Electronic Communications of the EASST
Volume 63 (2014)

Proceedings of the Eighth International Workshop on Software Clones (IWSC 2014)

Clone Detection in Matlab Stateflow Models

Jian Chen Thomas R. Dean Manar H. Alalfi

13 pages

Guest Editors: Nils Göde, Yoshiki Higo, Rainer Koschke
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Clone Detection in Matlab Stateflow Models

Jian Chen Thomas R. Dean Manar H. Alalfi

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

Abstract: Matlab Simulink is one of the leading tools for model based software development in the automotive industry. One extension to Simulink is Stateflow, which allows the user to embed Statecharts as components in a Simulink Model. These state machines contain nested states, an action language that describes events, guards, conditions and actions and complex transitions. As Stateflow has become increasingly important in Simulink models for the automotive sector, we extend previous work on clone detection of Simulink models to Stateflow components.

Keywords: Model, State Machine, Stateflow

1 Introduction

Models play an increasingly important part in software development, particularly in areas where risk to life or property is an issue such as the automotive sector. Simulink[1] is a modelling language that has been widely used in the development of automotive embedded systems. One component of Simulink is Stateflow[2], an environment for modeling and simulating combinatorial and sequential decision logic based on hierarchical state machines (i.e. state charts [Har87]) and flow charts. Stateflow can be used to combine graphical and tabular representations, including state transition diagrams, flow charts, state transition tables, and truth tables, to model how systems react to events, time-based conditions, and external input signals. Stateflow is used to design logic for supervisory control, task scheduling, and fault management applications.

Software clones are segments of code that are similar according to some definition of similarity [Kos06]. Software clones have an impact on maintenance, and it is important to identify duplicate artefacts [ACD+12a]. The potential impact of identifying redundancy at the higher levels of abstraction provided by models makes clone detection in models important since it can help in testing design consistency and completeness before implementation.

Model clones are different from code clones, as they are based on matching sub graphs. In previous model clone types categorization by our research group [ACD+12a], three types of model clones are defined:

- 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.

[1] www.mathworks.com/products/simulink
[2] www.mathworks.com/products/Stateflow

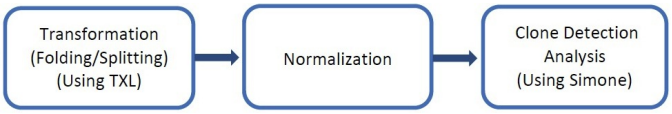


Figure 1: Steps of our approach

- Type 3 (near-miss) model clones are model fragments with further modifications such as small additions or removals of model elements, in addition to the variations from Type 1 and 2 clones.

For our purposes, clones in Stateflow are models that are structurally similar. For example, the same structure states and transitions with different labels, conditions and actions would be considered a clone. There are various techniques and tools for clone detection in program source code [RCK09] such as textual comparison and program dependency graph comparison. Rather than using computationally expensive sub-graph isomorphism to find similar graphs, we extend the approach our research group used for Simulink in SIMONE [ACD+12a] and apply clone detection techniques to the textual representation of the Stateflow model.

In Simulink, clones are structural in nature and follow natural syntactic boundaries such as subsystems. Stateflow is also represented as a nested set of graphs, but the graphs represent the behaviour of a component in the larger Simulink model as opposed to the structure of the model. The addition of Stateflow clone detection to SIMONE serves two purposes. The first is to detect common behavioural elements of the models that are expressed as state machines. The second is by extending the clone detection to Stateflow models, we can improve clone detection in those Simulink models that contain blocks that refer to the Stateflow models. SIMONE detects clones by considering model elements of the same type similar, so currently all blocks that encapsulate Stateflow models are also considered similar. We evaluate our approach on a large set of Stateflow models available to us from Mathworks.

2 Approach

Our approach, shown in figure 1 consists of three stages. The first stage transforms the Stateflow textual representation into a hierarchical textual structure as SIMONE's initial input. The second stage, implemented as a SIMONE plugin, normalizes the initial input to remove irrelevant elements and rename irrelevant naming differences to make the process of clone identification more accurate. The final stage identifies potential clone candidates and cluster them into classes. In the folllowing subsections, we discuss each of the stages in more detail.

