Models are Code too: Near-miss Clone Detection for Simulink Models

Manar H. Alalfi James R. Cordy Thomas R. Dean Matthew Stephan Andrew Stevenson

School of Computing, Queen's University, Kingston, Canada
{*alalfi, cordy, dean, stephan, andrews*}*@cs.queensu.ca*

***Abstract*—While graph-based techniques show good results in finding exactly similar subgraphs in graphical models, they have great difficulty in finding near-miss matches. Text-based clone detectors, on the other hand, do very well with near-miss matching in source code. In this paper we introduce SIMONE, an adaptation of the mature text-based code clone detector NICAD to the efficient identification of structurally meaningful near-miss subsystem clones in graphical models. By transforming graph-based models to normalized text form, SIMONE extends NICAD to identify near-miss subsystem clones in Simulink models, uncovering important model similarities that are difficult to find in any other way.**

I. Introduction

Model clone detection refers to the process of identifying similar or identical fragments in higher-level software models based on some measure of similarity. While its counterpart, code clone detection, is a mature and established area of research [1], model clone detection is relatively new and has not been investigated as thoroughly. This is an issue for two reasons: first, model driven development is rapidly becoming a dominant method of software development, and second, the potential impact of identifying redundancy at higher levels is greater than at lower levels.

Not surprisingly, approaches to model clone detection to this point have primarily utilized graph-based techniques [2, 3, 4]. That is, they represent the models as nodes and edges and use variations of subgraph matching techniques to find clones. While natural and efficient for exact matching in visual models, these methods have had less success in near-miss clone detection [4]. In this paper, we propose a method for leveraging existing near-miss textual code analysis techniques, and in particular the hybrid syntactic approach of the NICAD [5] code clone detector, to detect near-miss model clones.

Our use case is the analysis and reengineering of thousands of Simulink models as part of the ongoing maintenance of a production automotive product line. The contributions of our approach are:

- Adaptation of an efficient, scalable text-based clone detector (NICAD) to detect near-miss clones in graphical models. Existing methods use graph matching.
- Efficient detection of not only type 1 (exact) and type 2 (renamed) model clones, but also type 3 (near-miss) model clones. Existing approaches handle types 1 and 2, but have difficulty with near-miss.
- Detection of model clones on three different levels of syntactic granularity: Simulink (entire) "model", (sub-) "system" and (detailed) "block". Existing approaches concentrate on the block level.
- Detection of structurally meaningful (near-miss) syntactic clones. Existing approaches use flattened subgraph matching [6] rather than syntactic structure, and use post-filtering to find structural units.

In this paper we demonstrate our results with examples of type 1, 2 and 3 clone identification in Simulink example models at the subsystem level of syntactic granularity. Near-miss clone detection allows us to find clones that are fragments of subsystems or blocks as well as entire subsystems.

The remainder of this paper is structured as follows. In Section II we extend the usual definitions for clone types to model clones and demonstrate them by example. Section III outlines the challenges and implementation of our method as a set of NICAD clone detector plugins, using Simulink automotive models as a running example. Section IV presents an evaluation of our text-based method by comparison with the state-of-the-art ConQAT graph-based method on a larger Simulink model set. Section V compares to other related work in model clone detection, and Section VI concludes and sets out our plans for future work. This paper is based in part on our preliminary position paper at IWSC 2012 [7], which it fleshes out to provide the details of our method, evaluation and comparison with related work.

II. Model Clone Types

We begin by defining what we mean by a model clone. Unlike source code, which is represented as linear text, models are typically represented visually, as box-and-arrow diagrams, and the clones we are searching for are similar subgraphs of these diagrams. Code clone detection techniques can be categorized according to the types of clones they can identify [1]. We adopt a similar categorization for model clones. In our first experiment we have identified three types of model clones at the Simulink subsystem level. We begin by demonstrating these types using examples from the Simulink demo set [1].

[1] http://www.mathworks.com/help/techdoc/ref/demo.html

978-1-4673-2312-3/12/$31.00

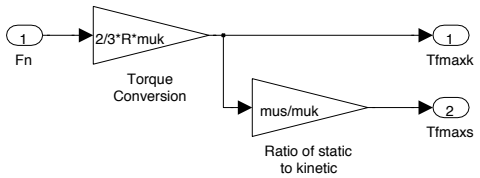

Figure 1. A type 1 (exact) model clone - the *Friction Mode* subsystem, which occurs in both the *Sldemo_Clutch* and the *Sldemo_Clutch_if* example models. NICAD similarity 100%.

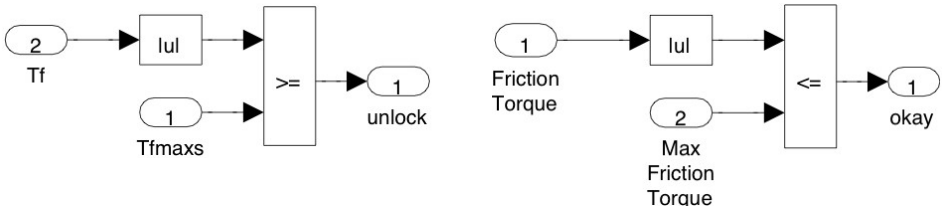

Figure 2. A type 2 (renamed) model clone - the *Required Friction for Lockup* subsystem (Left) and the *Break Apart Detection* subsystem (Right), both in the *Sldemo_Clutch_if* example model. NICAD similarity 85%.

