Figure 6 is shown in Figure 8.

```
rule restructBES
    replace [xmi_element*]
        <fragment 'xmi:type="uml:BehaviorExecutionSpecification" 'xmi:id=BESId [attvalue] 'covered=LifelineId [attvalue]
                ' start=StartId [attvalue] ' finish=FinishId [attvalue] />
        MoreElements [xmi_element*]

    deconstruct * MoreElements
        < Element [id] 'xmi:type=Type [attvalue] 'xmi:id=FinishId Attributes [tag_attribute *] />
        RemainingElements [xmi_element*]

    construct FinishFrag [xmi_element]
        < Element 'xmi:type=Type 'xmi:id=FinishId Attributes />

    construct MessagesOfBES [xmi_element*]
        _ [addElements FinishId MoreElements]

    by
        <BES 'start=StartId ' finish=FinishId>
            <fragment 'xmi:type="uml:BehaviorExecutionSpecification" 'xmi:id=BESId 'covered= LifelineId ' start= StartId
                    ' finish= FinishId >
                MessagesOfBES [. FinishFrag]
            </fragment>
        </BES>
        RemainingElements
end rule
```

Fig. 7 TXL rule to consolidate BES conversations

3.1.3 Contextualization - Making units whole

As shown in Figure 8, 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 elements use XML attributes to refer to other elements such as messages, types and lifelines that describe their meaning.

Figure 9 illustrates how the attributes of each MOS and the EOS in consolidated BES conversations refer to other elements in the XMI representation. The attributes of these elements in turn reference other elements, and so on, as illustrated in Figure 10.

Contextualization draws in the elements of the context that are referenced by the BES elements that are directly part of the conversation. That is, it brings all the elements involved – the lifelines, properties, events and messages – into the contextualized BES unit. The result is a complete self-contained description of each conversation, independent of its surroundings.

To contextualize a consolidated BES, each Message Occurrence Specification (MOS) and Event Occurrence Specification (EOS) fragment in the BES is converted to a container tag, and the elements referred to by the attributes of the fragment are inlined 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 10. These attributes represent the lifeline, event and message of the Message Occurrence Specification.

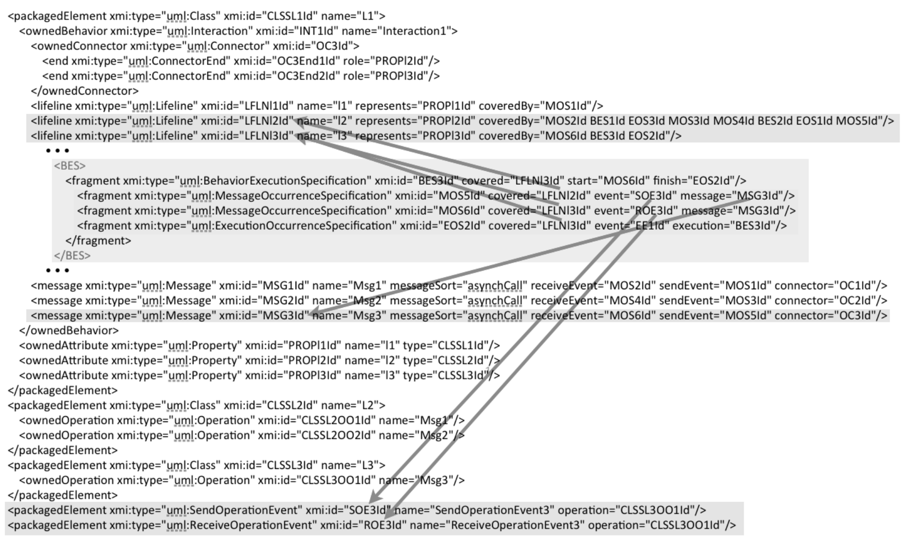
Contextualization proceeds recursively. Thus to inline the *covered* attribute of the second MOS in the third embedded BES example of Figure 9, the <*lifeline*> element with id *LFNI2Id*, referred to by the attribute, is located and copied into the container tag of the MOS. From the inlined <*lifeline*> element, the <*ownedAttribute*> element

```
 . . .
<packagedElement xmi:type="uml:Collaboration" xmi:id="Collaboration1Id" name="Collaboration1">
  <ownedBehavior xmi:type="uml:Interaction" xmi:id="INT1Id" name="Interaction1">
    <ownedConnector xmi:type="uml:Connector" xmi:id="OC3Id">
      <end xmi:type="uml:ConnectorEnd" xmi:id="OC3End1Id" role="PROPl2Id"/>
      <end xmi:type="uml:ConnectorEnd" xmi:id="OC3End2Id" role="PROPl3Id"/>
    </ownedConnector>
    < lifeline  xmi:type="uml:Lifeline" xmi:id="LFLNl1Id" name="l1" represents="PROPl1Id" coveredBy="MOS1Id"/>
    < lifeline  xmi:type="uml:Lifeline" xmi:id="LFLNl2Id" name="l2" represents="PROPl2Id" coveredBy="MOS2Id BES1Id
            EOS3Id MOS3Id MOS4Id BES2Id EOS1Id MOS5Id"/>
    < lifeline  xmi:type="uml:Lifeline" xmi:id="LFLNl3Id" name="l3" represents="PROPl3Id" coveredBy="MOS6Id BES3Id
            EOS2Id"/>
    <BES>
      <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BES1Id" covered="LFLNl2Id"
              start="MOS2Id' finish="EOS3Id"/>
        <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS1Id" covered="LFLNl1Id" event="SOE1Id"
                message="MSG1Id"/>
        <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS2Id" covered="LFLNl2Id" event="ROE1Id"
                message="MSG1Id"/>
        <BES>
          <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BES2Id" covered="LFLNl2Id" start="MOS4Id"
                  finish="EOS1Id">
            <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS3Id" covered="LFLNl2Id" event="SOE2Id
                    " message="MSG2Id"/>
            <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS4Id" covered="LFLNl2Id" event="ROE2Id
                    " message="MSG2Id"/>
            <fragment xmi:type="uml:ExecutionOccurrenceSpecification" xmi:id="EOS1Id" covered="LFLNl2Id" event="EE1Id"
                    execution="BES2Id"/>
          </fragment>
        </BES>
        <BES>
          <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BES3Id" covered="LFLNl3Id" start="MOS6Id"
                  finish="EOS2Id"/>
            <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS5Id" covered="LFLNl2Id" event="SOE3Id
                    " message="MSG3Id"/>
            <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS6Id" covered="LFLNl3Id" event="ROE3Id
                    " message="MSG3Id"/>
            <fragment xmi:type="uml:ExecutionOccurrenceSpecification" xmi:id="EOS2Id" covered="LFLNl3Id" event="EE1Id"
                    execution="BES3Id"/>
          </fragment>
        </BES>
        <fragment xmi:type="uml:ExecutionOccurrenceSpecification" xmi:id="EOS3Id" covered="LFLNl2Id" event="EE1Id"
                execution="BES1Id"/>
      </fragment>
    </BES>
    <message xmi:type="uml:Message" xmi:id="MSG1Id" name="Msg1" messageSort="asynchCall" receiveEvent="MOS2Id"
            sendEvent="MOS1Id" connector="OC1Id"/>
    <message xmi:type="uml:Message" xmi:id="MSG2Id" name="Msg2" messageSort="asynchCall" receiveEvent="MOS4Id"
            sendEvent="MOS3Id" connector="OC2Id"/>
    <message xmi:type="uml:Message" xmi:id="MSG3Id" name="Msg3" messageSort="asynchCall" receiveEvent="MOS6Id"
            sendEvent="MOS5Id" connector="OC3Id"/>
  </ownedBehavior>
  <ownedAttribute xmi:type="uml:Property" xmi:id="PROPl1Id" name="l1" type="CLSSL1Id"/>
  <ownedAttribute xmi:type="uml:Property" xmi:id="PROPl2Id" name="l2" type="CLSSL2Id"/>
  <ownedAttribute xmi:type="uml:Property" xmi:id="PROPl3Id" name="l3" type="CLSSL3Id"/>
</packagedElement>
  . . .
```

Fig. 8 Step 2: Consolidation of BES elements

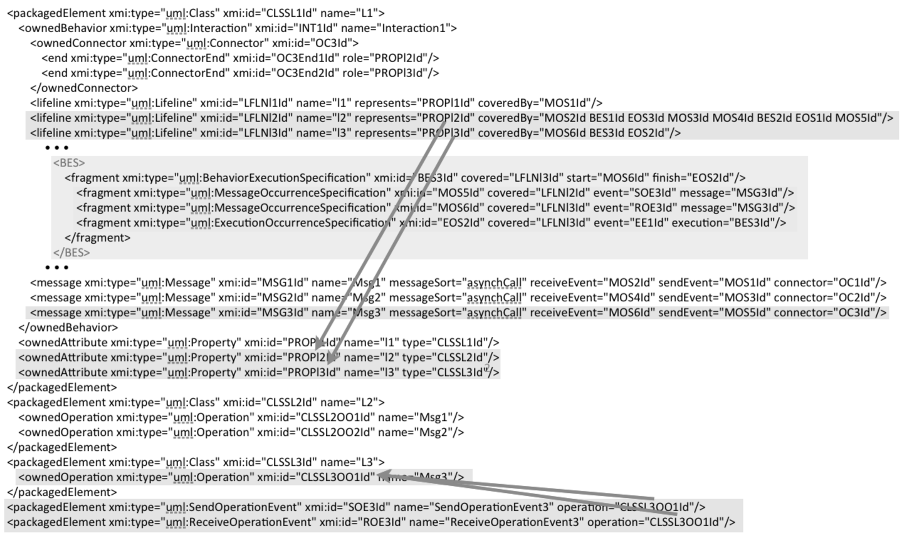
The sequence of messages and other elements comprising each BES conversation have been consolidated into new BES conversation elements. For example, all the elements between the starting and ending elements (red) of the first BES conversation (BES1Id), as well as its invoking MOS element (purple) have been moved inside its new BES wrapper, and similarly for the embedded BES elements BES2Id and BES3Id, recursively.

