Variability Identification and Representation for Automotive Simulink Models

Manar H. Alalfi, Eric J. Rapos,
Andrew Stevenson, Matthew Stephan,
Thomas R. Dean, and James R. Cordy

Abstract This paper presents an automated framework for identifying and representing different types of variability in Simulink models. The framework is based on the observed variants found in similar subsystem patterns inferred using Simone, a model clone detection tool, and an empirically derived set of variability operators for Simulink models. We demonstrate the application of these operators to six example systems, including automotive systems, using two alternative variation analysis techniques, one text-based and one graph-based, and show how we can represent the variation in each of the similar subsystem patterns as a single subsystem template directly in the Simulink environment. The product of our framework is a single consolidated subsystem model capable of expressing the observed variability across all instances of each inferred pattern. The process of pattern inference and variability analysis is largely automated and can be easily applied to other collections of Simulink models. We provide tool support for the variability identification and representation using the graph-based approach.

Manar H. Alalfi
Computer Science, Ryerson University, Canada, e-mail: manar.alalfi@ryerson.ca

Eric J. Rapos
Department of Computer Science and Software Engineering, Miami University, USA
e-mail: rapose@miamioh.edu

Andrew Stevenson
School of Computing, Queen's University, Canada,
e-mail: andrews@cs.queensu.ca

Matthew Stephan
Department of Computer Science and Software Engineering, Miami University, USA
e-mail: stephamd@miamioh.edu

Thomas R. Dean
Electrical and Computer Engineering, Queen's University, Canada
e-mail: dean@cs.queensu.ca

James R. Cordy
School of Computing, Queen's University, Canada, e-mail: cordy@cs.queensu.ca

1 Introduction

Software variability management (SVM) research has gained a lot of interest in the last two decades, especially for its vital role in developing reusable software product line (SPL) assets [5]. SVM is a complex, multifaceted problem that intersects with several traditional software engineering topics, including software configuration management, run-time dynamism, domain specific languages, model-driven engineering, and software architecture. SVM offers a powerful toolbox to help manage complexity in these fields and is rapidly evolving into an independent research area that is of vital importance for systems that include configuration and run-time dynamism of components, in addition to software product lines.

One facet of SVM is variability modelling, an enabling technology for delivering a variety of related software systems in a fast, consistent and comprehensive way. The key is to build a common base from which to efficiently express and manage variations. SVM is often closely associated with SPLs, which are mainly aimed at creating and maintaining a collection of similar software systems derived from a shared set of software assets. Variability can be expressed as stand-alone models, such as feature models in SPLs, or as annotations on a base model, by means of extensions to the base modelling language, such as UML profiles with stereotypes [15].

Variability modelling continues to gain interest from industry, and variability support in modelling tools, including Mathworks' Simulink and IBM's Rhapsody, is one of their most desirable features. Several industrial standards, such as SysUML and AUTOSAR, are actively working to create extensions that help to express variability.

Understanding variability in existing systems and the variation points of their artifacts is the first and most important step towards enabling variability modelling. Many methods have been proposed for analyzing commonality and variability from a requirements point of view, as well as connecting the analysis to the implementation [11, 21]. However, there remains a need for techniques that analyze existing system requirements and implementations for commonality and variability in an automated way.

In this chapter we present a framework for identifying variability candidates from existing software intensive systems modelled using Simulink [26], the most popular modelling languages for hybrid hardware/software systems. Automotive Simulink models are particularly prone to cloning due to the copy-paste authoring paradigm of the Simulink IDE, and the inherent similarity of elements and tasks in automotive applications. Our framework, shown in Figure 1, uses an efficient model clone detection technique to automatically identify subsystem variants from a large pool of existing Simulink models. Once all potential variants are identified, the framework classifies and represents those variants using a set of empirically-derived variability operators.