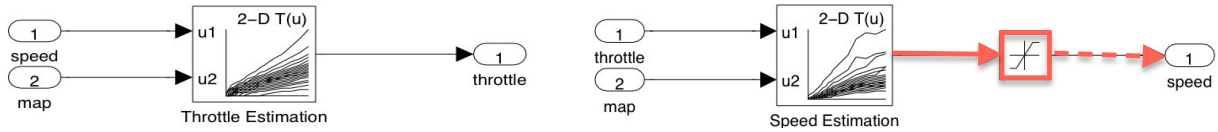

Figure 3. A type 3 (near-miss) model clone - the *Throttle.throttle_estimate* subsystem (Left) and *Speed.speed_estimate* subsystem (Right) of the *sldemo_fuelsys* model. NICAD similarity 76%.

1) Type 1 (exact) model clones: Identical model fragments except for variations in visual presentation, layout and formatting. Figure 1 shows an example from two different Simulink demo models, *Sldemo_Clutch* and *Sldemo_Clutch_if*, which include the identical subsystem *Friction Mode*.

2) Type 2 (renamed) model clones: Structurally identical model fragments except for variations in labels, values, types, visual presentation, layout and formatting. Figure 2 shows an example type 2 clone of two different subsystems, *Required Friction for Lockup* and *Break Apart Detection*, in the *Sldemo_Clutch* example model of the Simulink demo set.

3) Type 3 (near-miss) model clones: Model fragments with further modifications, such as changes in position or connection with respect to other model fragments and small additions or removals of blocks or lines in addition to variations in labels, values, types, visual presentation, layout and formatting. Figure 9 shows a type 3 clone between the *Throttle.throttle_estimate* and the *Speed.speed_ estimate* subsystems of the *sldemo_fuelsys* model of the Simulink demo set. A new block and line have been added, as well as naming and attribute changes to other blocks and lines.

Like code clones, model clones can cross structural and hierarchical levels. For example, the type 2 model subsystem clones of Figure 2 actually come from two different levels of abstraction in the *Sldemo_Clutch* model of the Simulink automotive examples set (Figure 4).

III. Approach

Our approach, called SIMONE (SIMulink clONE detector) extends NICAD [5], a language-sensitive code clone detection tool based on parsing and text-comparing syntactic fragments, to model clone detection. NICAD is explicitly designed to allow for unexpected differences in near-miss clones up to a given difference threshold. While NICAD is based on a plugin architecture that allows for new languages and normalizations, extending it to graphical models is a different kind of challenge that required both extensive transformations of the models and significant changes to NICAD itself.

This section outlines our plugins and changes to NICAD to extend it to support model clone detection for Simulink models. We use the automotive model set from the Simulink demo models as a running example to demonstrate the effect of each step. The Simulink automotive demonstration model set is appropriate both because the demonstration models are similar to automotive models used in industry, and because it contains several different versions of each example, yielding an unusually large number of clones to find. Our approach is designed to be used on the large scale production automotive models of our industrial partners at General Motors, which for proprietary reasons cannot be shown here.

There are several challenges to adapting a parser-based textual clone detector to find model clones. Models are typically represented visually as box-and-arrow-style diagrams, and the clones we are searching for are similar

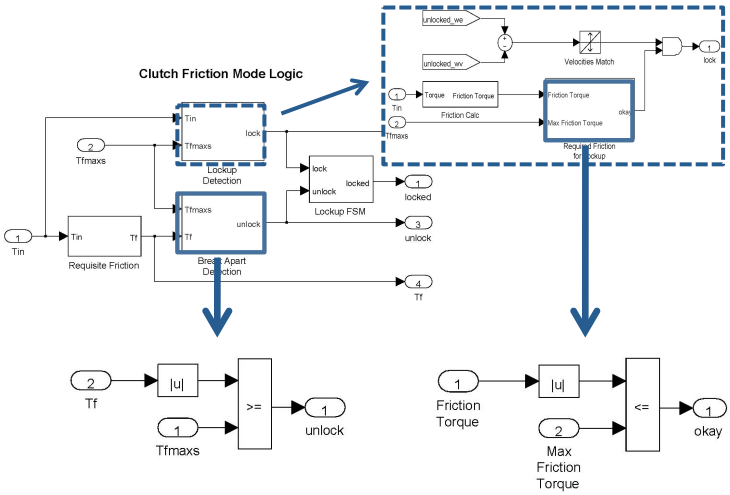

Figure 4. The type 2 (renamed) model clone of Figure 2, showing the multi-level context of the cloned subsystems.

```
...
System {
    Name                "onoff"
    Location            [168, 385, 668, 686]
    Open                on
    ModelBrowserVisibility  off
    ModelBrowserWidth   200
    ScreenColor         "automatic"
    PaperOrientationi   "landscape"
    PaperPositionMode   "auto"
    PaperType           "usletter"
    PaperUnits          "inches"
    ZoomFactor          "100"
    AutoZoom            on
    ReportName          "simulink-default.rpt"
    Block {
        BlockType       DiscretePulseGenerator
        Name            "Discrete Pulse\nGenerator"
        Position        [45, 25, 75, 55]
        Amplitude       "1"
        Period          "2"
        PulseWidth      "1"
        PhaseDelay      "0"
        SampleTime      "1"
    }
    Block {
        BlockType       Product
        Name            "Product"
        Ports           [2, 1, 0, 0, 0]
        Position        [145, 67, 175, 98]
        Inputs          "2"
        SaturateOnIntegerOverflow  on
    }
   ...
}
...
```

Figure 5. Example snippet of the textual representation used by Simulink to store graphical models. Note the large number of elements related to presentation and formatting, which have no meaning from the model cloning point of view.

subgraphs of the diagrams. Fortunately, for exchange reasons there are textual representations of these diagrams available, for example the XMI exchange format for UML diagrams. Simulink stores its models in a textual representation on disk, and we can use this representation (Figure 5) as the text input to SIMONE.

