An Approach to Clone Detection in Behavioural Models

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

School of Computing, Queen's University, Kingston, Canada
{antony, alalfi, cordy}@cs.queensu.ca

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

I. Introduction

UML behavioural models, or 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 backend database and various threads within them, sequence diagrams document behaviour as sequences of interactions between the lifelines using events, messages, and other communications. Sequence diagrams can be used in forward engineering to specify intended behaviour, or in reverse engineering to observe and document actual behaviour.

In our previous work in web application security analysis using role-based access control [1, 2], we have reverse engineered the run-time behaviour of web applications to UML sequence diagrams that describe the entire history of interactions in a web application session. Using an automated test harness based on WATIR [3] to exercise the application in various different roles, we were able to document the behaviour of the application for users in those roles and compare it to the behaviour of other roles.

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

In this paper, we propose an automated approach to analysing such models to identify repeated patterns of similar interactions in them using the near-miss code clone detector NiCad [4]. Clone detection is 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.

In our initial experiments, we have been able to identify a significant number of duplicated conversations with varying degrees of similarity in an initial set of sequence diagram models reverse engineered from the dynamic behaviour of the PhpBB web application exercised in various user roles. Clone detection in behavioural models has many applications. For example, clones in reverse engineered behavioural models from web applications can be used to find worrisome patterns such as security violations.

II. Background

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

Figure 1 shows an example highlighting the main elements in a basic sequence diagram. These include *Lifelines*, *Messages*, *Behavior Execution Specification (BES)*, *Message Occurrence Specification (MOS)*, *Events*, *Classes*. Other elements such as Events, Classes, Properties etc. are not shown in the figure for readability.

Sequence diagrams can be represented in text using XMI representation, the XML-based standard interchange format used for exchanging models between various modelling tools [5]. Interactions in a sequence diagram are represented in XMI format as "fragments", as shown in Figure 4. Every element of a sequence diagram has an XMI Id identified by "xmi:id" and a set of attributes that shows the relationship of this element with the others. In Figure 4, the attribute values have been renamed to aid comprehension.

Sequence diagram model-clone detection entails discovering similar or identical sequences of behavioural interaction ("conversations"). Unlike source code, which is represented

978-1-4799-2931-3/13/$31.00 © 2013 IEEE

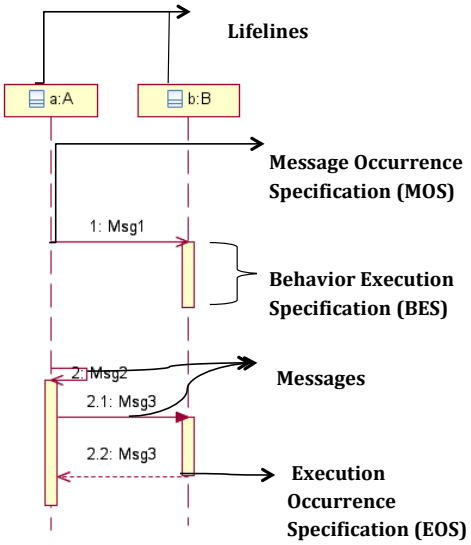

Fig. 1. Elements of a basic sequence diagram

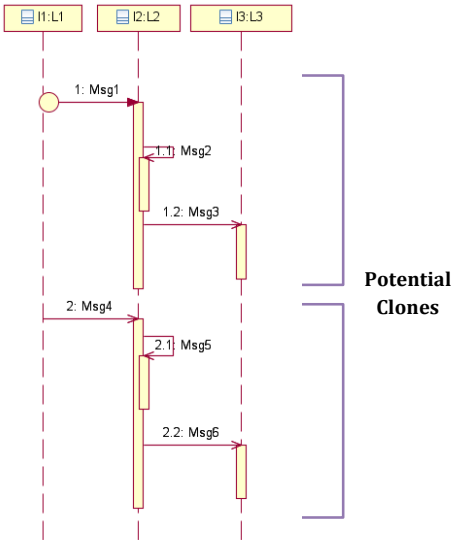

Fig. 2. Example of potential clones

as linear text, models are typically represented visually, as box-and-arrow diagrams. Model clones can thus be thought of as similar sub graphs of these diagrams. Figure 2 shows an example of a potential SD model clone.