The framework is aimed at providing tool support to automatically represent model subsystem variability directly in the Simulink environment, and thus provide practical assistance to engineers to identify, understand, and visualize patterns

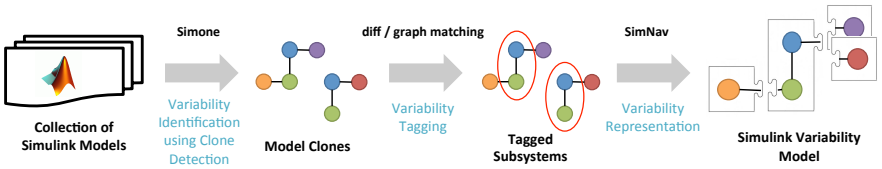


Fig. 1 Variability Identification and Representation Framework

of similar subsystems in a large model set. This understanding may help, among other things, in reducing maintenance effort and bug identification at an early stage of software development, both on the model level and before the model semantics are transformed into actual code. We demonstrate our framework on six systems from the Mathworks demonstration set and describe the stages of our framework using a running example.

In a previous short paper, we proposed a set of empirically-derived variability operators for Simulink models [2], and provided evidence of the soundness of our operators based on the analysis of six Simulink systems representing a range of diverse applications. In this chapter, we use those proposed variability operators as the basis of an automated framework for the identification and representation of system variability in Simulink models.
The contributions of this chapter are as follows:

- Detailed description of an automated framework for the identification and representation of variability in Simulink models.
- Demonstration of a text-based tagging approach to identify and mark variability in Simulink models using our previously proposed set of variability operators.
- Demonstration of a graph-based approach for the identification and representation of variability in Simulink models.
- Tool support for the graph-based approach that works directly in the Simulink IDE environment.

The following sections provide a detailed description of the proposed framework and our experience with it.

2 Variability Identification and Representation Framework

Figure 1 presents the stages of framework.

Stage 1: Variability Identification. This stage uses model clone detection to identify groups of similar subsystems in a repository of Simulink models. We used Simone [1], a hybrid text_based model clone detection tool, to identify common Simulink subsystem patterns and variability candidates. In this framework we have configured Simone to identify subsystems that are at least 80% similar to each other as a first approximation. Similar subsystems are clustered

```
System {
  Name "add"
    Block {
      BlockType Inport
      Name "In1"
      ZOrder 1
      IconDisplay "Port number"
    }
    Block {
      BlockType Inport
      Name "In2"
      ZOrder 2
      Port "2"
      IconDisplay "Port number"
    }
    Block {
      BlockType Sum
      Name "Add"
      Ports [2, 1]
      ZOrder 3
      InputSameDT off
      OutDataTypeStr "Inherit: Inherit via internal rule"
      SaturateOnIntegerOverflow off
    }
    Block {
      BlockType Outport
      Name "Out1"
      ZOrder 4
      IconDisplay "Port number"
    }
    Line {
      SrcBlock "In1"
      SrcPort 1
      DstBlock "Add"
      DstPort 1
    }
    Line {
      SrcBlock "In2"
      SrcPort 1
      DstBlock "Add"
      DstPort 2
    }
    Line {
      SrcBlock "Add"
      SrcPort 1
      DstBlock "Out1"
      DstPort 1
    }
}
```

(a) Variant A - Addition

```
System {
  Name "subtract"
    Block {
      BlockType Inport
      Name "In1"
      ZOrder 1
      IconDisplay "Port number"
    }
    Block {
      BlockType Inport
      Name "In2"
      ZOrder 2
      Port "2"
      IconDisplay "Port number"
    }
    Block {
      BlockType Sum
      Name "Subtract"
      Ports [2, 1]
      ZOrder 5
      Inputs "+-"
      InputSameDT off
      OutDataTypeStr "Inherit: Inherit via internal rule"
      SaturateOnIntegerOverflow off
    }
    Block {
      BlockType Outport
      Name "Out1"
      ZOrder 4
      IconDisplay "Port number"
    }
    Line {
      SrcBlock "In1"
      SrcPort 1
      DstBlock "Subtract"
      DstPort 1
    }
    Line {
      SrcBlock "In2"
      SrcPort 1
      DstBlock "Subtract"
      DstPort 2
    }
    Line {
      SrcBlock "Subtract"
      SrcPort 1
      DstBlock "Out1"
      DstPort 1
    }
}
```

(b) Variant B - Subtraction

Fig. 2 Textual Representations of example subsystem variants A & B

into clone classes, sets of subsystems that are similar to one another. Figure 2 shows an example of two instances of the same subsystem pattern as identified by Simone. Variant A takes two numbers and adds them together, while Variant B subtracts one number from the other - both models have two inports and one outport. Section 3 provides a more detailed discussion of this first stage.