A. Simulink TXL Grammar

Because NICAD is a parser-based language-sensitive clone detector based on the TXL [8] parser, the first extension we needed was a TXL grammar to parse Simulink model files. Unfortunately there is no publicly available BNF grammar or published meta-model reference for the textual representation of Simulink models. Thus we used grammar inference techniques to derive a formal TXL grammar from a large set of example Simulink models in the public domain.

Using iterative inference to extend the grammar as more features were discovered as we expanded the example set, we were able to construct a TXL grammar that accounts for all characteristics of Simulink models and its higher-level Stateflow extensions (Figure 6). Our grammar identifies all Simulink constructs, including models, systems, blocks, lines, ports, branches and other fine-grained model components, and has been validated on the complete set of all Simulink models publicly available to us.

B. Extractor Plugin

Unlike language-insensitive textual clone detectors, NICAD is designed to identify structurally meaningful clones, that is, those that correspond to entire syntactic units such as a classes, methods, blocks or statements rather than arbitrary text sections. It works by parsing the source code to identify, extract and normalize all instances of such a unit, called the “potential clones”, and then compares them line-wise to identify near-miss clones using relaxed textual comparison.

Fortunately, many modeling languages, including Simulink, are hierarchical and have natural syntactic units to compare. In Simulink we can identify at least three levels of granularity: whole models, (sub-) systems, and blocks. We can think of these levels as roughly corresponding

```
%%%%%%%%%%%%%%%%%%%%
% Line subgrammar -
%    parses Simulink "Line" structures, e.g.
%
%  Line {
%         Labels          [1, 0]
%         SrcBlock        "Sine Wave"
%         SrcPort         1
%         Points          [30, 0; 0, -50]
%         DstBlock        "Product"
%         DstPort         2
%   }
%%%%%%%%%%%%%%%%%%%%

define line_list
   'Line '{                    [NL][IN]
       [repeat line_element]   [EX]
   '}                          [NL]
end define

% line elements of interest
define line_element
        [srcblock_element]
    |   [srcport_element]
    |   [dstblock_element]
    |   [dstport_element]
    |   [default_element]
end define
```

Figure 6. A small sample of the inferred TXL grammar for Simulink model files. The entire grammar consists of 90 productions in about 550 lines of TXL definitions.

to whole source files, functions, methods or classes, and (possibly compound) statements in other languages.

1) Model Granularity: At the model granularity, we treat entire Simulink models as the potential clones. Simulink models consist of (sub-) systems, which themselves are built up from blocks, lines and ports. At this level of granularity we can evaluate similarity of entire models, such as the *sldemo_clutch* and *sldemo_clutch_if* near-miss whole model clones in the Simulink demo set.

2) System Granularity: The Simulink "system" (i.e., subsystem) level of granularity is probably the most important level for model clone detection, exposing structurally meaningful clones of parts of models. Simulink subsystems are nested and hierarchical, and correspond roughly to functions, methods or classes in other languages, yielding clones of many sizes at many levels. We have identified several classes of both internal (within model) and external (cross-model) subsystem clones in the Simulink automotive examples. For example, the *Friction Model* subsystem in both the *sldemo_clutch* and *sldemo_clutch_if* models, shown in Figure 1, is an exact subsystem clone across those models; and the *Required Friction for Lockup* and *Break Apart Detection* subsystems of the *Sldemo_Clutch_if* example model, shown in Figure 2, are near-miss subsystem clones within a single model.

3) Block Granularity: Blocks are the finest-grained elements of Simulink models. However, blocks can also contain subsystems, which represent a group of blocks and lines that work together to provide a specific functionality.

While we have built "potential clone" extractor plugins for all three levels of Simulink granularity, we have concentrated on model clones at the (sub-) system level. Since in general Simulink whole models consist of a single system with subsystems, and since blocks below the level of systems are in general too small to be interesting for clone detection, we do not consider this to be a significant limitation. Subsystem granularity is also the case that is of primary interest to our industrial partners in the context of our work with them.

Due to differences in Simulink by comparison with traditional programming languages, our Simulink system extractor plugin has been developed in a different way than usual for NICAD. NICAD's plugin process order involves parsing and extraction of potential clones, optionally followed by renaming, filtering, abstraction and custom normalization. However, in the case of Simulink we found that filtering (removal of irrelevant parts) and custom normalization (sorting the order of Simulink blocks) must precede renaming (normalization of identifiers and literals), since they depend on original names and values. As a result, we could not use the standard NICAD plugin process order. To work around this limitation of NICAD, the SIMONE extractor plugin was developed to include the filtration and sorting normalization steps as part of extraction. They remain conceptually separate steps however, and if in future NICAD is enhanced to allow for specification of other process orders, then these steps can easily be separated into NICAD plugins.

We tested and validated our original subsystem extractor on the entire set of Simulink demo models provided by MathWorks, concentrating on the automotive example models in detail. Even without any modification of the original source text form of the Simulink models, our Simulink parser and subsystem extractor allowed many exact and exact near-miss subsystem clones in the automotive examples to be found (Table I). However, on closer examination most of them were uninteresting clones of presentation data, and only ten were of any interest. A great many other clones were not found, and clearly more work needed to be done.