III. Approach

Our approach (Figure 3) consists of three stages. The first stage transforms the XMI sequence diagram serialization into a contextualized form in which we localize references to reveal the hidden structure of the textual SD representation. This transformation identifies self-contained units of behavioural interaction that we will use later as the fragments for clone comparison. The second stage normalizes the resulting contextualized units to remove irrelevant elements and rename irrelevant naming differences to make the process of clone identification more accurate. The final stage uses a standard code clone detector to automatically identify cloned behavioural interactions from the large set of contextualized and normalized interaction units. In the following subsections we discuss each of the above stages in more detail using a running example.

A. Clone Detection Analysis

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 detection technique, NiCad [4], to work on behavioural models.

In previous work we have used NiCad to identify near-miss sub-system clones in Simulink models [6]. As a code clone detector, NiCad was mainly designed to identify clones in languages with a well defined nesting structure. Unlike Simulink models, the XMI serialization of UML sequence diagrams does not have an explicit nesting structure, thus one of the main challenges in adapting it to this task was understanding how to reverse engineer the hidden nested structural representation of interaction conversations from the flat representation of the original XMI SD serialization. This is the purpose of the contextualization stage that we discuss in Subsection III-B below.

A second challenge was the identification of the right level of granularity for comparison. In SD conversations, granularity at the message level is very small and would lead to huge number of clones that would not be useful and probably irrelevant for most applications. On the other hand, comparing the entire conversation of a whole lifeline would likely reveal very few clones and would miss many interesting sub similar conversation. Thus we decided to break conversations into smaller windows at the sub-conversation level. This led us to choose SD Behavioural Execution Specifications (BESs) as the best units for comparison. BESs have identified start and finish points that correspond roughly to the start and end of the sub-conversations happening on specific lifeline. The example clone pair shown in Figure 2 are sub-conversation clones at the BES level of granularity.

B. Transformation and Contextualization

In the flat structure of the XMI sequence diagram representation (Figure 4), there is little locality, and fragments and elements of sub-conversations are spread across the text. XML attributes and the order of elements are used to reference and implicitly group related elements. Behaviour Execution Specifications (BESs), for example, reference the lifeline they are part of using the *covered* attribute, and the sequence of messages and events of the conversation using the *start* and *finish* attributes, and these elements in turn refer to their parts in similar fashion. In order to restructure this scattered representation for comparison purposes, we need to recursively gather the referenced and related elements of the BES conversations together and organize them explicitly into the structures they represent.

We have used TXL [7] source transformations to implement this restructuring. TXL is a structural transformation system, thus the first step was to define a general grammar for parsing the original XMI representation of the sequence diagrams.

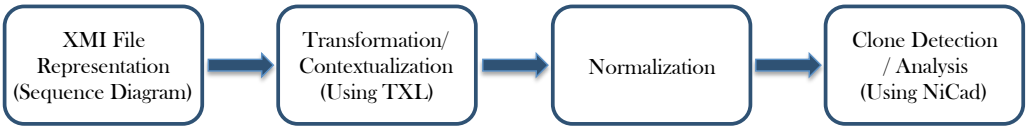

Fig. 3. Steps of our approach

```
<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>
 <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="SOE2Id" name="SendOperationEvent2" operation="CLSSL2OO2Id"/>
 <packagedElement xmi:type="uml:ReceiveOperationEvent" xmi:id="ROE2Id" name="ReceiveOperationEvent2" operation="CLSSL2OO2Id"/>
 <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. 4. Step I: Identification of BES fragments in the XMI representation *(The example is only a snapshot of the larger SD model)*

Next, a set of source transformation rules were created to identify, consolidate and contextualize BES specifications into self-contained hierarchical units representing the sub-conversations to compare. The restructuring process consists of 3 steps:

1) Identify: This step identifies all the Behaviour Execution Specification elements in the model representation and restructures them into units identified by *<BES>...</BES>* tags. In the example of Figure 4, three BES elements have been identified. Figure 5 shows the result of tagging and applying the consolidation of the next subsection to the identified BESs.

2) Consolidate: Once the tag container for the BES is created, we gather and nest all of the BES's conversation elements into the container. Each BES element has a *start* and *finish* attribute, the ids of which specify the elements of the flat representation that begin and end the BES's conversation. Because message, behaviour and event occurrences are totally ordered in the XMI representation, this step primarily involves moving the adjacent elements of the conversation before and after the BES inside the <BES> tag to consolidate the whole conversation (Figure 5).

3) Contextualize: Consolidated BES conversations consist of embedded BESs, Message Occurrence Specifications (MOSs) and Execution Occurrence Specifications (EOSs) describing the conversation's interactions with other lifelines. Similarly to BESs themselves, these use their XML attributes to link to the elements such as messages, types and other lifelines that describe their meaning. For example, in Figure 5, the third embedded BES unit (highlighted in orange), has two MOS and one EOS fragment.

To contextualize a consolidated BES, each MOS and EOS fragment in the BES is converted to a container tag, and the elements referred to by the attributes of the fragment are then 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 6. These attributes represent the covered lifeline, event and message of this particular Message Occurrence Specification.

Contextualization proceeds recursively. For example, to inline the *covered* attribute of the second MOS in the third embedded BES example of Figure 5, the *<lifeline>* element with id *LFNI3Id* referred to by the attribute is located and copied into the container tag of the MOS. From the inlined *<lifeline>* element, the *ownedAttribute* element with xmi:id="PROPl3Id" referred to by the lifeline's *represents* attribute is then inlined to include the property/object of the class that the lifeline covers. 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="ROE3Id", 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 send and receive MOS's of the message along with the *<ownedConnector>* element referred to by its *connector* attribute, which represents the end points of the message where they connect to the lifelines.

The EOS fragment of the example BES is converted to a container tag and contextualized by recursively inlining