Stage 2: Variability Tagging. This stage uses two techniques, #ifdef preprocessor text tagging and subgraph similarity algorithms, to identify and tag subsystem variability between the subsystems in each clone class identified in Stage 1. This identification and tagging process is based on a set of proposed variability operators that we have empirically inferred from a large set of observations of variance in pattern candidates identified by Simone. Section 4 presents our proposed variability operators for Simulink. The example in Figure 2 shows portions of the model that are similar among all instances (highlighted in green), while the elements in different colours represent the variation between the two models. This stage is aimed at automating the identification of commonality and variability in this way, and at marking the type of variability according to our variability operators. A detailed description of this stage is presented in Section 5.

Stage 3: Variability Representation. This stage presents our approach to representing the identified Simulink subsystem variability using Simulink's built-in Variant Subsystem Block capability. Referring to our running example in Figure 2, portions of the model that are similar among all instances (highlighted in

Fig. 3 Textual Representation of the Variance Model for variants A and B of Figure 2

green) are placed directly into the new variance model, with some renaming for sources and destinations pertaining to Variant Subsystem Blocks. The variability (for example, the "Add" block of Variant A (highlighted in pink), and the "Subtract" block of Variant B (highlighted in cyan)) must be encapsulated in Variant Subsystem Blocks for the corresponding variability operators, with additional source lines to implement the variants. Figure 3 presents the textual representation of the created variant model representing both Variant A and Variant B. The sections highlighted in green are the common elements from Variant A and Variant B, the pink highlighting represents the "Add" block from Variant A, and the cyan highlighting represents the "Subtract" block from Variant B. All of the other necessary information to construct the textual representation of the variance model is available from the text of Variants A and B. Section 6 discusses variability representation in Simulink based on our proposed operators.

System Name	# Subsystems	# Clone Pairs	# Clone Classes
Automotive	357	189	24
Aerospace	188	62	15
Industrial	16	4	2
Features	935	85	25
General	146	11	7
Others	28	6	4

Table 1: Simone Clone detection results at a difference threshold of 20%

We believe that automating the application of variability operators will streamline the process of representing subsystem variability in Simulink Models, and reduce the risk of error introduced through manual representation. In the following sections we discuss the stages of our approach in more detail, beginning with a discussion of the automated similarity identification inferred by Simone. We then introduce two options for tagging variation in the identified sets of similar subsystems, one based on *diff* and *#ifdef* on the normalized textual representation of the models, and one using graph matching algorithms to determine identical sub-graphs. Finally, we discuss the final step of translating the tagged variations to Simulink Variant Subsystem Blocks, completing an end-to-end automated process for the identification and representation of subsystem variability in Simulink models.

3 Variability Identification

To determine an appropriate set of Simulink subsystem variability operators, we used the set of models in the six diverse Simulink systems of the Mathworks Simulink demonstration set as a starting point. These systems include models for a range of applications in industrial, automotive, aerospace and other domains, and are intended to demonstrate the range of ways to represent model features in these applications using Simulink. They include a range of model versions and variants for each application, and represent a rich source of examples of Simulink model variation.

3.1 Simone: An Initial Approximation

To begin, we first required some indication of which subsystems in the models of each system were similar enough to be considered variants of each other. For this we used the Simone sub-model clone detector [1]. Simone is a hybrid model clone detection technique that uses a normalized text representation of graphical models to efficiently identify near-miss subsystem clones, that is, those that are similar up to a given threshold of difference (in this experiment, up to 20% different). Simone is based on the NICAD code clone detector [17], extended to handle graphical models.