C. Filtering

A quick investigation showed that one of the major problems was that the text form of Simulink model files is dominated by information irrelevant to the meaning of the models. Figure 5 shows an example of the large numbers of irrelevant elements relating to position, color, font, spacing, orientation, printing and other attributes that dominate Simulink models in their text representation. Even a few small changes in attributes such as color and font can make identical model subsystems look very different when compared in this original text form, and prevent NICAD from finding them as clones.

In order to remove these irrelevant differences from the comparison of subsystems, we designed a filtering plugin to identify and remove irrelevant elements and blocks from

Total nontrivial subsystems 357	Extractor only		Filtered		Filtered & Sorted		Filtered, Sorted & Renamed	
Clone Type	Type 1	Type 3-1 @30%	Type 1	Type 3-1 @30%	Type 1	Type 3-1 @30%	Type 2	Type 3-2 @30%
Clone Pairs	116 / 10*	364 / 164*	204	204	303	181	279	1938
Clone Classes	8 / 4*	57 / 56*	44	55	45	52	48	24
Clone Coverage	8% / 3%	52% / 46%	37%	48%	42%	45%	49%	75%

Table I
NUMBER AND SIZE OF SUBSYSTEM MODEL CLONES FOUND IN THE AUTOMOTIVE EXAMPLES OF THE SIMULINK DEMO MODEL SET

The Simulink demo automotive examples are a good test of model clone detection since they contain a great many variants of the same models. Even with raw extraction and no filtering, NICAD finds some of these exact model subsystem clones. However, without filtering the majority of these clones consist entirely of Simulink presentation information. When these are ignored, very few true clones are found (*). The derived clone types 3-1 and 3-2 refer to near-miss exact and near-miss type 2 clones respectively.

extracted subsystem potential clones. Due to the continued lack of definitive documentation for the form of Simulink model files mentioned above, we once again were faced with an inferential process, in which we gradually tuned our filters to remove irrelevant attributes as they were discovered. In the end, our filtering transformation removes more than 200 different kinds of attribute lists and elements to reduce the representation of model subsystems to their core elements representing the model's blocks, lines and ports themselves, unadorned by layout, formatting and presentation attributes.

Filtering significantly improved recall in finding exact and near-miss exact subsystem clones in the automotive example system (Table I). The total number of subsystem clones found was significantly increased, and virtually all of those detected were of interest. Removing layout attributes also allowed us to find larger subsystem clones, covering a much large proportion of the extracted subsystems. Hand validation of all clones found in the automotive example models showed that all were valid subsystem clones.

D. Sorting

While filtering had improved our subsystem clone detection a great deal, we found that there were still some subsystem clones we could identify by hand that were still not detected. The remaining problem was a fundamental one when using a linear text-based comparison method on graphical data: the order of blocks, lines and ports in the textual representation of a model subsystem does not change its graphical meaning. Figure 8 demonstrates this problem well, showing the original Simulink text representation of two subsystems of the Simulink automotive models, *Driver Switch* and *Passenger Switch*, that we later found to be clones. Applying NICAD to these two subsystems will report that they are different, even though they are not, since the order of the matching blocks in the text form is different.

Our solution to this problem was simple: we defined and implemented a cannonical sort of blocks, lines, ports and branches as a sort transformation on the filtered subsystem potential clones. Each subsystem in Simulink consists of a set of blocks, lines, ports, and branches, possibly containing deeper subsystems. In the text representation, the sequence of these model elements may not be the same, even in identical subsystems. Thus we developed a sorting plugin that sorts model elements according to the following criteria:

- Sort Blocks by *Type Name*
- Sort Lines by *Source Block*
- Sort Ports by *Port Name*
- Sort Branches by *Destination Block*

Figure 7 shows an example of a type 1 (exact) subsystem clone detected with the sorting plugin added. A clone class of 13 instances of this subsystem clone from different models in the automotive example set was identified, with a joint similarity of 98%. Before sorting, only some of these instances were found to be clones, while others were missed altogether. Sorting significantly increased the number of exact subsystem clones found, but allowed much larger near-miss clones to be identified, reducing the total number reported (Table I).

E. Renaming

With the sorting plugin resolving problems of linear representation, SIMONE was able to find all exact and near-miss exact subsystem clones in the automotive example model set using near-miss thresholds of 5% and 30% respectively. While the 30% threshold could find some type 2 (renamed) subsystem clones as well, to find all of them we would need to remove naming differences using a NICAD renaming plugin for Simulink.

Unfortunately, the generic renaming algorithm provided with NICAD to rename identifiers in other programming languages could not be used for Simulink. Unlike other languages, names in the text representation of Simulink