with *xmi:id="PROPl2Id"*, referred to by the lifeline's *represents* attribute, is then inlined to include the property/object of the class that the lifeline covers. Similarly, the elements referenced by the *event* attribute are inlined recursively until there are no more elements left.



```
<packagedElement xmi:type="uml:Class" xmi:id="CLSSL1Id" name="L1">
  <ownedBehavior xmi:type="uml:Interaction" xmi:id="INT1Id" name="Interaction1">
    <ownedConnector xmi:type="uml:Connector" xmi:id="OC3Id">
      <end xmi:type="uml:ConnectorEnd" xmi:id="OC3End1Id" role="PROPl2Id"/>
      <end xmi:type="uml:ConnectorEnd" xmi:id="OC3End2Id" role="PROPl3Id"/>
    </ownedConnector>
    <lifeline xmi:type="uml:Lifeline" xmi:id="LFLNl1Id" name="l1" represents="PROPl1Id" coveredBy="MOS1Id"/>
    <lifeline xmi:type="uml:Lifeline" xmi:id="LFLNl2Id" name="l2" represents="PROPl2Id" coveredBy="MOS2Id BES1Id EOS3Id MOS3Id MOS4Id BES2Id EOS1Id MOS5Id"/>
    <lifeline xmi:type="uml:Lifeline" xmi:id="LFLNl3Id" name="l3" represents="PROPl3Id" coveredBy="MOS6Id BES3Id EOS2Id"/>
      • • •
      <BES>
        <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BES3Id" covered="LFLNl3Id" start="MOS6Id" finish="EOS2Id"/>
          <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS5Id" covered="LFLNl2Id" event="SOE3Id" message="MSG3Id"/>
          <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS6Id" covered="LFLNl3Id" event="ROE3Id" message="MSG3Id"/>
          <fragment xmi:type="uml:ExecutionOccurrenceSpecification" xmi:id="EOS2Id" covered="LFLNl3Id" event="EE1Id" execution="BES3Id"/>
        </fragment>
      </BES>
      • • •
    <message xmi:type="uml:Message" xmi:id="MSG1Id" name="Msg1" messageSort="asynchCall" receiveEvent="MOS2Id" sendEvent="MOS1Id" connector="OC1Id"/>
    <message xmi:type="uml:Message" xmi:id="MSG2Id" name="Msg2" messageSort="asynchCall" receiveEvent="MOS4Id" sendEvent="MOS3Id" connector="OC2Id"/>
    <message xmi:type="uml:Message" xmi:id="MSG3Id" name="Msg3" messageSort="asynchCall" receiveEvent="MOS6Id" sendEvent="MOS5Id" connector="OC3Id"/>
  </ownedBehavior>
  <ownedAttribute xmi:type="uml:Property" xmi:id="PROPl1Id" name="l1" type="CLSSL1Id"/>
  <ownedAttribute xmi:type="uml:Property" xmi:id="PROPl2Id" name="l2" type="CLSSL2Id"/>
  <ownedAttribute xmi:type="uml:Property" xmi:id="PROPl3Id" name="l3" type="CLSSL3Id"/>
</packagedElement>
<packagedElement xmi:type="uml:Class" xmi:id="CLSSL2Id" name="L2">
  <ownedOperation xmi:type="uml:Operation" xmi:id="CLSSL2OO1Id" name="Msg1"/>
  <ownedOperation xmi:type="uml:Operation" xmi:id="CLSSL2OO2Id" name="Msg2"/>
</packagedElement>
<packagedElement xmi:type="uml:Class" xmi:id="CLSSL3Id" name="L3">
  <ownedOperation xmi:type="uml:Operation" xmi:id="CLSSL3OO1Id" name="Msg3"/>
</packagedElement>
<packagedElement xmi:type="uml:SendOperationEvent" xmi:id="SOE3Id" name="SendOperationEvent3" operation="CLSSL3OO1Id"/>
<packagedElement xmi:type="uml:ReceiveOperationEvent" xmi:id="ROE3Id" name="ReceiveOperationEvent3" operation="CLSSL3OO1Id"/>
```

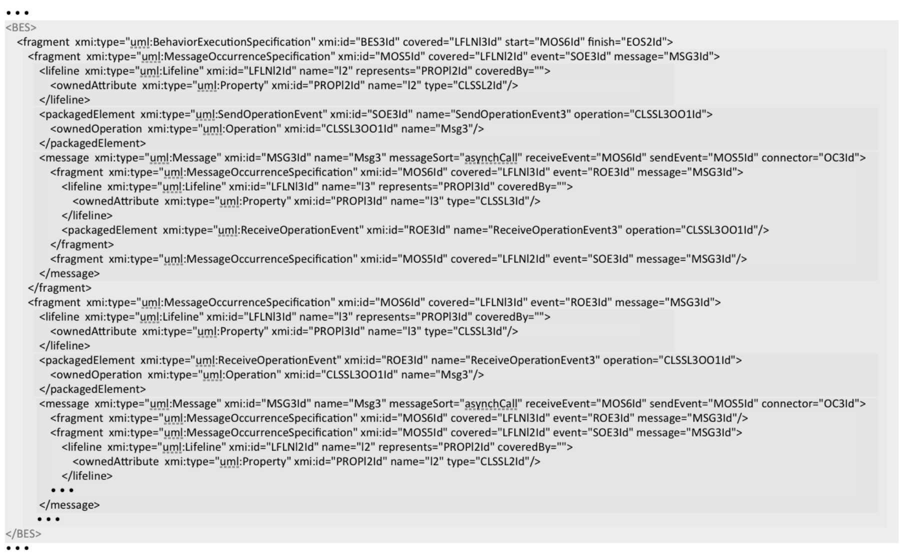
Fig. 9 Elements referenced by the attributes of elements in the XMI text of a BES (adapted from [7])



```
<packagedElement xmi:type="uml:Class" xmi:id="CLSSL1Id" name="L1">
  <ownedBehavior xmi:type="uml:Interaction" xmi:id="INT1Id" name="Interaction1">
    <ownedConnector xmi:type="uml:Connector" xmi:id="OC3Id">
      <end xmi:type="uml:ConnectorEnd" xmi:id="OC3End1Id" role="PROPl2Id"/>
      <end xmi:type="uml:ConnectorEnd" xmi:id="OC3End2Id" role="PROPl3Id"/>
    </ownedConnector>
    <lifeline xmi:type="uml:Lifeline" xmi:id="LFLNl1Id" name="l1" represents="PROPl1Id" coveredBy="MOS1Id"/>
    <lifeline xmi:type="uml:Lifeline" xmi:id="LFLNl2Id" name="l2" represents="PROPl2Id" coveredBy="MOS2Id BES1Id EOS3Id MOS3Id MOS4Id BES2Id EOS1Id MOS5Id"/>
    <lifeline xmi:type="uml:Lifeline" xmi:id="LFLNl3Id" name="l3" represents="PROPl3Id" coveredBy="MOS6Id BES3Id EOS2Id"/>
      • • •
      <BES>
        <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BES3Id" covered="LFLNl3Id" start="MOS6Id" finish="EOS2Id"/>
          <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS5Id" covered="LFLNl2Id" event="SOE3Id" message="MSG3Id"/>
          <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS6Id" covered="LFLNl3Id" event="ROE3Id" message="MSG3Id"/>
          <fragment xmi:type="uml:ExecutionOccurrenceSpecification" xmi:id="EOS2Id" covered="LFLNl3Id" event="EE1Id" execution="BES3Id"/>
        </fragment>
      </BES>
      • • •
    <message xmi:type="uml:Message" xmi:id="MSG1Id" name="Msg1" messageSort="asynchCall" receiveEvent="MOS2Id" sendEvent="MOS1Id" connector="OC1Id"/>
    <message xmi:type="uml:Message" xmi:id="MSG2Id" name="Msg2" messageSort="asynchCall" receiveEvent="MOS4Id" sendEvent="MOS3Id" connector="OC2Id"/>
    <message xmi:type="uml:Message" xmi:id="MSG3Id" name="Msg3" messageSort="asynchCall" receiveEvent="MOS6Id" sendEvent="MOS5Id" connector="OC3Id"/>
  </ownedBehavior>
  <ownedAttribute xmi:type="uml:Property" xmi:id="PROPl1Id" name="l1" type="CLSSL1Id"/>
  <ownedAttribute xmi:type="uml:Property" xmi:id="PROPl2Id" name="l2" type="CLSSL2Id"/>
  <ownedAttribute xmi:type="uml:Property" xmi:id="PROPl3Id" name="l3" type="CLSSL3Id"/>
</packagedElement>
<packagedElement xmi:type="uml:Class" xmi:id="CLSSL2Id" name="L2">
  <ownedOperation xmi:type="uml:Operation" xmi:id="CLSSL2OO1Id" name="Msg1"/>
  <ownedOperation xmi:type="uml:Operation" xmi:id="CLSSL2OO2Id" name="Msg2"/>
</packagedElement>
<packagedElement xmi:type="uml:Class" xmi:id="CLSSL3Id" name="L3">
  <ownedOperation xmi:type="uml:Operation" xmi:id="CLSSL3OO1Id" name="Msg3"/>
</packagedElement>
<packagedElement xmi:type="uml:SendOperationEvent" xmi:id="SOE3Id" name="SendOperationEvent3" operation="CLSSL3OO1Id"/>
<packagedElement xmi:type="uml:ReceiveOperationEvent" xmi:id="ROE3Id" name="ReceiveOperationEvent3" operation="CLSSL3OO1Id"/>
```