2.1 Stateflow TXL Grammar

Our approach is implemented in TXL [Cor06], a structural transformation and parser-based language. Thus the first step is to build a Stateflow TXL grammar allowing TXL to parse Stateflow models. We derive a TXL grammar form a large set of example Stateflow models in the public domain by using iterative inference techniques. Figure 3 shows a small snippet of the Stateflow

:

```
Stateflow {
  machine {
    id     1
    name    "powerwindow"
    ...
  }
  chart {
    id     2
    name    "control"
    windowPosition [24 266 702 602]
    viewLimits  [0 843.043 2.915 444.795]
    zoomFactor  1.282
    screen   [1 1 1280 1024 1.041666666666667]
    treeNode       [0 22 0 0]
    ...
  }
  state {
    id     3
    labelString   "passengerneutral\nentry:\nmoveUp = 0;\nmoveDown = 0;"
    position   [724.059 27.423 98.524 90.095]
    fontSize   12
    ...
    treeNode          [15 0 0 6]
    ...
  }
  ...
  junction {
    id     23
    ...
    linkNode          [5 0 0]
    ...
  }
  transition {
    id     24
    labelString   "after(100,ticks)"
    src { ... }
    dst { ... }
    ...
    linkNode          [5 0 25]
    ...
   }
 ...
}
```

Figure 2: Example snippet of the textual representation used by Stateflow

TXL grammar. Our grammar identifies all observed elements of the Stateflow models, including machines, charts, states, translations, junctions, events, data, instances, targets and other elements.

2.2 Representation Transformation

The first stage of our approach is representation transformation. This stage performs two tasks. The first task is to transform the sequential representation of the model into a nested version that explicitly represents the hierarchy of the model. The second task is to separate the actions of the

:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Stateflow component
% Top level containing object, presumably used
% to separate stateflow from other simulink entities
% in the model file.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

redefine stateflow_list
   'Stateflow {    [NL][IN]
    [repeat stateflow_list_element]
         [EX]
   }          [NL]
end redefine

define stateflow_list_element
  [sf_machine_list]
 | [sf_chart_list]
 | [sf_state_list]
 | [sf_transition_list]
 | [sf_event_list]
 | [sf_data_list]
 | [sf_instance_list]
 | [sf_target_list]
 | [sf_junction_list]
end define
```

Figure 3: Example snippet of the inferred TXL grammar

state from the name of the state, representing each as separate attributes.

2.2.1 Explication of Hierarchical Structure

Figure 2 shows an excerpt of the textual representation of a Stateflow model, which in turn is embedded in a Simulink model file. Each item in the figure (i.e. state, transition, or junction) is stored sequentially, independent of the hierarchy inherent in the model. Instead, the hierarchy of the model is represented using the *treeNode* and *linkNode* attributes. The *treeNode* attributes are used in *chart* and *state* elements of the model to encode hierarchy by identifying the parent, first child and sibling elements. In the figure, the first state at the top level of the chart (or root of the Stateflow machine) has the id 22. State 22 is not shown in the figure due to length but is located before junction 23. The *linkNode* attributes are used in *junction* and *transition* elements to identify sibling elements at the same hierarchical level. The *src* and *dst* attributes of transitions identify the source and destination attributes of transitions.

SIMONE takes advantage of the natural nesting of model elements that is used in the Simulink textual representation. Subsystems are textually nested within the block that represents the subsystem at the next highest level of abstraction in the model. Thus to handle a Stateflow model, we must provide an initial transform that moves the textual description of any substates, junctions and transitions and nests them within the description of the parent state.

The transformation takes a folding approach that examines each element in turn and inserts it into the appropriate parent element. At the same time, the nested elements are sorted by type:

:

```
Model {
 ... simulink model...
}
Stateflow {
   chart {
      state{
         state {
        }
        transition{
        }
      }
   }
}
```

Figure 4: Simplified example of folded Stateflow model

first states, then transitions, and finally junctions. Figure 4 shows a simplified outline of the new representation. In the figure, the state flow section of the Simulink file contains a single chart, with a single state, that contains a single nested substate and one nested transition. We have omitted the other attributes to emphasize the nested structure.

2.2.2 Label Splitting

Figure 2 also shows that the state actions are encoded along with the state name into a single string given by the *labelString* attribute. The state shown in the figure has the name *passengerneutral* as well as an entry action that initializes two variables to zero. States may have actions associated with the entry and exit from the state, actions that are performed while the state is active and actions that are performed if an event occurs while the state is active. Transition labels are also complex, having triggers, conditions, condition actions and transition actions.

As our approach is based on comparing line as a whole, a difference in a single component of a state or transition label renders the entire line different. Thus we split the state labels into the constituent parts, each with its own attribute. The state name, if present, is encoded using a new *textlabel* attribute. The entry, during and exit actions, when present, are encoded using separate attributes of similar names (*entrylabel*, *duringlabel*, and *exitlabel*). If the author of the state machine model has included formatting such as newlines into the actions, such as in the example in figure 2, the actions are split into multiple attributes with the same name. Thus a change in code associated with an entry action will not also imply that the during and exit actions of the state are also different.

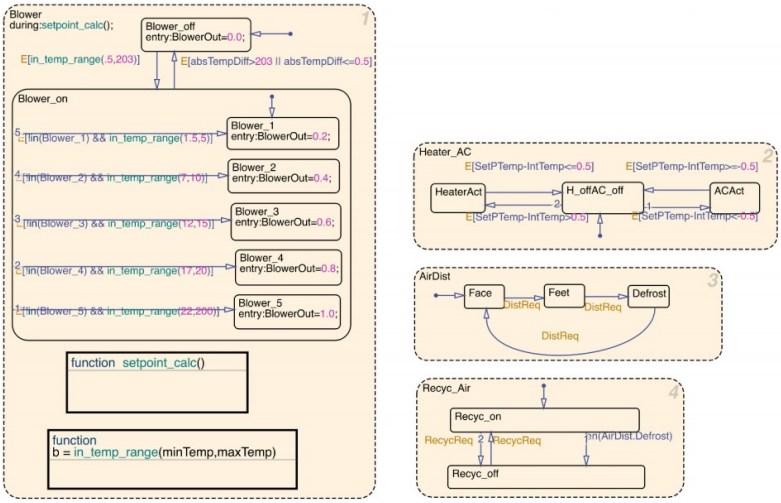
Figure 5 shows the labels generated for the state label in figure 2. The state name has been provided in a *textlabel* attribute, and the entry actions have been transformed to three *entrylabel* attributes.

Transitions do not have names. However we also separate each of the components of the transition labels into separate attributes. These components are identified by the new attributes *eventlabel*, *conditionlabel*, *condition action*, and *actionlabel*. This provides us finer grained control over the comparisons used for clone detection. For example we can distinguish between a change in an event label from a change to both an event label and the code given by the action

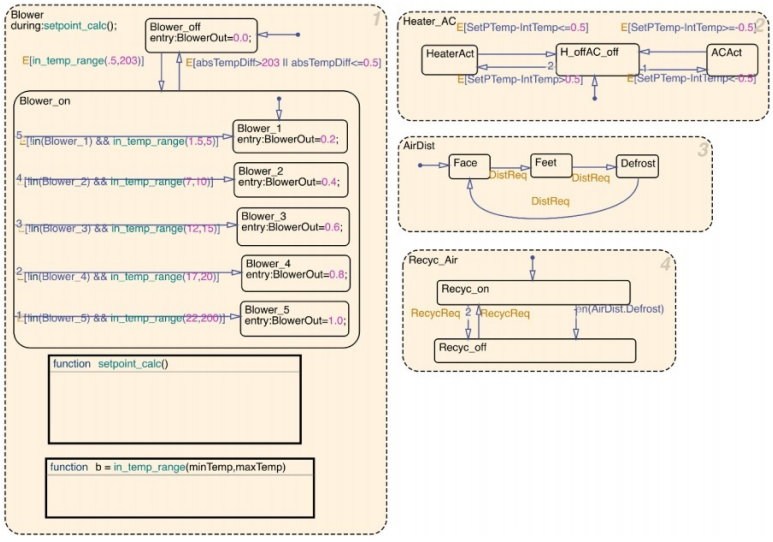
:

```
entrylabel "entry:"
entrylabel "moveUp = 0;"
entrylabel "moveDown = 0;"
textlabel "passengerneutral"
```

Figure 5: Example of a split state label



(a) ClimateControlSystem/Temperature Control Chart



(b) Temperature Control Chart

Figure 6: Example of Stateflow clones from sldemo_auto_climate_elec.mdl and sldemo_auto_climatecontrol.mdl in Matlab demo automotive models

of the transition.

2.3 Extractor Plugin

The extractor in SIMONE is responsible for identifying and extracting the clone candidates from the models. Our extractor for Stateflow provides two granularities of clone candidates. The first, *chart granularity* extracts all of the Stateflow charts as clone candidates. Charts in Stateflow represent entire machines. A Simulink model may have more than one chart, each of which may be instantiated multiple times as blocks in the Simulink Model. The second level of granularity, *state granularity*, extracts all states in all charts as clone candidates. This allows us to identify cloned state machines that are nested within states.

We tested our extractor on the set of Stateflow demo models provided by MathWorks. There are total of 269 model files that contain Stateflow in the demo set. Our initial, baseline experiment uses only the candidates extracted at both levels of granularities without any normalization. Using a threshold of 30% difference(i.e. at least 70% of the lines are the same) and a minimal clone size of 100 lines, we were able to find several clones in the demo set. A clone class is the equivalence class induced by the clone pair relationship. If a and b are clone pairs, and b and c are clone pairs, then a, b and c form a clone class. Figure 9, the *Extractor only* column, shows the initial results. We found 205 state clone pairs clustered in 24 clone classes, and 514 chart clone pairs clustered in 27 clone classes. Examination of the results showed relatively small Stateflow models, with limited nesting, making clone detection at the state level uninteresting.

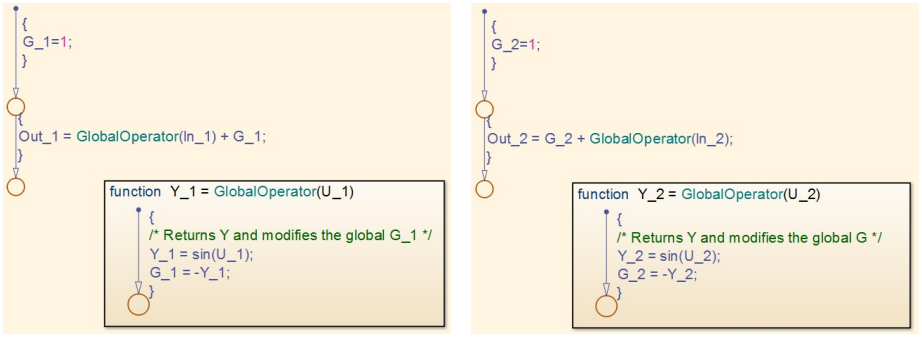
Further examination of the results reveal that models that are identical in the graphical view do not have one hundred percent similarity. The most obvious differences were differences in layout attributes, and normalizing these attributes could improve clone detection. To evaluate the effectiveness of the normalization of the states at improving clone detection, we have concentrated on clone detection of charts. These normalizations may also be applied at the state level of comparison.

2.4 Normalization

In this stage, we normalize the result of the model files from stage one. The task of this stage includes removing irrelevant elements, renaming elements and sorting elements. These steps can improve the precision and recall of the clone detection phase.

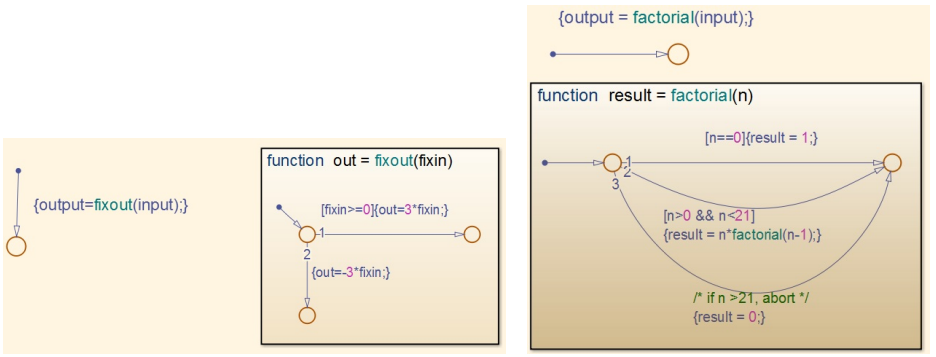
2.4.1 Filtering

In order to improve the detection of clones, we introduce a filtering plugin similar to the filtering plugin we used for Simulink models in our previous research. This plugin removes the irrelevant elements from the potential clones. While some of these irrelevant attributes are the same as used in Simulink, others were new. Since there is no published reference for the textual representation of Stateflow models, we were obliged to infer which of the attributes were important and which were irrelevant. Figure 2 shows some of these irrelevant elements: the position of each element, the font size used to display labels, the zoom factor of the chart. While these attributes are important to the human understanding of the model, they are not relevant to the semantic meaning



(a) ex_mg_hisl_0062/One (b) ex_mg_hisl_0062/Two

Figure 7: A Type 2(renamed model clone), both (a) and (b) in the ex_mg_hisl_0062 model. Simone similarity 81%.



(a) Fixed-Point Graphical Functions/Chart (b) Recursive Graphical Function Demonstration/Chart

Figure 8: A Type 3(near-miss model clone), Fixed-Point Graphical Functions in sf_fxptgf model and Recursive Graphical Function Demonstration in sf_gfrecursive. Simone similarity 75%.

of the models. Two models identical when comparing states, transitions, junctions and actions may actually contain differences based on how they are rendered for view. The differences in these attributes can overwhelm the similarities in the attributes that carry the semantic meaning of the model.

The similarity of some of the clone pairs identified by only using the extractor is increased when using these filters. However, the filters do not identify more clone pairs and clone classes. The filtering can improve some similarity but not significantly. Figure 6 shows an example from two different Stateflow demo models, sldemo_auto_climatecontrol and sldemo_auto_climate_elec, which include the identical Temperature Control Chart.

Total nontrivial states(1372) & charts(339)	Extractor Only		Filtered Only		Filtered & Renamed		Filtered, Sorted & Renamed	
	state	chart	state	chart	state	chart	state	chart
Clone pairs	205	514	198	378	317	1639	314	1482
Clone class	24	27	21	23	50	29	54	29

Figure 9: Initial results of the Stateflow model clones found in the Matlab demo set.

2.4.2 Renaming

While filtering improves the similarity of the clones, we found that there are still some clones we could identify manually that are still not detected. Some of the values of the attributes represent internal information such as the *id* number of a state and are used to build structural information such as the relationship between transitions and states.

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

We adapted the SIMONE blind renaming plugin to rename the attributes, giving them all the same value. Renaming significantly improved the similarity and also new cloned pairs were found. All detected clones have been checked by hand to compare the graphical representation, all clones found in the example set models were all valid clones.

Figure 7 shows an example type 2 clone of two different chart, One and Two, in the ex_mg_hisl_0062 model of the Simulink example set. As you can see from the figure, the structure is the same, but the labels have been changed, replacing the string "_1" with the string "_2". Figure 8 shows a type 3 clone between the Fixed-Point Graphical Functions Chart of the sf_fxptgf model and the Recursive Graphical Function Demonstration Chart of the sf_gfrecursive model of the Stateflow demo set. A new transition has been added and one junction has been removed, as well as naming and attribute changes to other transition and lines.

2.4.3 Sorting

The last source of difference in similar models was the order in which the elements were saved to the file. For example, there are two clone models *X* and *Y*, have the same states A B and C. In model X the order of the states are A B and C. In model Y the order of the states might be C B and A . In Simulink models, we use the *Type*, *Name*, *Source*, *Block*, and *Port* attributes as sorting criteria. Unfortunately, Stateflow does not have a name attribute we can use. Our solution is to sort the elements by size. Each state may contain a different number of substates, junctions and transitions. We sort the states by the number of nested elements. After sorting, the similarity of clones is improved, but no new clone pairs were detected.

Figure 10 and Figure 11 show the textual presentation of sf_aircraft_screen_library model in the Simulink demo set. They both have the identical states "zero","one","two","three" and "no valid signals" and they have different order inside the model file. We sort them by the size of each state, and we have the new order "no valid signals", "one", "three","two", and "zero".

:

```
chart {
   id 2
   name "Position Monitor,\n(No Filter)/Screen Signals/screen logic"
   ...
    state {
          id 4
          labelString "zero"
  ...
 }
 state {
          id 6
          labelString "three"
  ...
 }
 state {
          id 7
          labelString "two"
  ...
 }
 state {
          id 8
          labelString "one"
  ...
 }
 state {
          id 11
          labelString "no valid signals"
  ...
 }
...
}
```

Figure 10: Example snippet of sf_aircraft_screen_library

Figure 9 shows the total number of clone pairs and classes detected by each of these options. Filtering reduced the total number of clones by removing the false positives generated by similarities only in unimportant attributes. Renaming increased the number of clones detected by allowing different names to match. Sorting improved the quality resulting in slightly fewer clone pairs, but a few more clone classes.

3 Related Work

While code clone detection was extensively researched [RCK09], research on model clones identification has received less attention [DHJ+10]. Thus far, the majority of approaches are tailored for Simulink models[DHJ+10, ACD+12a, DHJ+08, ACD+12b, SASC12, PNN+09], and these techniqes either use graphical based comparison or text base techniques to do clone detection on Simulink models. None of them has been applied to Matlab Stateflow models.

Störrle [Stö13] proposed an approach to identify clones in UML models, specifically class, activity and use case diagrams, and claims the approach is extendable to simulink and Stateflow models. However, it has not been demonstrated on StateFlow. Störrle uses a different definition

:

```
chart {
   id 165
   name "Hydraulic Monitor,\n(No Filter)/Screen Signals/screen logic"
 ...
 state {
         id 167
         labelString "one"
  ...
 }
 state {
         id 171
         labelString "three"
  ...
 }
 state {
         id 172
         labelString "two"
  ...
 }
 state {
         id 173
         labelString "no valid signals"
  ...
 }
 state {
         id 174
         labelString "zero"
  ...
 }
...
}
```

Figure 11: Example snippet of Hydraulic Monitor of sf_aircraft_screen_library.

of model clone. His definition requires that the structure of the models be the same, and that the labels on each of the model elements be similar. Thus his approach identifies Type 1 and Type 2 near miss clones, but not Type 3 near miss clones.

4 Conclusion and future work

In this paper we extend SIMONE to perform clone detection on Stateflow models. The initial clone detection results from the Matlab example set are similar machines with variations in labels(i.e. state and transition names) and other attributes such as position. We still need to evaluate out approach on more Stateflow models, as well as to refine our SIMONE plugin to improve clone detection. We are also investigating explicating the state machines into the parent Simulink model in a similar manner to Grant et al. [SS11]. This will allow us to use similarity of state machines to improve the accuracy of Simulink clones. We also found some clone classes that appear to be embedded Matlab code for use by state and transition labels. Improving our approach to better deal with embedded code is also a line of future research.

Acknowledgements: This work is supported in part by NESERC, as part of the NECSIS Automotive Partnership, and by the Ontario Research Fund through a Research Excellence grant in model-driven engineering.

Bibliography

[ACD+12a] M. Alalfi, J. Cordy, T. Dean, M. Stephan, A. Stevenson. Models are code too: Near-miss clone detection for Simulink models. In *ICSM*. Pp. 295–304. 2012.

[ACD+12b] M. Alalfi, J. Cordy, T. Dean, M. Stephan, A. Stevenson. Near-miss model clone detection for Simulink models. In *IWSC*. Pp. 78–79. 2012.

[Cor06] J. R. Cordy. The TXL source transformation language. *Sci. Comput. Program.* 61(3):190–210, 2006.

[DHJ+08] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, S. Teuchert. Clone detection in automotive model-based development. In *ICSE*. Pp. 603–612. 2008.

[DHJ+10] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, B. Schaetz. Model clone detection in practice. In *IWSC*. Pp. 57–64. 2010.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8(3):231–274, June 1987.

[Kos06] R. Koschke. Survey of Research on Software Clones. In *Dagstuhl Seminars*. 2006.

[PNN+09] N. Pham, H. Nguyen, T. Nguyen, J. Al-Kofahi, T. Nguyen. Complete and accurate clone detection in graph-based models. In *ICSE*. Pp. 276–286. 2009.

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

[SASC12] M. Stephan, M. Alafi, A. Stevenson, J. Cordy. Towards qualitative comparison of Simulink model clone detection approaches. In *IWSC*. Pp. 84–85. 2012.

[SS11] J. C. S. Grant, D. Martin, D. Skillicorn. Contextualized Semantic Analysis of Web Services. In *WSE 2011*. Pp. 33–42. 2011.

[Stö13] H. Störrle. Towards Clone Detection in UML Domain Models. *Software and Systems Modeling* 12(2):307–329, 2013.