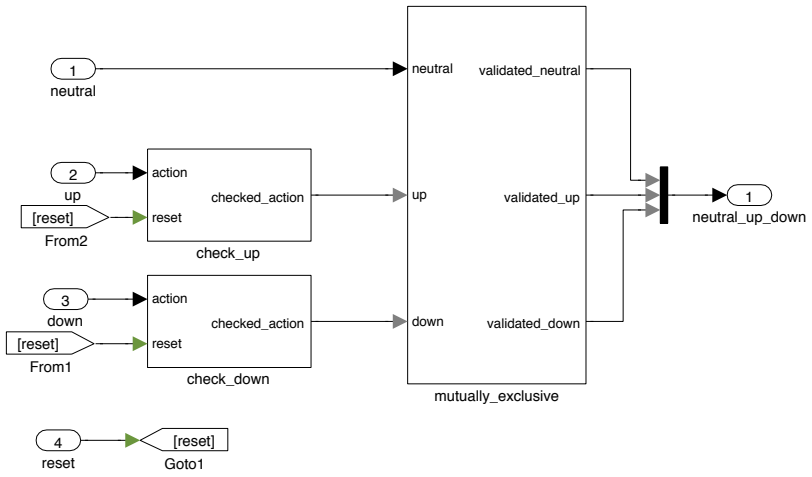

Figure 7. A type 1 (exact) model clone - the *Validate_driver* subsystem, which occurs in *powerwindow01* and five other Simulink automotive models in the Simulink demo set. NICAD similarity before sorting 77% (type 3), after sorting 98% (near exact).

```
518             Block \{\
519               BlockType               PMIOPort\
520               Name                    "Vin"\
521               SID                     "33"\
522               Position                [250, 35, 270, 55]\
523               BlockMirror             on\
524               Port                    "1"\
525               Side                    "Left"\
526             \}\
527             Block \{\
528               BlockType               Outport\
529               Name                    "active"\
530               SID                     "34"\
531               Position                [35, 38, 65, 52]\
532               BlockMirror             on\
533               IconDisplay             "Port number"\
534             \}\
535             Line \{\
536               SrcBlock                "ADC\\ndriver up"\
537               SrcPort                 1\
538               DstBlock                "active"\
539               DstPort                 1\
540             \}\
541             Line \{\
542               SrcBlock                "conditioning\\ndriver up"\
543               SrcPort                 1\
544               DstBlock                "ADC\\ndriver up"\
545               DstPort                 1\
546             \}\
547             Line \{\
548               LineType                "Connection"\
549               SrcBlock                "Vin"\
550               SrcPort                 RConn1\
551               DstBlock                "conditioning\\ndriver up"\
552               DstPort                 LConn1\
553             \}\
554           \}\
555         \}\
```

```
517             Block \{\
518               BlockType               PMIOPort\
519               Name                    "Vin"\
520               SID                     "112"\
521               Position                [250, 35, 270, 55]\
522               BlockMirror             on\
523               Port                    "1"\
524               Side                    "Left"\
525             \}\
526             Block \{\
527               BlockType               Outport\
528               Name                    "active"\
529               SID                     "113"\
530               Position                [35, 38, 65, 52]\
531               BlockMirror             on\
532               IconDisplay             "Port number"\
533             \}\
534             Line \{\
535               LineType                "Connection"\
536               SrcBlock                "Vin"\
537               SrcPort                 RConn1\
538               DstBlock                "conditioning\\ndriver up"\
539               DstPort                 LConn1\
540             \}\
541             Line \{\
542               SrcBlock                "conditioning\\ndriver up"\
543               SrcPort                 1\
544               DstBlock                "ADC\\ndriver up"\
545               DstPort                 1\
546             \}\
547             Line \{\
548               SrcBlock                "ADC\\ndriver up"\
549               SrcPort                 1\
550               DstBlock                "active"\
551               DstPort                 1\
552             \}\
553           \}\
554         \}\
```

Figure 8. An example of how unsorted model elements (such as Lines) can affect the precision and recall of NICAD on model clones.

models are represented as quoted strings, for example “ADC driver up” and “Vin” in Figure 8. Moreover, some names in the text representation of Simulink models, such as block and line types, should not be renamed when comparing for clones.

We therefore needed a much more sophisticated blind renaming plugin for Simulink than for other languages handled by NICAD. To implement type 2 renaming we used TXL agile parsing techniques to grammatically distinguish elements to be renamed from those that should not be, and installed this transformation as a renaming plugin for Simulink. The plugin anonymizes all names and values associated with elements and blocks, preserving only BlockType and LineType elements for comparison, allowing for detection of type 2 and near-miss type 2 (type 3-2) subsystem clones in Simulink models.

Figure 9 shows the effect of renaming in the detection of a near-miss type 2 model clone. Before renaming, the *Pressure.map_estimate* subsystem (A) and *Speed.speed_estimate* subsystem (B) were detected as near-miss clones, but only after renaming was a third clone, *Speed.speed_estimate* subsystem (C), added to the same clone class.

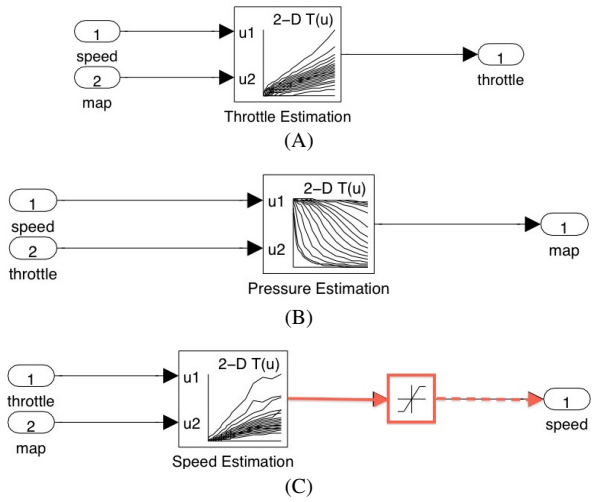

Figure 9. An example of the effect of renaming on NICAD detection of type 2 model clones. The *Pressure.map_estimate* subsystem (A) and *Speed.speed_estimate* subsystem (B) were initially detected without renaming. After renaming the new clone *Speed.speed_estimate* subsystem (C) of the *sldemo_fuelsys* model was recognized as another clone in the same class. NICAD similarity 74% (near-miss, type 3).