Fig. 10 Elements indirectly referenced by attributes of the elements in the XMI text of a BES (adapted from [7])

For the *event* attribute of the MOS, which specifies whether the message occurrence is a send or receive event, we inline the corresponding <*packagedElement*> with *xmi:id=“SOE3Id”* and, from its *operation* attribute, the corresponding <*ownedOperation*> element is then inlined. Finally, the *message* attribute of the MOS references the corresponding <*message*> element. The <*message*> element, in turn inlines the sending and receiving MOS of the message with the *sendEvent receiveEvent* at-



```
•••
<BES>
  <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BES3Id" covered="LFLNl3Id" start="MOS6Id" finish="EOS2Id">
    <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS5Id" covered="LFLNl2Id" event="SOE3Id" message="MSG3Id">
      <lifeline xmi:type="uml:Lifeline" xmi:id="LFLNl2Id" name="l2" represents="PROPl2Id" coveredBy="">
        <ownedAttribute xmi:type="uml:Property" xmi:id="PROPl2Id" name="l2" type="CLSSL2Id"/>
      </lifeline>
      <packagedElement xmi:type="uml:SendOperationEvent" xmi:id="SOE3Id" name="SendOperationEvent3" operation="CLSSL3OO1Id">
        <ownedOperation xmi:type="uml:Operation" xmi:id="CLSSL3OO1Id" name="Msg3"/>
      </packagedElement>
      <message xmi:type="uml:Message" xmi:id="MSG3Id" name="Msg3" messageSort="asynchCall" receiveEvent="MOS6Id" sendEvent="MOS5Id" connector="OC3Id">
        <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS6Id" covered="LFLNl3Id" event="ROE3Id" message="MSG3Id">
          <lifeline xmi:type="uml:Lifeline" xmi:id="LFLNl3Id" name="l3" represents="PROPl3Id" coveredBy="">
            <ownedAttribute xmi:type="uml:Property" xmi:id="PROPl3Id" name="l3" type="CLSSL3Id"/>
          </lifeline>
          <packagedElement xmi:type="uml:ReceiveOperationEvent" xmi:id="ROE3Id" name="ReceiveOperationEvent3" operation="CLSSL3OO1Id"/>
        </fragment>
        <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS5Id" covered="LFLNl2Id" event="SOE3Id" message="MSG3Id"/>
      </message>
    </fragment>
    <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS6Id" covered="LFLNl3Id" event="ROE3Id" message="MSG3Id">
      <lifeline xmi:type="uml:Lifeline" xmi:id="LFLNl3Id" name="l3" represents="PROPl3Id" coveredBy="">
        <ownedAttribute xmi:type="uml:Property" xmi:id="PROPl3Id" name="l3" type="CLSSL3Id"/>
      </lifeline>
      <packagedElement xmi:type="uml:ReceiveOperationEvent" xmi:id="ROE3Id" name="ReceiveOperationEvent3" operation="CLSSL3OO1Id">
        <ownedOperation xmi:type="uml:Operation" xmi:id="CLSSL3OO1Id" name="Msg3"/>
      </packagedElement>
      <message xmi:type="uml:Message" xmi:id="MSG3Id" name="Msg3" messageSort="asynchCall" receiveEvent="MOS6Id" sendEvent="MOS5Id" connector="OC3Id">
        <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS6Id" covered="LFLNl3Id" event="ROE3Id" message="MSG3Id"/>
        <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="MOS5Id" covered="LFLNl2Id" event="SOE3Id" message="MSG3Id">
          <lifeline xmi:type="uml:Lifeline" xmi:id="LFLNl2Id" name="l2" represents="PROPl2Id" coveredBy="">
            <ownedAttribute xmi:type="uml:Property" xmi:id="PROPl2Id" name="l2" type="CLSSL2Id"/>
          </lifeline>
        •••
      </message>
      •••
</BES>
•••
```

Fig. 11 A fully contextualized BES conversation unit (adapted from [7])

tributes, 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 a completely contextualized BES conversation, as shown in Figure 11.

Inlining 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 inlining each operation description element into Web Service Cells, or WSCells. Similarly to WSDL documents, the contextualization stage is necessary to consolidate all the conversation elements into self-contained units from the XMI representation of SDs. While in general all elements referenced by element attributes are inlined, in order to avoid repetition of information and unbounded recursion, some attributes must not be inlined during contextualization:

1. *type:* The *type* attribute of the <*ownedAttribute*> element is not expanded as it would inline the entire element's class. We are only interested in the name of the class and its operations in the conversation. Operations are inlined from the *operation* attribute of MOS fragments.
2. *connector:* Similarly, the connector attribute of <*message*> elements is not contextualized, because the *end* attribute of the <*ownedAttribute*> element inlined from a <*lifeline*> element already inlines it as end point of the MOS.
3. *execution:* The *execution* attribute of an Execution Occurrence Specification (EOS) refers to the BES element that it is part of. Inlining this link would simply duplicate the information.

```
<source file="SDModels/AnonSeq..." startline="3341" endline="3404">
    <BES start="34505RHttp" finish="EventOccHttp56">
        <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BehivExecHttp56" covered="0L"
                start="3450..."  finish="Event...">
            <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="34505RHttp" covered="0L"
                    event="ReciveOpHttp56" message="Mess..56">
                ...
            </fragment>
        </fragment>
    </BES>
</source>

<source file="SDModels/AnonSeq..." startline="3573" endline="3648">
    <BES start="34505R" finish="EventOcc56">
        <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BehivExec56"
                covered="phpbb_forumsphpbb_topics" start="34505R" finish="EventOcc56">
            <fragment xmi:type="uml:MessageOccurrenceSpecition" xmi:id="34505Rback" covered="0L"
                    event="Reciveback56" message="Messback56">
                ...
            </fragment>
        </fragment>
    </BES>
</source>
...
```

Fig. 12 Example potential clones extracted by NICAD
Every consolidated, contextualized BES conversation element in the XMI textual representation of the model is extracted as a potential clone to be compared. Potential clones are wrapped in <*source*> tags that track their original location in the XMI model files so that clones can be related back to their original models and BES elements.

3.2 Extraction and Clone Comparison

The NICAD clone detector works by parsing a source program and extracting all of the code fragments of a particular granularity (*potential clones*) to be compared for similarity. NICAD comes packaged with number of extractor modules for a number of standard programming languages at different granularities such as classes, functions and blocks. NICAD uses a plug-in architecture [10] which allows for easy addition of new languages and granularities by supplying a TXL [9] grammar and fragment extractor for the new language.

3.2.1 Extraction

In order to use NICAD to find SD conversation clones, we used a generic XMI element grammar to parse the consolidated and contextualized XMI textual form of the SD models, and specified our contextualized <*BES*> elements as the fragments to serve as potential clones. The XMI grammar simply defines the generic form of XMI elements. This unconstrained definition yields a rough parse of the contextualized XMI text sufficient for our purposes.

To make a NICAD extractor for contextualized BES conversation fragments, we simply specialize the generic XMI grammar to recognize <*BES*> elements specially, and tell NICAD to use *bes_fragment* as the unit of granularity for clone detection.

An example of the extracted potential clones is shown in Figure 12. Each extracted potential clone (pc) is wrapped in <*source*> tags with the original file name, starting line (line number where the text representation of the BES unit begins in the

XMI model file) and ending line (line number where the text representation of the BES unit ends) as attributes.

3.2.2 Clone Comparison

The extracted conversation potential clones are compared by NICAD line by line using an optimized longest common subsequence (LCS) algorithm up to a specified near-miss difference threshold. The threshold specifies an upper limit on the fraction of lines in potential clones that can differ in order for them to be considered near-miss clones. For example, a difference threshold of 20% would allow for up to two lines in ten to be different in conversation clones.

An example of a near-miss conversation clone pair reported by NICAD in a reverse engineered SD model is shown in Figure 13. The near-miss conversation clones in the example are identical except for the message that initiates the conversation. These clones are exact near-miss (clone Type 3) clones, that is, there are identical in every respect except for a small number of individual differences (in this case, one).

3.3 Normalization

While near-miss identical clones are interesting, they are rare in the XMI representation of SD models. This is because the XMI textual representation is constrained to use unique names for each element in the model, in order to facilitate graphical rendering in the user interface. These unique names cause multiple instances of visually identical to be very different in their XMI textual representation - different enough that their textual similarity does not fall within the small difference thresholds needed to accurately detect conversation clones.