```
<BES start="MOS2Id" finish="EOS3Id">
      <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 start="MOS4Id" finish="EOS1Id">
             <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 start="MOS6Id" finish="EOS2Id">
             <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>
```

Fig. 5. Step II: Consolidation of BES conversation elements.

```
<BES start="MOS6Id" finish="EOS2Id">
      <fragment xmi:type="uml:BehaviorExecutionSpecification" xmi:id="BES3Id" covered="LFLNl3Id" start="MOS6Id" finish="EOS2Id">
         ...
         <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">
               </ownedAttribute>
            </lifeline>
            <packagedElement xmi:type="uml:ReceiveOperationEvent" xmi:id="ROE3Id" name="ReceiveOperationEvent3" operation="CLSSL3OO1Id">
               <ownedOperation xmi:type="uml:Operation" xmi:id="CLSSL3OO1Id" name="Msg3">
               </ownedOperation>
            </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>
               <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">
                     </ownedAttribute>
                  </lifeline>
                  <packagedElement xmi:type="uml:SendOperationEvent" xmi:id="SOE3Id" name="SendOperationEvent3" operation="CLSSL3OO1Id">
                  </packagedElement>
               </fragment>
               <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>
            </message>
         </fragment>
         <fragment xmi:type="uml:ExecutionOccurrenceSpecification" xmi:id="EOS2Id" covered="LFLNl3Id" event="EE1Id" execution="BES3Id">
            <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">
               </ownedAttribute>
            </lifeline>
            <packagedElement xmi:type="uml:ExecutionEvent" xmi:id="EE1Id" name="ExecutionEvent1">
            </packagedElement>
         </fragment>
      </fragment>
</BES>
```

Fig. 6. Step III: Contextualization of embedded elements and attributes

the elements referred to by its *covered*, *event* and *execution* attributes in a similar fashion, yielding the completely contextualized BES shown in Figure 6 for the example.

Larger SDs may contain any number of MOS, EOS, embedded BESs and other fragments, all of which are contextualized by inlining in the same way. Each element is restructured to include the elements referenced by its attributes. Inlining the elements of each BES in this way creates a set of self-contained interaction units for comparison in clone detection.

C. Normalization

Thus far in our work we have done little normalization of BESs. In order to improve the precision and recall of our clone results, we are implementing a filtering transformation to remove irrelevant elements from the comparison.

Code clone detection techniques can be categorized according to the types of clones they can identify, and in [6] we have adopted a corresponding categorization for model clone types. Type 1 (exact) model clones are identical model fragments, ignoring variations in visual presentation, layout, and formatting. Type 2 (renamed) model clones are structurally identical model fragments, ignoring variations in labels, values, types, and the variations from Type 1. Type 3 (near-miss) model clones are model fragments with further modifications such as small additions or removals of messages in the conversation, in addition to the variations from Type 1 and 2 clones.

In order to identify Type 2 (renamed) behavioural clones, a blind or consistent renaming of elements will be necessary. Thus far, we detect all BES near-miss exact clones (Type 3-1), but only some near-miss renamed clones (Type 3-2).

IV. Early Evaluation

Currently, we have evaluated our approach on a set of reverse-engineered models from execution traces of interactions in web applications such as PhPBB. These sequence diagram models capture the interactions of different users in different roles while exercising the applications from a browser. Thus far, we have applied our approach on four

TABLE I
EARLY RESULTS USING BES CLONE DETECTION

Model	# Lines	#BES	Difference Threshold	#Clone Pairs	#Clone Classes
1	752	30	35%	18	6
2	5376	223	35%	1260	12
3	9504	142	35%	407	16
4	53861	314	35%	3330	22

models of various sizes and are currently experimenting with larger models to test the scalability of our approach.