Renaming also increased the size of (now type 2) subsystem clones that could be found, reducing the total number but increasing their coverage (Table I). Using renaming allowed us to find over 1,900 near-miss type 2 (type 3-2) subsystem clones in the automotive models, covering 75% of the subsystems, consistent with the large number of different versions of models in the demonstration examples.

IV. Evaluation and Comparison with ConQAT

We have evaluated SIMONE on all of the publicly available Simulink models, including all of Matlab Central, and all of the demonstration systems distributed with Simulink. As a parser-based technique, precision is not an issue for the NICAD engine [9], and the real issue is recall, which we have addressed in Table I. Even so, precision was validated for all our test results by comparing to the original models by hand. Our tests included systems with models of over 100,000 source lines, which are parsed and processed in under a minute, and we continue to test larger scalability.

We compared SIMONE against ConQAT [2], a state-of-the-art graph-based approach to finding model clones in Simulink and other data-flow models. In the ConQAT approach, models are flattened into graphs consisting of their basic blocks and linear connections, normalized by assigning each block and line a value that can be used to compare them. Clone pairs are discovered by finding matching subgraphs, and clustered to form clone classes. Normalized values include the block type and some of the block attributes while excluding "semantically irrelevant" information, such as name and layout.

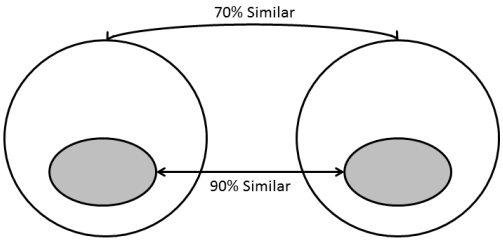

Figure 10. The nested clone problem. It is difficult to compare the results of a near-miss method with an exact method due to differences in similarity thresholds.

ConQAT compares models at the basic block level, ignoring subsystem boundaries, with a configurable parameter that defines the minimum number of matched blocks that need to be matched to be reported as a clone. This is in contrast with the syntactic method of SIMONE, which uses the natural Simulink structural boundaries for models, (sub-) systems, and blocks.

Using the definition of model clone types in Section II, ConQAT detects and identifies only type 2 (renamed) model clones. ConQAT detects but does not distinguish type 1 clones because of its renaming strategy.

To compare and contrast with ConQAT, we set SIMONE to use a NICAD 30% near-miss difference threshold. To mimic ConQAT's configuration we chose to use blind renaming to ignore differences in names and values while including block type. This set-up both allows us to detect and compare the set of type 2 clones discovered by SIMONE with those detected by ConQAT, and also to demonstrate SIMONE's detection of type 3 clones ConQAT may not be able to find.

Our comparison evaluation is primarily qualitative. The reason for this is that it is difficult to obtain any comparable numbers because of differences in the tools, specifically in the way they report nested clones, and because of larger block clusters. Figure 10 demonstrates the problem of nested clones. The outer circles represent a near-miss clone pair that has 70% similarity as reported by SIMONE. The inner circles represent a clone pair that is contained within the outer clone pair and is greater than 70% similar, for example 90% similar according to SIMONE, as shown in the diagram. If SIMONE's NICAD difference threshold is set to 30% then only the outer pair will be reported as a clone pair, and the inner clone pair will be contained in that result. Because the outer pair is not a type 2 clone, ConQAT would display only the inner pair, assuming it met ConQAT's type 2 criteria. Neither result is incorrect, it only depends on the specific type of result one is looking for. Because of these and other differences in output of the tools, we perform a qualitative evaluation rather than a quantitative one.

In order to compare the two approaches, we use the same

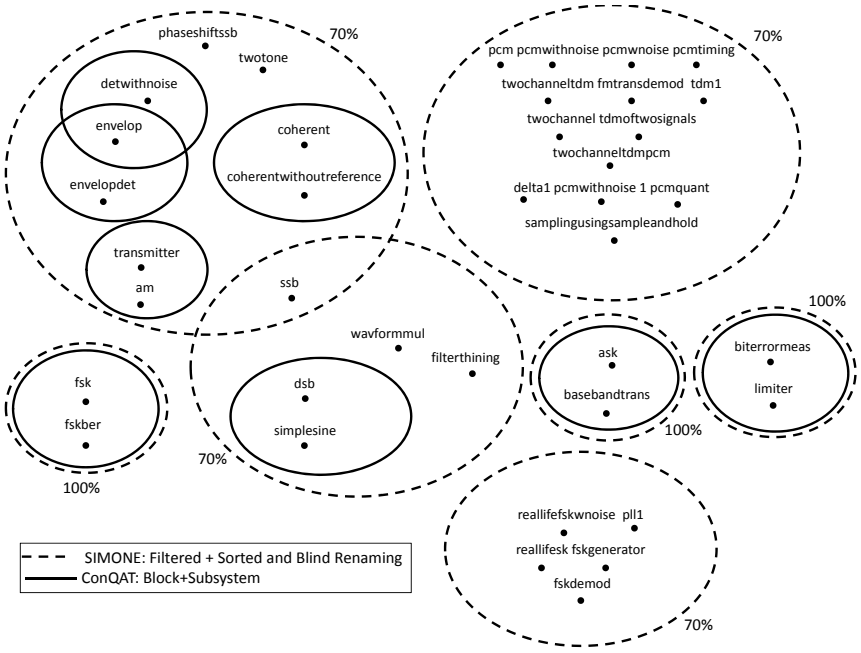

Figure 11. Simulink model subsystem clone classes discovered by SIMONE - - - and ConQat —- in the SIM project. The percentages are the minimum pairwise similarities reported by SIMONE for the clone class.

publicly available models from Matlab Central [2] that were used by the ConQAT authors previously [10].