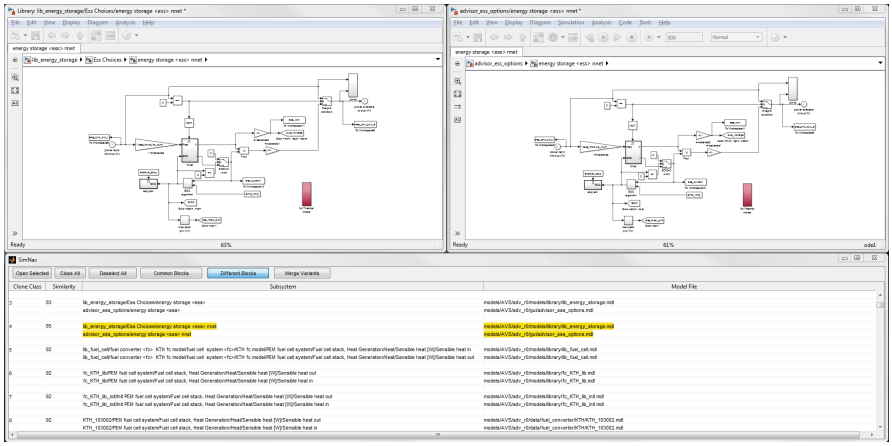


Fig. 4 The SimNav subsystem clone class exploration interface in the Simulink IDE (from [8])

To identify and categorize subsystem variations, we applied Simone to the entire set of models in each of the six Simulink demonstration systems. From each set of models, Simone generated a database of near-miss subsystem clone pairs, representing pairs of model subsystems which are largely similar by may vary up to 20% in components, connections, inputs, outputs or other attributes. Simone automatically groups these clone pairs into "clone classes", which are sets of subsystems that are nearly similar to one another. It inherits from NICAD an efficient exemplar-based algorithm to achieve this clustering, choosing a particular cloned subsystem and then gathering all those other cloned subsystems that are similar to it within the difference threshold. By beginning with the largest exemplars, it automatically identifies the most inclusive set of variants of each cloned subsystem.

In practice even the raw clone classes resulting from this analysis can already be used by Simulink model engineers to understand variations in their systems directly from the examples in each class. In our previous work we have integrated the results of Simone directly into the Simulink IDE using a Simulink plugin called SimNav [8, 16], that directly presents similar subsystems in the Simulink model editor (Figure 4).

Table 1 presents the initial clustering results provided by Simone for the set of models in each of the six Simulink demonstration systems. Each subsystem in each clone class has at least 80% common elements with the others in the class. A particular element of each clone class is chosen by our framework as an exemplar, from which the others in the class are considered to be variants. We then classified the nature of these variants to empirically derive the variability operators presented in the next section.

Fig. 5 Block Variability

4 Variability Operators

Using a manual inspection of the Simone results for the six systems using SimNav, and investigating the variants in each Simone-reported clone class, we identified the following types of variability in similar Simulink subsystems:

Block Variability Changes at the block level, such as added or removed blocks, or one block replaced with another. An example of this type of variability is shown in Figure 5 (circled in red).

Input/Output Variability Changes in the input/output ports for a specific block. These can be changes to the number of ports, or the signatures of the ports. A changed signal falls into this category as well. This type of variability is shown in Figure 6 (circled in red).

Function Variability Changes to the contained function of a specific block or set of blocks, such as constant values, data parameters, or the entire function. This type of variability is shown in Figure 7 (note the different functions and constants in the corresponding blocks of the two subsystems).

Layout Variability Changes to the layout information of the model elements, such as block position. This type of variability is shown in Figure 8 (note the mirroring of parts of the model).

Subsystem Name Variability Changes to the names of similar subsystems. This type of variability is shown in Figure 9 (circled in red).