When faced with these artificial differences in representation, there are two choices for a clone detector: either raise the difference threshold to a higher number, allowing for example 50% of XMI text lines to be different, or stay with a small difference threshold, finding very few clones. In the first case, the resulting precision will be very low, reporting many false positives, because very little similarity is required. In the second case, the resulting recall with be very low, missing many visually similar conversation clones because the XMI textual representation has different naming.

The solution to this problem is *normalization*, the removal of artificial differences from the comparison of potential clones. In this section we highlight the normalization steps used in our method and their effect on enhancing results. Our normalization has been specifically refined to improve precision from the perspective of our application of clone detection to detecting security access violations as detailed in Section 4. 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.

3.3.1 Reducing Redundancy

To address this issue, we apply two kinds of normalization: removal of redundant elements, which tend to multiply the effect of small differences, and "blind" renaming,

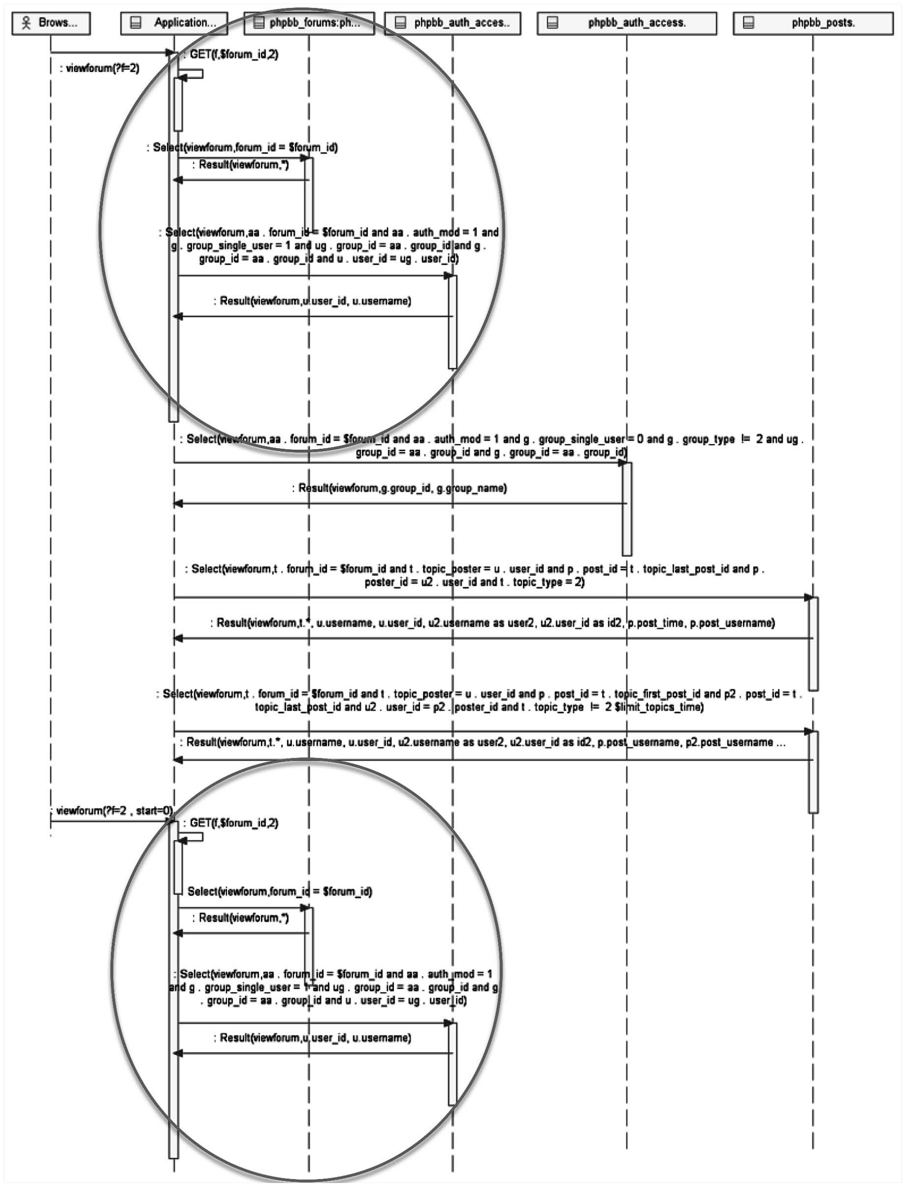

Brows...
Application...
phpbb_forums:ph...
phpbb_auth_acces..
phpbb_auth_access.
phpbb_posts.
: viewforum(?f=2)
: GET(f,$forum_id,2)
: Select(viewforum,forum_id = $forum_id)
: Result(viewforum,*)
: Select(viewforum,aa . forum_id = $forum_id and aa . auth_mod = 1 and g . group_single_user = 1 and ug . group_id = aa . group_id and g . group_id = aa . group_id and u . user_id = ug . user_id)
: Result(viewforum,u.user_id, u.username)
: Select(viewforum,aa . forum_id = $forum_id and aa . auth_mod = 1 and g . group_single_user = 0 and g . group_type != 2 and ug . group_id = aa . group_id and g . group_id = aa . group_id)
: Result(viewforum,g.group_id, g.group_name)
: Select(viewforum,t . forum_id = $forum_id and t . topic_poster = u . user_id and p . post_id = t . topic_last_post_id and p . poster_id = u2 . user_id and t . topic_type = 2)
: Result(viewforum,t.*, u.username, u.user_id, u2.username as user2, u2.user_id as id2, p.post_time, p.post_username)
: Select(viewforum,t . forum_id = $forum_id and t . topic_poster = u . user_id and p . post_id = t . topic_first_post_id and p2 . post_id = t . topic_last_post_id and u2 . user_id = p2 . poster_id and t . topic_type != 2 $limit_topics_time)
: Result(viewforum,t.*, u.username, u.user_id, u2.username as user2, u2.user_id as id2, p.post_username, p2.post_username ...
: viewforum(?f=2 , start=0)
: GET(f,$forum_id,2)
Select(viewforum,forum_id = $forum_id)
: Result(viewforum,*)
: Select(viewforum,aa . forum_id = $forum_id and aa . auth_mod = 1 and g . group_single_user = 1 and ug . group_id = aa . group_id and g . group_id = aa . group_id and u . user_id = ug . user_id)
: Result(viewforum,u.user_id, u.username)

Fig. 13 BES conversation clone pair reported by NICAD visualized in Rational Software Architect

which removes differences between element names. Both normalizations are implemented as TXL source transformation plugins to the NICAD clone detector, which allows for normalization of the extracted potential clones (in our case the contextualized BES conversations) before clone comparison. The first normalizing transformation simply removes redundant elements in the contextualized potential clones, such

```
rule blindRenameAttributes
    % Attributes that should not be blind renamed
    construct RelevantIds [id*]
        'name 'type 'role

    replace $ [ attribute ]
        XMIColon [xmi_colon] TagId [id] = TagAttr [attribute_value]

    % Guard to make sure it is not one of the relevant ones
    deconstruct not * [id] RelevantIds
        TagId

    by
        XMIColon TagId = "BR"
end rule
```

Fig. 14 TXL Rule to Blind Rename Irrelevant XMI Element Attributes

as the <*operation*> element referred to by the *event* attribute of a <*packagedElement*> with *xmi:type="ReceiveOperationEvent"*, which is always the same as the <*operation*> element referenced by the corresponding *xmi:type="SendOperationEvent"*, and thus is redundant.

3.3.2 Blind Renaming

The second normalizing transformation implements "blind" renaming of element names in the extracted contextualized potential clones. The general strategy of blind renaming is to replace all identifiers in the potential clones with the same identifier, for example "X".

The generic blind renaming algorithm that is packaged with NICAD is context-independent and does this for all identifiers in a potential clone. For this reason, it can not be used for Sequence Diagram (SD) models, because it does not distinguish between element names (XMI ids) and other identifiers, such as element types. These can only be distinguished by their context in the XMI structure, and thus it was necessary to craft a custom context-dependent blind renaming plugin for SD models.

The context-dependent blind renaming transformation for SD models is implemented in TXL using a technique called *agile parsing* [11], in which the generic XMI grammar is specialized to distinguish the forms we are interested in renaming from those that should not be renamed. For example, in our approach, the TXL grammar "overrides" is used to distinguish XMI elements whose attributes that should be blind renamed from those that should not. Within the distinguished contexts, blind renaming is applied to all XMI attribute identifiers, except those that carry relevant distinguishing information, such as the type of the XMI element, the role of the lifelines involved in an operation, or the name of the class of a message, event or operation.

Figure 14 shows the TXL source transformation rule that does the actual renaming for most contexts. All attributes except *xmi:type, xmi:role* and *xmi:name* are renamed to the identifier "BR" by the rule. In some special contexts, the *xmi:name* attribute must also be renamed. Figure 15 shows the effect of blind renaming on a small section of an extracted contextualized BES conversation.