We could not use the Simulink demo model set because it contains Stateflow extensions, which are not yet handled by ConQAT. Thus our direct comparison was restricted to three systems from Matlab Central (MPC, MUL, and SIM). In these systems our near-miss process finds all of the clones detected by ConQAT, including all the groups of blocks that ConQAT identifies at the block level. We have room to report only the results of one system, SIM, for which Figure 11 displays a Venn diagram of the Simulink subsystem level clone classes discovered by running both tools on the communications system project, SIM, from the Matlab Central model set. Circles outlined with dashes represent SIMONE near-miss clone classes discovered using filtering, sorting, and blind renaming, while the circles outlined with lines represent ConQAT clone classes discovered using the default ConQAT configuration. The percentages next to the circles represent the NICAD minimum pairwise similarity among all the subsystems in the clone class.

In this specific project, we can see that SIMONE was able to find all of the clones that ConQAT did, although in some instances, the corresponding ConQAT classes were embedded in larger near-miss SIMONE classes. Also of note are the near-miss (70%) clone classes identified by SIMONE only. Figure 12 shows an example of a near-miss clone pair discovered by SIMONE in the SIM system. The subsystems are relatively similar, differing only by the summation block (little circle with two + signs) which splits the subsystem into two halves in the *pcmwithnoise* subsystem. ConQAT and other graph based approaches will not detect this clone because the summation block creates a cut vertex that partitions the graph into two smaller sub-clones. These sub-clones may, individually, be beneath the size threshold for detection even if the near-miss clone as a whole is above the threshold, as is the case for this clone, which is not reported at all using ConQAT's default settings. By comparison, SIMONE reports these subsystems as near-miss clones with 72% similarity.

Overall, we found that SIMONE and ConQAT find many of the same subsystem clones in Simulink models. However, in each case there are some clones that one finds that the other can not. Because ConQAT's graph-matching technique flattens Simulink models to ignore hierarchical structure and does not enforce detection of entire subsystems, ConQAT is able to find cloned clusters of blocks that do not form whole subsystems. By contrast, SIMONE is structure sensitive, and is designed to find only complete structural clones. Thus, only near-miss or exact clone pairs representing entire subsystems will be identified, and smaller nested cloned clusters of blocks will be reported only if they are part of a clone of a whole subsystem.

On the other hand, because ConQAT is not designed to implement near-miss clone detection, SIMONE finds many more challenging Type-3 clones that ConQAT does not. We are currently automating the comparison process and trying to resolve the parsing problems of ConQAT so we can report

[2]http://www.mathworks.com/matlabcentral/

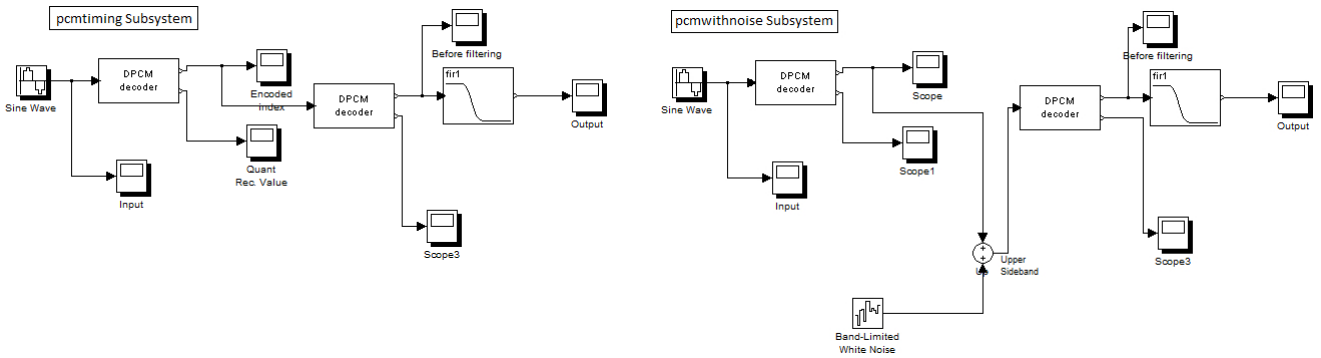

Figure 12. Example of a type 3 (near-miss) subsystem clone pair discovered by SIMONE that will not be found by ConQAT. The introduction of the summation block (labelled + +) in the subsystem on the right creates a cut vertex that partitions the graph into two exact sub-clones each too small to be of interest.

on all of the models available to us [11].

In theory, both SIMONE and ConQAT can each be adapted to approximate what the other does: ConQAT could be extended to allow for configuration of a similarity threshold to detect near-miss graph clones, and SIMONE could use a source transformation to flatten the structure, filter out the lines, and do block-level evaluation. However, there will still be some kinds of unexpected near-miss changes that ConQAT may never detect, and extending SIMONE to ignore structure is fundamentally contrary to its intended purpose.

V. Related Work

In addition to ConQAT, there are two other approaches that employ graph-based techniques for Simulink model clone detection. Pham et al. [4] developed ModelCD, including the eScan and aScan algorithms to detect exact- and near-miss- clones, respectively. ModelCD attempts to improve on ConQAT by utilizing graph mining techniques and Simulink-specific properties rather than relying on subgraph matching heuristics alone. We chose not to include it in our comparison because ModelCD operates in roughly the same way as ConQAT and, as the ConQAT authors have demonstrated [10], the improvements in ModelCD have little impact on results, and do not scale well. ModelCD is also not publicly available.