For each of the six systems, we determined the number of instances of each type of variability, and ensured that all observed variations could be covered by the set of variability operators. The results of this categorization can be found in Table

Fig. 6 Input/Output Variability

System Name	Block	Input/Output	Function	Layout	Subsystem Name
Automotive	10	6	1	3	8
Aerospace	5	17	2	4	13
Industrial	5	2	0	0	0
Features	22	22	17	2	4
General	5	3	1	1	1
Others	14	24	4	3	5

Table 2: Observed Instances of the Identified Variability Operators

2. The most common types of variability we observed were Block Variability and Input/Output Variability, with the others occurring less frequently. There were no instances of variability that did not fall into one of these five categories.

5 Tagging Subsystem Variability

To model the variability across the instances of a given subsystem pattern, we must first determine the common components of the subsystem across all of the instances

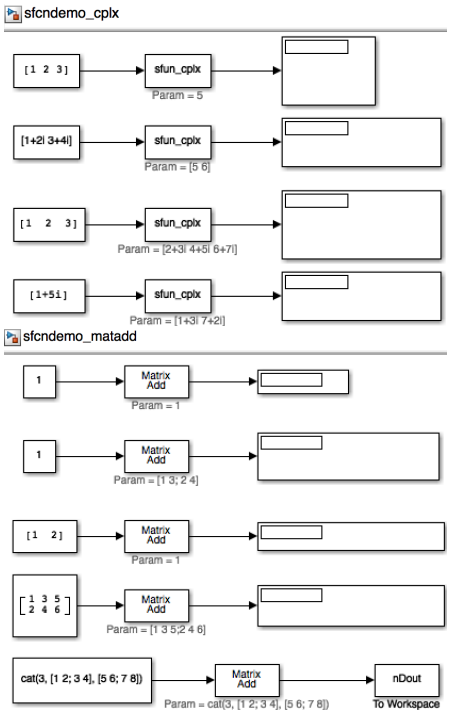


Fig. 7 Function Variability

in the clone class. Once we determine the commonalities between all instances, the remaining components of the subsystem represent the variations we wish to model using the variability operators.

5.1 Tagging Using #ifdef

Since Simone computes the clone classes based on a normalized textual difference between the subsystems, one straightforward way to tag the variability between models is in their textual representation. In this section we describe how to use the unix *diff* command and source transformation to tag the variation in similar subsystems.

We begin with a single difference file generated using the command *diff -DFIRST model1.mdl model2.mdl*. This command merges the two files using C-style *#ifdef* statements to characterize the differences. For example, Figure 10 shows two *Outport* blocks from two models taken from the Simulink example model set. Figure 11 shows the difference between the two outport blocks.

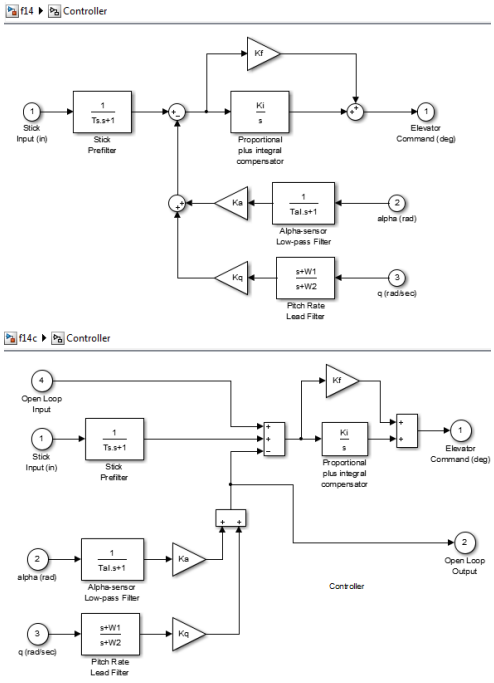


Fig. 8 Layout Variability

In this example, the differences between the corresponding outport blocks are three different attributes: *Name*, *SID*, and *Position*. The *Name* attribute identifies the name of the block, while the *Position* attribute is part of the layout of the model. The *SID* attribute is the unique identifier given to each element of a Simulink model. We manually split the difference into three differences, and append a tag to each difference condition to indicate which of the variability operators of Section 4 applies.

The resulting difference file is shown in Figure 12. The first difference uses the condition *FIRST_Name* to indicate that this is a subsystem name variability. The third difference appends the variability tag *_Layout* to indicate a layout difference. The second difference is a difference internal to the representation of the model, and we use the variability tag *_Other* for this case.