(a) Before blind renaming

```
<source file="SDModelsDec2T10BR/AnonWthAdmnLnks/AnonWithAdminLinks.sd" startline="13198" endline="13443"
         pcid="73">
  <BES start="52045R" finish="ownedOp1252">
    <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BehivExec1252" covered="phpbb_forums..."
              start="52054R" finish="ownedOp1252">
      <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="52054S" covered="0L" event="SendOp1282"
                message="Mess1282">
        ...
        <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 ... "/>
                </ownedRule>
                <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/>
                <ownedParameter xmi:type="uml:Parameter" xmi:id="AcRt52045"'
                         name="t.∗, f .forum_id, f .forum_name, ..."/>
              </ownedOperation>
            </operationTag>
          </packagedElement>
        </eventTag>
        ...
      </fragment>
    </fragment>
  </BES>
</source>
```

(b) After blind renaming

```
<source file="SDModelsDec2T10BR/AnonWthAdmnLnks/AnonWithAdminLinks.sd" startline="13198" endline="13443"
         pcid="73">
  <BES start="BR" finish="BR">
    <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BR" covered="phpbb_forumsphpbb_posts..." start=
              "BR" finish="BR">
      <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="BR" covered="BR" event="BR" message="BR">
        ...
        <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 ..."/>
                </ownedRule>
                <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"/>
                <ownedParameter xmi:type="uml:Parameter" xmi:id= "BR" name="t.∗, f.forum_id, f.forum_name, ..."/>
              </ownedOperation>
            </operationTag>
          </packagedElement>
        </eventTag>
        ...
      </fragment>
    </fragment>
  </BES>
</source>
```

Fig. 15 Example blind renaming XMI attributes in a small section of a contextualized BES conversation

4 A Case Study in SD Clone Detection

In this section, we use an application of SD clone detection to address the following research questions and to evaluate the precision and recall of our method.

RQ1*: Is our approach capable of detecting patterns of similar conversations in SDs?*

RQ2*: To what extent does the normalization stage of our approach enhance accuracy of the results?*

RQ3*: Can the approach be useful in real world applications, and how does it perform regarding precision and recall?*

In this section, we first describe the SD models used in the experiment, and address the first research question by applying our approach to these models without any normalization. We then address the second research question by comparing the results of our SD clone detection on these models with and without normalization. Finally, we address the third research question by using our SD clone detection technique to detect access control violations that appear as patterns in a set of reverse engineered sequence diagrams from a vulnerable web application with forced browsing, and comparing the results to an existing heavyweight security analysis approach. The results of the experiment are promising, using lightweight SD clone detection to find potential security risks with 100% recall and 86% precision when compared to the previous heavyweight model-checking approach.

4.1 Clone Detection in SD Models

In this section, we discuss the results obtained from basic clone detection on the reverse engineered SD models. We first provide results without normalization, followed by a careful analysis of results after normalization. Near-miss clone detectors use a similarity threshold to specify how close two fragments need to be to be considered clones. In NICAD, this is specified using a “difference threshold", the percentage of total lines that may differ in a clone pair. Selection of an appropriate difference threshold depends on the application, and must be tuned empirically. In the next section we demonstrate, using a brief example, both the need for normalization and the selection and appropriate difference threshold for SD models.

4.1.1 Design-recovered SD Models

Our approach differs from existing techniques in its ability to handle very large, reverse-engineered sequence diagrams efficiently. The SD models we use here were obtained from previous work by Alalfi et al. [2, 3] on automatically recovering SD behavioural models from dynamic web applications, in particular from multiple execution scenarios of the popular web forum application phpBB 2.0 in different user roles. We worked with seven such recovered 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 size depending on the extent of web application coverage during visits to the forum in each user role. 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 phpBB forum. They differ in size according to the extent of coverage.

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.

Next, we have two sets of models representing execution traces of an anonymous user with forced browsing (that is, an anonymous user attempting to directly access links only intended for an administrator), ForcedAnonSD1 and ForcedAnonSD2.

Finally, the last model represents an administrator user's role, showing the interaction traces of an administrator exploring the phpBB forum (AdminSD).

Each model is preprocessed as detailed in Section 3.1.3 to a contextualized representation, and the resulting file is given the extension *.sd*. 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 both clone pairs (with and without original source), and clone classes (groups of mutually-similar clone pairs within the difference threshold). Clones grouped in classes are also reported both with and without original XMI source text, and with a class identifier ("classid") assigned to each class. For our analysis we used the NICAD clone pairs reports, both with and without source. The reader is referred to the work by Roy et al. [20] for further details on how clone pairs are created and grouped into classes by NICAD.

4.1.2 Initial Results Before Normalization

In the first research question, we ask *RQ1: Is our approach capable of detecting patterns of similar conversations in SDs?* To address this, in the first part of our experiment we worked with reverse engineered SD Models without any normalization. This resulted in a large number of clones, due to the artificial similarity of a large amount of redundant information in the XMI representation, mainly in the *coveredBy* attribute of <*lifeline*> elements and the *connector* attribute of <*message*> elements. Without normalization to anonymize the *coveredBy* attribute and eliminate the redundant elements referred to by the *connector* and *type* attributes, the size of contextualized conversational units is extremely large, and the process is very expensive in terms of memory and CPU time. But more importantly, these raw results yielded large numbers of clones of little interest and very low accuracy, so we do not show them here.

To avoid the problems with redundancy yielding artificially similar clones, we next experimented with removing redundancy in the BES conversational units as described in Section 3.3.1.

Table 1 shows the initial results using NICAD to find BES conversation clones in our seven example models using a difference threshold of 35%. The first column indicates the different reverse engineered SD models used in the experiment. The second

Table 1 Unnormalized clone detection results at a difference threshold of 35%

Model Information			Without Normlization	
Model Name	**# Lines**	**# BES**	**# Clone Pairs**	**# Clone Classes**
AnonSD1	751	30	18	7
RegSD1	5375	223	1311	14
ForcedAnon SD1	9501	142	407	27
AdminSD	469356	513	9918	44
AnonSD2	53860	314	3330	27
RegSD2	455686	954	51334	42
ForcedAnon SD2	37915	1232	75947	51

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 Behavioural Execution Specifications (BESs) extracted as conversational units in each model. The fourth column reports the number of conversational clone pairs detected in the model. NICAD reports two potential clones as a clone pair if the pair differs by at most the percentage of lines specified in the difference threshold. For example, at a difference threshold of 10%, if two conversational units of 100 lines each have at least 90 lines in common (i.e., differ by at most 10 lines), then they will be reported as clones. The fifth column reports the number of clone classes the clones are grouped into, based on the similarity of the clones as specified by the threshold value.

While we experimented with many difference thresholds, using redundancy reduction alone we were only able to expose any interesting similarities at higher thresholds, hence the use of a 35% difference threshold.

Most of the clones generated using redundancy reduction alone are meaningful. However, many of them are undesired clone pairs. For instance, we noticed that some "Select" operations of the web application were being paired with "Update" operations in the conversation clones. For example, in the ForcedAnonSD1 model, representing the Anonymous User accessing administrator links, we noted 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 (Figure 16). While this is expected at the larger threshold, it is a false positive from the perspective of finding access violations (Section 4). Thus it was deemed necessary to normalize the contextualized representation of BES conversational units before comparison, to allow us to use lower difference thresholds and more precise matching of relevant elements.

4.1.3 Results After Normalization

In the second research question, we ask *RQ2: To what extent does the normalization stage of our approach enhance accuracy of the results?* To answer this question, we compared the results of clone detection on the the reverse engineered SD models with and without normalization. The steps of our normalization are detailed in Section 3.3. In brief, normalization involves eliminating irrelevant differences by selectively "blind renaming" most attribute values, and by filtering out irrele-

Table 2 Clone detection results after normalization at a difference threshold of 10%

Model Information			Without Normalization		With Normalization	
Model Name	**# Lines**	**# BES**	**# Cl. Pairs**	**# Cl. Classes**	**# Cl. Pairs**	**# Cl. Classes**
AnonSD1	751	30	0	0	14	8
RegSD1	5375	223	0	0	1116	14
ForcedAnon SD1	9501	142	0	0	298	19
AdminSD	469356	513	0	0	9584	27
AnonSD2	53860	314	0	0	3156	24
RegSD2	455686	954	0	0	39770	29
ForcedAnon SD2	37915	1232	0	0	60455	27

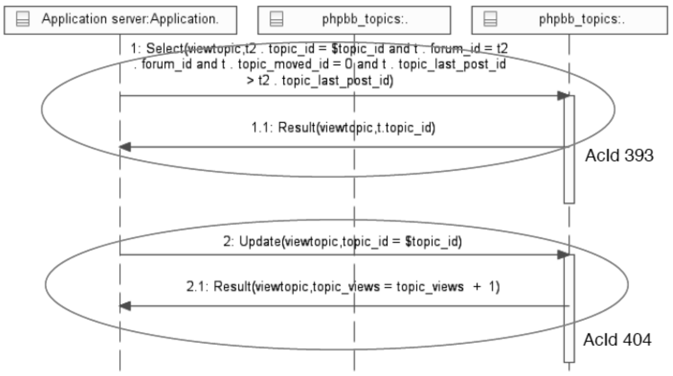

Application server:Application.
phpbb_topics:.
phpbb_topics:.
1: Select(viewtopic,t2 . topic_id = $topic_id and t . forum_id = t2 . forum_id and t . topic_moved_id = 0 and t . topic_last_post_id > t2 . topic_last_post_id)
1.1: Result(viewtopic,t.topic_id)
AcId 393
2: Update(viewtopic,topic_id = $topic_id)
2.1: Result(viewtopic,topic_views = topic_views + 1)
AcId 404

Fig. 16 An Example of a false positive.

vant elements and repeated information. For example, *operation* elements associated with *xmi:type="uml:ReceiveOperationEvent"* elements are redundant, since the same *operation* element will be associated with the corresponding *xmi:type="uml:SendOperationEvent"*. Renaming and filtering to remove redundancy and irrelevant differences allows our potential clone comparisons to be more focussed and precise.