Because NiCad is a parser-based technique, precision is not an issue [8]. Recall can be improved using additional filtering and normalization steps [6], which is our next priority. Table I shows our early results for a number of web application sequence models, giving the number of extracted conversations (BESs), the number of cloned conversations identified (clone pairs), and the number of groups into which these pairs can be clustered (clone classes). The near-miss difference threshold allows for slight variances in similar conversations.

V. RELATED WORK

Liu et al. [9] 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 elements as arrays and used the longest common prefix to check for duplicates. Duplicate fragments were refactored if they were considered a bad smell.

Tree comparison has been used by Rattan et al. [10] for finding duplicates in class diagrams from the XMI representation using the DOM's API and XML parsing. Rubin et al. [11] work with both structural and behavioural models, specifically class and statechart diagrams with the intent of refactoring models into product lines.

Störrle [12, 13, 14] talks about challenges and possibilities in clone detection in all types of UML models. His work is based on earlier work on model querying, where he used Prolog to represent model elements as facts and models as a set of facts, then encode Prolog rules to find clones using a similarity measure of model elements to identify clones.

Of these techniques, only [9] and [12] handle UML 2.0 sequence diagrams, and only [9] also targets conversations. Our work is based on identifying similar patterns in sequences of message interactions using BES in SDs. With contextualization and consolidation steps, the BES units created are complete sequences of interactions and the clones reported are thus extractable as entire conversations. Our work also differs from others in its goal of characterizing and identifying patterns of potential security violations in web applications.

None of the other methods have been tested on large models, and only exact (Type 1) clones are handled. By contrast our work uses a similar approach to the one developed in [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.

VI. CONCLUSION

In this paper we propose an approach to identify 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 initial experiments, our approach has efficiently detected Type 3-1 (exact near-miss) conversation clones in four sequence diagrams of various sizes reverse-engineered from monitored interactions with web applications. In our next step we will analyze the clones reported to distinguish which are important and which are not. 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.

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 be applied to other kinds of UML and behavioural model representations.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] M. H. Alalfi, J. R. Cordy, and T. R. Dean, "Automated Reverse Engineering of UML Sequence Diagrams for Dynamic Web Applications," in *ICSTW*, 2009, pp. 295–302.

[2] ——, "WAFA: Fine-grained Dynamic Analysis of Web Applications," in *WSE*, 2009, pp. 41–50.

[3] WatirCraft, "WATIR," last access March 2009. [Online]. Available: http://wtr.rubyforge.org

[4] J. R. Cordy and C. K. Roy, "The NiCad clone detector," in *ICPC*, 2011, pp. 219–220.

[5] "XMI," http://www.omg.org/spec/XMI/.

[6] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson, "Models are code too: Near-miss clone detection for Simulink models," in *ICSM*, 2012, pp. 295–304.

[7] J. R. Cordy, "The TXL source transformation language," *Sci. Comput. Program.*, vol. 61, no. 3, pp. 190–210, 2006.

[8] C. K. Roy and J. R. Cordy, "A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools," in *Mutation*, 2009, pp. 157–166.

[9] H. Liu, Z. Ma, L. Zhang, and W. Shao, "Detecting Duplications in Sequence Diagrams Based on Suffix Trees," in *APSEC*, 2006, pp. 269–276.

[10] D. Rattan, R. Bhatia, and M. Singh, "Model clone detection based on tree comparison," in *INDICON*, 2012.

[11] J. Rubin and M. Chechik, "Combining Related Products into Product Lines," in *FASE*, 2012, pp. 285–300.

[12] H. Störrle, "Towards clone detection in UML domain models," *Softw. and Syst. Modeling*, vol. 12, no. 2, pp. 307–329, 2013.

[13] ——, "VMQL: A generic visual model query language," in *VL/HCC*, 2009, pp. 199–206.

[14] H. Störrle, "Towards clone detection in UML domain models," in *ECSA*, 2010, pp. 285–293.