In general, the variability can be categorized based solely on the entity attributes involved in the differences. In the Simulink demonstration systems, the attributes that reflect the names of entities are the *Name* and *Text* attributes. Some of the attributes associated with Layout are *Position*, *Location*, *ZoomFactor*, *Points*, *FontName* and *FontSize*. The value attributes are much more varied as they specify the options for each of the function blocks in the models. Some examples of value attributes in the demonstration models are *Value*, *DataFormat*, *TimeRange*, *YMin* and *YMax*.

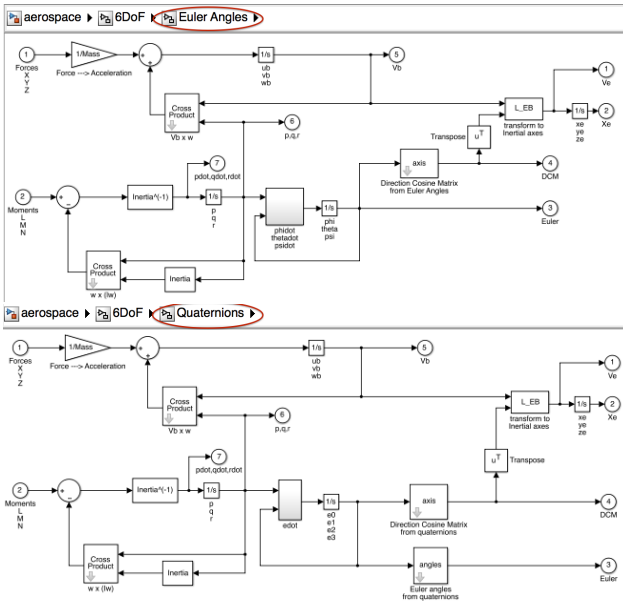


Fig. 9 Subsystem Name Variability

```
File m_SimulinkDemoModels_aerospace_0.3__2_171_so.mdl:
   Block {
      BlockType Outport
      Name "alpha, rad"
      SID "71"
      Position [675, 357, 705, 373]
      IconDisplay "Port number"
      InitialOutput "0"
   }

File m_SimulinkDemoModels_aerospace_0.3__2_123_so.mdl:
   Block {
      BlockType Outport
      Name "alpha (rad)"
      SID "60"
      Position [630, 235, 650, 255]
      IconDisplay "Port number"
      InitialOutput "0"
   }
```

Fig. 10 Example Subsystem Difference

Variability in structure appears in the output of diff in two different ways. First, they may appear as differences in attributes that express the connectivity between elements of the model. These are attributes such as *DstBlock*, *SrcBlock*, *DstPort*,*SrcPort*, *PortNumber* and *Port*. The second way these differences can appear is as additions and deletions of structural elements such as blocks and lines. Figure 13 shows the diffs resulting from the addition of a block in the one of the models, *m_SimulinkDemo Models_aerospace_0.3__2_123_so.mdl*. The *diff* algorithm trig-

```
Block {
   BlockType Outport
#ifndef FIRST
   Name "alpha (rad)"
   SID "60"
   Position [630, 235, 650, 255]
#else /* FIRST */
   Name "alpha, rad"
   SID "71"
   Position [675, 357, 705, 373]
#endif /* FIRST */
   IconDisplay "Port number"
   InitialOutput "0"
}
```

Fig. 11 Example Subsystem Difference

```
  Block {
   BlockType Outport
#ifndef FIRST_Name
   Name "alpha (rad)"
#else /* FIRST */
   Name "alpha, rad"
#endif /* FIRST */
#ifndef FIRST_Other
   SID "60"
#else /* FIRST */
   SID "71"
#endif /* FIRST */
#ifndef FIRST_Layout
   Position [630, 235, 650, 255]
#else /* FIRST */
   Position [675, 357, 705, 373]
#endif /* FIRST */
   IconDisplay "Port number"
   InitialOutput "0"
  }
```

Fig. 12 Example Subsystem Difference Split Into Tags

gers on the first difference, and since the element following an additional block is often another block, the first line of that block (i.e. *Block {*) often matches and appears outside of the difference at the beginning and inside the difference at the end (Figure 13). This is easily handled by moving the difference markers slightly earlier as shown in Figure 14. We use *_Structure* when tagging both structure attribute differences and added/deleted structural elements, also shown in Figure 14.