Table 2 shows the new BES conversation clone detection results after normalization, using a difference threshold of 10%. We can also see that, without normalization, no clones are reported at 10%, due to the irrelevant differences.

Compared to the results at 35% shown in Table 1, the combination of normalization and the low difference threshold of 10% eliminated false positive clone pairs such as those shown in Figure 16, and additionally removed the large number of irrelevant very small clones reported at 35%. As expected at a low difference threshold of 10%, many fewer near miss clone pairs are reported overall, reflecting a more precise analysis.

Analysis of the results after normalization revealed that all reported clone pairs were paired based on the main action performed, and with a similarity percentage of (96 - 100)%. The action performed is represented as part of the operation elements in the contextualized representation. With these more accurate results, we were ready to apply our clone detection to finding access violations.

Table 3 Approach Performance Analysis (in CPU seconds)

Model Information		**Performance Analysis per Stage**					**Total Time**	
Model Name	**XMI Lines**	**Contex-tualize**	**Extract**	**Rename**	**Clones 10%**	**Clones 35% (-BR)**	**All stages @ 10%**	**All stages @ 35%**
AnonSD1	751	0.1	0.0	0.1	0.1	0.1	0.4	0.4
RegSD1	5375	4.9	0.4	0.6	1.0	1.3	6.9	7.2
ForcedAnon SD1	9501	4.7	0.2	0.3	0.3	0.4	5.6	5.7
AdminSD	469356	2000.6	2.5	2.7	39.2	60.7	2045.0	2066.5
AnonSD2	53860	62.3	0.6	0.7	2.5	3.7	66.2	67.3
RegSD2	455686	1492.3	1.8	2.1	23.6	35.1	1519.8	1531.3
ForcedAnon SD2	37915	76.1	0.9	1.1	5.5	7.8	83.6	85.9

4.1.4 Performance Analysis

We conducted our experiments on a 2.3 GHz Intel Core i7 Macintosh Mini with 16 Gb of memory running OSX 10.11.3 El Capitan. Our approach performed very well, analyzing large models of almost half a million lines of XMI model code in about 34 minutes. Table 3 provides detailed information about performance analysis for each stage of our approach. The actual clone analysis after the contextualization stage takes a total of less than 1.1 minutes for the largest model, which suggests that more optimization for the contextualization stage will help enhance performance of the approach.

4.2 Detecting Access Control Vulnerabilities Using Cross-Clone Detection

The third research question asks *RQ3: Can the approach be useful in real world applications, and how does it perform regarding precision and recall?* To answer this question, in the third part of our experiment, we applied our approach to the problem of detecting suspicious conversations in SDs recovered from a vulnerable web application. We compared the results using our SD clone detection approach to uncover potential security violations to the published results from a state of the art model-checking technique. In the following, we detail the problem to be addressed, the application of SD clone detection to it, and the results as compared with an existing heavyweight model-checking approach. In summary, our lightweight clone detector-based approach achieved 100% recall and 86% precision when compared to the model-checking approach.

4.2.1 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 [18]. Most web applications implement some sort of authentication or authorization mechanism to limit access to resources or functions. Such limitations are normally specified using access control policies set up by the administrator of the web application. Authentication determines a user's privileges by verifying that he/she is who he/she claims to be. The most common method of authentication is password-based, but device-based authentication (using physical cards or keys), and biometrics-based authentication is also possible.

Once a user is authenticated, access control policies determine which resources and functions of the application he/she can use based on his/her role (e.g., administrator, registered user, guest, and so on). However, there are cases where these security mechanisms can fail. Many web applications implement the access control by hiding links from the user, depending on their privilege level [25]. This kind of vulnerability is also known as *forced browsing*. This obscurity-based strategy is highly vulnerable to security breaches, because attackers may be able to simply bypass the access control mechanisms by guessing or inferring these hidden links and accessing them directly to get to unauthorized pages.

Authorization by user role is known as Role-Based Access Control (RBAC). The idea is that privileges are associated with particular roles, such as administrator, registered user or guest, and access to resources is governed by that role. When a user logs in in a particular role, they inherit the privileges associated with that role. An example of a web application using RBAC is the popular open source internet bulletin board system phpBB. Like most dynamic web applications, phpBB interacts extensively with a database back-end. Both user information and privileged, session-critical information such as roles and access permissions are stored in the database. Like many web applications, phpBB has three main user roles, administrator, registered user, and guest, and access to restricted pages is implemented in part by hiding links to such pages from the user's role. This property makes phpBB vulnerable to many forms of security attacks.

In previous work by Alalfi et al., we have reverse engineered the execution traces of various user roles of phpBB and other web applications into UML 2.1 Sequence Diagrams in XMI representation using the PHP2XMI framework [2], and then on to role-based SecureUML security models by joining them with ER models of the application database and structure [5]. The SecureUML security model is then transformed into a Prolog model, which is used to check that the model conforms to the specified access control security properties [4]. Using this process, we were able to identify a list of unauthorized SQL access actions which represent an anonymous user attempting to access the administrator's privileged links.

In this work, we use the same set of recovered SD models and use lightweight SD 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 the 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 of an administrator, and the third model (ForcedAnon) represents the execution sequences of an anonymous user attempting to follow the links that only administrators can access when navigating the same phpBB bulletin board.

In the next few sections, we explain the application of the NICAD cross-clone detector on the recovered SD models to identify the various cases of access violation

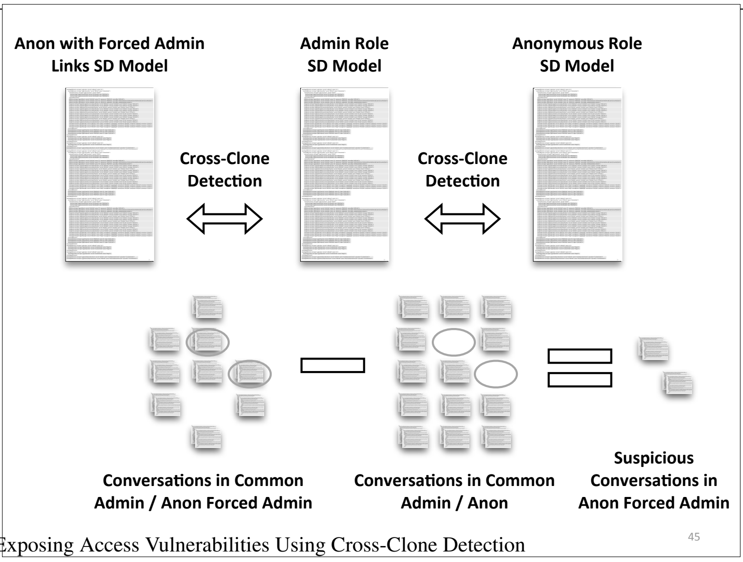

Anon with Forced Admin Links SD Model
Admin Role SD Model
Anonymous Role SD Model
Cross-Clone Detection
Cross-Clone Detection
Conversations in Common Admin / Anon Forced Admin
Conversations in Common Admin / Anon
Suspicious Conversations in Anon Forced Admin
45

Fig. 17 Exposing Access Vulnerabilities Using Cross-Clone Detection

occurring in the interaction sequences or conversations in the models. We begin with basic clone detection in SD models.

In this first application of our SD conversational clone detector, we aim at uncovering potential access control vulnerabilities in the recovered SD behavioural models of users in different roles. Our approach uses cross-cloning (i.e., conversational clones between rather than within models) to expose conversations that potentially violate access control policies. In particular, by comparing the SD model of an administrator to the SD model of an anonymous or registered user attempting to access the same links using forced browsing, we hope to expose conversations involving access to privileged information.

To implement cross-clone detection, we used the NICAD cross-clone detector [10], which runs NICAD to identify clones between rather than within systems. In our application, the cross-clone detector takes as input two recovered SD behavioural models in XMI form. Using the BES conversation extraction and normalization processes described in Section 3, the BES conversations in each model are identified, extracted and normalized. The clone detection engine then searches for only those clone pairs consisting of one BES conversation from the first model, and one from the second. The result is a list of all of the near-miss conversation clones that the two models have in common.

Figure 17 shows how we use these cross-clone results between models to expose potential access violations. Our purpose is to check whether an anonymous user can access any unauthorized content by explicitly attempting to access links that only the administrator can access while navigating the same forum (i.e., using “forced browsing”).

We first run the BES conversation cross-clone detector between the AdminUser model (the recovered model of an administrator interacting with the phpBB forum) and the ForcedAnon model (the recovered model of an anonymous user interacting with the phpBB forum, attempting to access all of the same urls as the administrator).