Peterson [3] has developed the Naive Clone Detector to detect exact Simulink clones. Like ModelCD, it uses graph-based modeling and Simulink information, but by contrast, it employs a top-down approach. We did not include Naive Clone Detector in our evaluation as it also is similar to ConQAT, and the author was unable to release it to us.

Al-Batran et al. [6] 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 Simulink model representations.

In theory graph-based algorithms such as ConQAT may be able to detect the model clones we present in Figure 12 if the parameters are relaxed sufficiently. However, this would result in large numbers of false positives, yielding very poor precision and making the result meaningless. By contrast, SIMONE efficiently detects such clones at a 30% near-miss threshold with no false positives.

In previous work, we surveyed the entire area of code-clone detection [1]. NICAD was chosen as the code-clone technique to adapt as the basis of SIMONE 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 [12], 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 [13], which performs similarity comparison on structural system models. We leave clone detection in that area as future work, as we are currently interested in Simulink behavioral models only.

Gold et al. [14] have identified new clone types for data flow languages similar to our proposed extension of code clone types to models. Gold's data flow clone types include: 1) exactly-copied code fragments; 2) exactly-copied code fragments with layout and comment changes; 3) exactly-copied code fragments with layout, comment, and literal values changes; and 4) code fragments with modifications. Since Simulink is a data-flow modeling language, we could adapt to these slightly different definitions by modifying NICAD appropriately.

VI. Conclusions and Future Work

In this paper we have introduced a method for using text-based code clone detectors to find near-miss clones in graphical models, and demonstrated SIMONE, an implementation for Simulink models based on the NICAD clone detector. We outlined the challenges of using a parser- and text-based method on graphical models, and described our solutions using filtering and sorting of the text representation. Finally, we compared the results obtained with our new method with a state-of-the-art graph-based method and showed that our near-miss detection can find meaningful clones that graph-based methods can miss. Our approach generalizes to other modeling languages, such as UML-based ones, with only customization of the filtering and sorting algorithms.

While our method has been tested and hand-validated on a large set of publicly available Simulink models, it has yet to be used on industrial examples, and we look forward to working with our industrial partners at General Motors to analyze their systems. Although we inherit the proven textual clone accuracy and scalability of NICAD [9], it is not clear that all of these attributes transfer to its use on graphical models, and we are currently running a larger scale experiment to statistically validate and compare our method on a much larger set of models. We are also working on extending our method with a new plugin to detect consistently renamed model clones, those whose labels and values are renamed in a consistent fashion rather than arbitrarily.

A practical issue for model clone detection is the graphical presentation of results. While we are eventually planning to use the ConQAT Eclipse plugin designed for this task, at the moment we generate Simulink colorization scripts to show SIMONE results directly in the Simulink IDE.

Acknowledgements

This work is supported by NSERC, the Natural Sciences and Engineering Research Council of Canada, as part of the NECSIS Automotive Partnership with General Motors, IBM Canada and Malina Software Corp. We gratefully acknowledge the assistance of Benjamin Hummel of the Technical University of Munich in helping us to understand and run ConQAT, and the inspiration of Mark Harman's SCAM 2010 keynote address [15], in which he observed that "models are source code too".

References

[1] 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.*, vol. 74, no. 7, pp. 470–495, 2009.

[2] F. Deissenboeck, B. Hummel, E. Jurgens, B. Schatz, S. Wagner, J. F. Girard, and S. Teuchert, "Clone detection in automotive model-based development," in *30th Int. Conf. on Softw. Eng.*, 2009, pp. 603–612.

[3] H. Petersen, "Clone detection in Matlab Simulink models," Master's thesis, Tech. Univ. Denmark, 2012.

[4] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Complete and accurate clone detection in graph-based models," in *31st Int. Conf. on Softw. Eng.*, 2009, pp. 276–286.

[5] 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.*, 2008, pp. 172–181.

[6] B. Al-Batran, B. Schätz, and B. Hummel, "Semantic clone detection for model-based development of embedded systems," *Model Driven Eng. Languages and Syst.*, vol. 6981, pp. 258–272, 2011.

[7] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson, "Near-miss model clone detection for Simulink models," in *6th Int. Works. on Softw. Clones*, 2012, pp. 78–79.

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

[9] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *4th Int. Works. on Mutation Analysis*, 2009, pp. 157–166.

[10] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz, "Model clone detection in practice," in *4th Int. Works. on Softw. Clones*, 2010, pp. 57–64.

[11] M. Stephan, M. H. Alafi, A. Stevenson, and J. R. Cordy, "Comparison of model clone detection approaches," in *6th Int. Works. on Softw. Clones*, 2012, pp. 84–85.

[12] M. Stephan and J. R. Cordy, "A survey of methods and applications of model comparison," Queen's Univ., Tech. Rep. 2011-582 Rev. 2, 2011.

[13] C. Brun and A. Pierantonio, "Model differences in the Eclipse modelling framework," *The European Journal for the Informatics Professional*, pp. 29–34, 2008.

[14] N. Gold, J. Krinke, M. Harman, and D. Binkley, "Issues in clone classification for dataflow languages," in *4th Int. Works. on Softw. Clones*, 2010, pp. 83–84.

[15] M. Harman, "Why source code analysis and manipulation will always be important," in *10th Int. Conf. on Source Code Analysis and Manip.*, 2010, pp. 7–19.