The result of this cross-clone detection yields a list of all of the cloned instances of administrator conversations that are in common with the anonymous user conversations using forced browsing, that is, a list of everything that the administrator can do that the anonymous user can also do by attempting to follow the same links (Figure 17, bottom left).

Next, we run a second cross-clone detection in which the same AdminUser model is compared with the Anon model (the recovered model of an anonymous user interacting with the phpBB forum in its normal way, without forced browsing). The result of this cross-clone detection is a list of all of the cloned instances of administrator conversations that are in common with a normal anonymous user conversations, that is, a list of everything the administrator can do that the normal, non-forcing anonymous user can also do (Figure 17, bottom middle).

We then take the difference in the sets of extracted clone pairs from the two cross-clone detections based on the NICAD potential clone identifiers (pcids) of the cloned conversations (Figure 17, bottom right). NICAD identifies each extracted potential clone (in our case each BES conversation) using a unique potential clone identifier, or *pcid*, which allows us to easily compare the cross-clone results. The resulting list of remaining clone pairs represents the set of BES conversations that the ForcedAnon model has in common with the AdminUser model, which are not in common with the normal Anon model. In other words, the actions that the anonymous user was able to do while pretending to be the administrator (using forced browsing) that he/she could not do normally.

Naturally, this set of remaining clone pairs indicates potential access violations that the anonymous user was able to force by trying administrator links. These clones are then analyzed to see what database actions (SQL statements) are executed in these conversations, and thus what privileged information may have been exposed the anonymous user.

Figure 18 is a snapshot of an SD model cross-clone pair reported by NICAD, showing the XMI source text of the cloned conversations. 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 original model file, and the potential clone Id *(pcid).*

In order to do the cross-clone set differencing, we used the NICAD-generated XML reports of clone pairs (without the XMI source), as shown above. These reports provide a list of all of the *pcid* pairs for the cross-clone pairs detected between each of the (AdminUser x ForcedAnon) and the (AdminUser x Anon) model comparisons. These lists were imported into an Excel spreadsheet and sorted to expose the Admin conversations that are present in the (Admin x ForcedAnon) clone pairs but not present in the (Admin x Anon) pairs. These suspicious ForcedAnon conversations were then reported as potential access violations.

```
<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">
        ...
        <eventTag event="SendOp179839">
            <packagedElement xmi:type="uml:SendOperationEvent" xmi:id="SendOp179839" name="
                   SendOperationEvent179839" operation="ownedOp179839">
            <operationTag operation="ownedOp179839">
                <ownedOperation xmi:type="uml:Operation" xmi:id="ownedOp179839" name="Select(posting,p . post_id = $
                       post_id and t . topic_id = p . topic_id and f . forum_id = p .  forum_id and pt .  post_id = p .  post_id and
                        u .  user_id = p .  poster_id)"  precondition="48550">
                    <ownedRule xmi:type="uml:Constraint" xmi:id="48550" name="Select" constrainedElement="
                           ownedOp179839">
                     ...
                    </ownedRule>
               <ownedParameter xmi:type="uml:Parameter" xmi:id="AcID48550" name="228"/>
               <ownedParameter xmi:type="uml:Parameter" xmi:id="PN48550" name="225"/>
                ...
            </operationTag>
            </packagedElement>
        </eventTag>
        ...
    </fragment>
    </BES>
</source>
<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">
        ...
        <eventTag event="SendOp9151">
           <packagedElement xmi:type="uml:SendOperationEvent" xmi:id="SendOp9151" name="SendOperationEvent9151"
                  operation="ownedOp9151">
            <operationTag operation="ownedOp9151">
                <ownedOperation xmi:type="uml:Operation" xmi:id="ownedOp9151" name="Select(posting,p . post_id = $
                       post_id and t . topic_id = p . topic_id and f . forum_id = p .  forum_id and pt .  post_id = p .  post_id and
                        u .  user_id = p .  poster_id)"  precondition="52612">
                    <ownedRule xmi:type="uml:Constraint" xmi:id="52612" name="Select" constrainedElement="
                           ownedOp9151">
                 ...
                <ownedParameter xmi:type="uml:Parameter" xmi:id="AcID52612" name="228"/>
                <ownedParameter xmi:type="uml:Parameter" xmi:id="PN52612" name="225"/>
                </ownedOperation>
                ...
            </operationTag>
</packagedElement>
</eventTag>
 ...
</fragment>
</BES>
</source>
</clone>
```

Fig. 18 An SD Model Cross-Clone Pair Reported by NICAD (in XMI form)

4.2.2 Evaluation

In order to evaluate the accuracy of our lightweight security analysis using SD cross-clone detection, we compared the results to the previously published security analysis of the same models using heavyweight SecureUML model checking [4]. Each suspicious conversation clone was traced to its associated actions in the original web application. Each action represents an SQL database access and is identified in the model by an action id (AcId). By comparing the action ids of the suspicious conversation clones to those identified as potential security violations in the published

Table 4 Cross-clone detection results after normalization at a difference threshold of 10%

Models Compared	# Cross-clone Pairs	# Cross-clone Classes	Analysis time (s)
Admin x ForcedAnon	1154	31	62.1
Admin x Anon	3030	19	73.8
Set Difference (Suspicious Conv.)	62		

SecureUML analysis, we are able to evaluate the accuracy of our cross-clone based lightweight method.

Table 4 shows the results of the cross cloning experiment done using a difference threshold of 10%. For the first cross-clone detection, comparing the Admin model to the ForcedAnon model (Admin x ForcedAnon), the NICAD cross clone detector reports 1154 conversation clone pairs. Based on the similarity of the clones, the reported cross clone pairs are grouped into 31 clone classes. Similarly, the cross-clone detection comparing the Admin model to the Anon model (Admin x Anon) reported 3030 clone pairs grouped into 19 clone classes. Differencing the clone pairs reported in (Admin x ForcedAnon) with those in (Admin X Anon) yielded 62 suspicious conversations.

The 62 suspicious conversation clones were traced to their ActIds in the original models, yielding a total of 14 action ids, each corresponding to a unique SQL database access.

The original SecureUML model analysis reported by Alalfi et al. [4] identified 12 actions as access violations in the ForcedAnon model, all of which appeared in our list of 14. Of the 62 suspicious pairs identified by the approach in this paper, 58 are instances of the 12 unauthorized database accesses identified in the previously published security analysis, and 4 are instances of 2 other database accesses which are benign (that is, they are false positives by comparison with the previously published analysis).

Thus based on the standard definitions, our lightweight security analysis using only SD clone detection on the recovered SD models yielded a recall of 12/12 or 100%, and a precision of 12/14 or 86% by comparison with the previously published and validated SecureUML model checking analysis of Alalfi et al. [4]. This is a remarkable result: the SecureUML analysis uses two more models and three more steps to achieve essentially the same result that we obtain using simple cross-clone detection and set differencing.

4.3 Threats to validity

There are several possible threats to the validity of our results. The first is that we have used a set of recovered sequence diagram models for our experimental validation. The extent to which these models are representative will affect the applicability of the results. The recovered sequence diagrams were derived from execution of a single but production medium-sized web application. They span a set of several dynamic pages visited by users in three different roles, and include some very large models, making them good candidates for the experiment. More sample models or recovered models from other applications would generalize our results.

Second, our approach is widely tuneable, depending on the purpose of the clone analysis. In this paper, we discussed one example application on which this approach can provide accurate answers, with high precision and recall when compared to an independent heavyweight approach to the problem. While the case study used specific values for the threshold, renaming, minimum and maximum number of lines per conversation parameters, different values of these parameters may be more appropriate or provide better results for other kinds of applications.

Third, there are very limited available resources when it comes to finding model repositories for similar experiments. The only tool for sequence diagrams that is comparable to ours is Störrle's *match* tool [30], which handles only models built using magicDraw and expects them in the mdxml format. The match tool is also designed for target models built using a forward engineering approach. When we tried applying the tool on one of our large models after transforming it to the mdxml format, the match tool was unable to digest the model. Thus for a lack of comparable tools to compare our results to, we were forced to hand validate the correctness of our results. However, our tool is based on the NiCad engine, which has been validated against all existing tools designed for software clone detection in a recent comprehensive experiment, which found that NiCad outperformed all other clone detection tools in both precision and recall [31].

In addition to hand validating our results, we evaluated the precision and recall of our approach using a specific application case study, detecting access control anti-patterns. In that case study the approach gave promising results. More related experiments are needed to generalize our findings.

5 Related work

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 then 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 to be a bad smell.

Tree comparison has been used by Rattan et al. [19] for finding duplicates in class diagrams from the XMI representation using the DOM's API and XML parsing.

Rubin et al. [23, 24] 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 [28, 29] talks about the challenges and possibilities of clone detection in all types of UML domain models. His work is based on an earlier work on model matching and model querying [27]. He observes that UML models are loosely connected graphs of heavy nodes, and implements a graph matching algorithm in Prolog, representing models as a set of facts, and using Prolog rules to find clones using various similarity heuristics . The clone detection algorithm and the evaluation of the heuristics is implemented as the MQ$_{lone}$ tool, a plugin in the MagicDraw UML CASE tool that reports clones to the user.

Like us, Störrle handles near-miss (Type 3) clones, but using metrics on the graph structure of the models rather than our approximate string matching. The distinguishing characteristic of our work is the identification and extraction of separate "conversations". While Störrle works on entire models and derives his graphs from the UML/UMI representation in the raw, we identify, normalize and separate sequence diagram conversational units representing meaningful and complete interactions. His method is general rather than tailored to sequence diagrams models and their meaning.

Nejati et al. [17] use a match function to compare the input models using both static (structural and textual attributes like element names) and behavioural (to identify element with similar dynamic behaviours) 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 behavioural 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 behavioural 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.

Al-Batran et al. [1] identify a number of semantics-preserving transformations that allow for detection of semantically equivalent Simulink clones. By performing these transformations, model clone detection recall is increased: semantically similar model clone instances are returned in addition to the structurally similar clones detected by other approaches. We may be able to incorporate their work into our approach by representing these transformations as textual source transformations and applying them to our normalized NICAD SD model representations. .

Of these techniques, only Liu et al. and Störrle handle UML 2.0 sequence diagrams, and only Liu et al. 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 with the exception of Störrle, only exact (Type 1) clones are handled. By contrast, our work uses a similar approach to the one developed by Alalfi et al. [6] 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. No other method enriches the precision of their similarity detection using such localization.

The NICAD clone detector [10] has been successfully used in finding clones in many source code languages. It is a hybrid 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, statements, or even classes in object-oriented languages. For modelling languages such as Simulink, a new tool called Simone [6] has been based on NICAD to identify near miss subsystem clones in Simulink models. Simone adapts and specializes NICAD to enable scalable and accurate clone detection in large scale Simulink models.

In previous work, we surveyed the entire area of code-clone detection [21]. NICAD was chosen as the code-clone technique to adapt as the basis of our approach because of its parsing, normalizing, and text-comparing abilities and because it was specifically designed to efficiently detect near-miss clones, something which had not yet been accomplished in the model clone detection domain.

We have also surveyed work on model comparison techniques [26], which included ConQAT and ModelCD. The majority of research in the area of model comparison is based on finding corresponding and differing model elements in a set or sets of models and much of it is geared towards model versioning. Model clone detection, especially near-miss model clone detection, differs from this idea: Model clone detection attempts to find a group of similar or related elements that have likely been reproduced from one another rather than explicitly trying to identify what individual elements are the same or are different. Thus, many of approaches in our survey are not applicable for model clone detection. The only approaches that may be leveraged are those that use similarity based metrics for comparison, such as EMFCompare [8], which performs similarity comparison on structural system models. We leave clone detection in that area as future work, as we are currently interested in SD behavioral models only.

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. In our case, we use cross-clone detection to identify patterns of conversations in the administrator model that contain actions (SQL accesses) at the administrator level in anonymous user models given access to administrator links with forced browsing.

6 Conclusions and Future Work

In this paper we have presented a practical approach to identifying near-miss cloned conversations in behavioural models, using consolidation and contextualization of the XMI interchange representation of UML sequence diagram models to identify and compare interaction sequences for clones.

In our experiments, our approach has efficiently detected Type 3 (exact near-miss) conversation clones in seven sequence diagrams of various sizes reverse-engineered from monitored interactions of web applications. Depending on the analysis, a set of normalizations may need to be applied to further refine the results to only include the most significant clones.

We have applied our approach on the problem of identifying access control security vulnerability patterns in recovered models of interaction with web applications. Our approach shows promising results with high precision and recall compared to a state-of-the-art model-checking based method. All results were obtained in less than 1.1 minutes for sequence diagrams of up to half a million XMI lines with between 30 to 1232 conversations, exclusive of the time required for preprocessing (contextualization), which required a maximum of about 33.3 minutes for the largest model. We are presently working on enhancing the contextualization stage to further improve the total performance.

Currently, the results we obtain from the clone detector are presented in NICAD's default XML and HTML text formats. We plan to trace the clones back to the original diagrams and visualize them in the model. We believe that our approach can relatively easily be extended to other kinds of UML and behavioural model representations.

We are interested in testing our approach on behavioural models other than SD, such as statecharts, and to experimenting with a large variety of models of various types, including those with more advanced features such as loops and components. It will be interesting to test our approach on behavioural models designed using forward engineering, with the aim of refactoring to improve design and maintenance properties of the models.

Acknowledgments

This work is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) as part of the NECSIS Automotive Partnership, and by the Ontario Research Fund through a Research Excellence grant.

References

1. B. Al-Batran, B. Schätz, and B. Hummel. Semantic clone detection for model-based development of embedded systems. *Model Driven Eng. Languages and Syst.*, 6981:258–272, 2011.
2. Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. Automated reverse engineering of UML sequence diagrams for dynamic web applications. In *1st International Workshop on Web Testing, WebTest 2009*, pages 287–294, 2009.

3. Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. WAFA: Fine-grained dynamic analysis of web applications. In *11th International Symposium on Web Systems Evolution, WSE 2009*, pages 41–50, 2009.
4. Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. Automated verification of role-based access control security models recovered from dynamic web applications. In *14th International Symposium on Web Systems Evolution, WSE 2012*, pages 1–10, 2012.
5. Manar H. Alalfi, James R. Cordy, and Thomas R. Dean. Recovering role-based access control security models from dynamic web applications. In *12th International Conference on Web Engineering, ICWE 2012*, pages 121–136, 2012.
6. Manar H. Alalfi, James R. Cordy, Thomas R. Dean, Matthew Stephan, and Andrew Stevenson. Models are code too: Near-miss clone detection for Simulink models. In *28th IEEE International Conference on Software Maintenance, ICSM 2012*, pages 295–304, 2012.
7. Elizabeth P. Antony, Manar H. Alalfi, and James R. Cordy. An approach to clone detection in behavioural models. In *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, pages 472–476, 2013.
8. C. Brun and A. Pierantonio. Model differences in the Eclipse modelling framework. *The European Journal for the Informatics Professional*, pages 29–34, 2008.
9. James R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.
10. James R. Cordy and Chanchal K. Roy. The NICAD clone detector. In *19th IEEE International Conference on Program Comprehension, ICPC 2011*, pages 219–220, 2011.
11. Thomas R Dean, James R Cordy, Andrew J Malton, and Kevin A Schneider. Agile parsing in TXL. *Automated Software Engineering*, 10(4):311–336, 2003.
12. M.R. Farhadi, B.C.M. Fung, P. Charland, and M. Debbabi. BinClone: Detecting code clones in malware. In *8th International Conference on Software Security and Reliability, SERE 2014*, pages 78–87, June 2014.
13. François Gauthier, Thierry Lavoie, and Ettore Merlo. Uncovering access control weaknesses and flaws with security-discordant software clones. In *29th Annual Computer Security Applications Conference, ACSAC 2013*, pages 209–218, 2013.
14. Saruhan Karademir, Thomas Dean, and Sylvain Leblanc. Using clone detection to find malware in Acrobat files. In *23rd Conference of the Center for Advanced Studies on Collaborative Research, CASCON 2013*, pages 70–80, 2013.
15. Hui Liu, Zhiyi Ma, Lu Zhang, and Weizhong Shao. Detecting duplications in sequence diagrams based on suffix trees. In *13th Asia-Pacific Software Engineering Conference, APSEC 2006*, pages 269–276, 2006.
16. Douglas Martin and James R. Cordy. Towards web services tagging by similarity detection. In *The Smart Internet*, pages 216–233, 2010.
17. Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. Matching and merging of statecharts specifications. In *International Conference on Software Engineering, ICSE 2007*, pages 54–64, 2007.

18. OWASP. Forced browsing, *https://www. owasp.org/index.php/Forced_browsing*, last access November 2013.
19. Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Model clone detection based on tree comparison. In *IEEE India Conference, INDICON 2012*, pages 1041–1046, 2012.
20. C. K. Roy and J. R. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *16th Int. Conf. on Program Compreh.*, pages 172–181, 2008.
21. C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.
22. Chanchal Kumar Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.
23. Julia Rubin and Marsha Chechik. From products to product lines using model matching and refactoring. In *2nd International Workshop on Model-Driven Software Product Line Engineering, MAPLE 2010*, pages 155–162, 2010.
24. Julia Rubin and Marsha Chechik. Combining related products into product lines. In *15th International Conference on Fundamental Approaches to Software Engineering, FASE 2012*, pages 285–300, 2012.
25. Rob Shapland. Forced browsing: Understanding and halting simple browser attacks, *http://www.computerweekly.com/answer/Forced-browsing-Understanding-and-halting-simple-browser-attacks*, last access December 2013.
26. M. Stephan and J. R. Cordy. A survey of methods and applications of model comparison. Technical Report 2011-582 Rev. 2, Queen's Univ., 2011.
27. Harald Störrle. VMQL: A generic visual model query language. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2009*, pages 199–206, 2009.
28. Harald Störrle. Towards clone detection in UML domain models. In *VIII Nordic Workshop on Model-Driven Software Engineering, ECSA 2010 workshops*, pages 285–293, 2010.
29. Harald Störrle. Towards clone detection in UML domain models. *Software and System Modeling*, 12(2):307–329, 2013.
30. Harald Störrle. MACH 5 hypersonic, http://www2.compute.dtu.dk/ rvac/hypersonic/, last access February 2015.
31. Jeffrey Svajlenko and Chanchal K. Roy. Evaluating clone detection tools with bigclonebench. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pages 131–140, 2015.
32. WatirCraft. Watir, *http://watir.com*, last access November 2014.