The differences can interact in interesting ways, but they can always be broken down into either additional elements or changes to attributes. Thus complex differences such as the diffs shown in Figure 15 can be separated into multiple diffs, as shown in Figure 16. This one difference encompasses all of a name change for an annotation, layout changes for the annotation, and an additional annotation.

These transformations, which we explored manually, can be implemented in a straightforward manner as a source transformation in TXL [7], following the approach taken by Malton et al. [14]. Malton et al. used a trace based approach to

```
  Block {
#ifndef FIRST
    BlockType Gain
    Name "Gain5"
    SID "42"
    Position [530, 222, 580, 268]
    ShowName off
    Gain "1/Uo"
   }
   Block {
#endif /* ! FIRST */
```

Fig. 13 Example Additional Block Difference

```
#ifndef FIRST_Structure
  Block {
    BlockType Gain
    Name "Gain5"
    SID "42"
    Position [530, 222, 580, 268]
    ShowName off
    Gain "1/Uo"
 }
#endif /* ! FIRST */
  Block {
```

Fig. 14 Example Transformed Additional Block Difference

```
  Annotation {
#ifndef FIRST
    Name "F-14 Flight Control(an updated
       version of this demo is available
       by running 'sldemo_f14')"
    Position [328, 377]
#else /* FIRST */
    Name "F-14 Longitudinal Flight Control"
    Position [368, 17]
    FontName "Arial"
    FontSize 18
    FontWeight "bold"
  }
   Annotation {
  Name "This demonstration models a flight
       control algorithm for the
       longitudinal motion of a Grumman
       Aer" "ospace F-14."
  Position [367, 47]
#endif /* FIRST */
  }
```

Fig. 15 Example Complex Difference

```
  Annotation {
#ifndef FIRST_Name
    Name "F-14 Flight Control(an updated
         version of this demo is available
         by running 'sldemo_f14')"
#else /* FIRST */
    Name "F-14 Longitudinal Flight Control"
#endif
#ifndef FIRST_Layout
    Position [328, 377]
#else /* FIRST */
    Position [368, 17]
    FontName "Arial"
    FontSize 18
    FontWeight "bold"
#endif
  }
#ifdef FIRST_Layout
  Annotation {
    Name "This demonstration models a flight
         control algorithm for the
         longitudinal motion of a Grumman
         Aer" "ospace F-14."
    Position [367, 47]
  }
#endif /* FIRST */
```

Fig. 16 Example Transformed Complex Difference

expand preprocessor statements in conventional programming languages, handling overlaps between macros as expansions in the scope of the replacement.

Figure 17 shows the beginning of three way diff (using the command *diff3*) of three elements of a clone class from the Matlab demonstration model set. As can be seen from the figure, the differences are the names of the systems, the names of several blocks, the location of one block and the internal identifier *SID*. All of the remaining differences in these three files are to the internal identifiers of the blocks. All of the remaining block types, values and other attributes are identical.

Figure 18 shows the final result, we have merged the contents of the diff with the original files. The highlights show the line common to all three files. Each of the *ifdef* lines have also been annotated with the type of the difference based on the attribute within.

The remaining issue is that blocks in the model are occasionally in different orders in different model files. Simone performs a canonical sort of elements on the subsystems extracted from the models before making comparisons when identifying clone pairs. We can apply this same sorting algorithm to the original model files before performing the diff for tagging.

```
====
1:2c
     Name "m_SimulinkDemoModels_automotive_10_13_so"
2:2c
     Name "m_SimulinkDemoModels_automotive_10_17_so"
3:2c
     Name "m_SimulinkDemoModels_automotive_10_33_so"
====2
1:4c
3:4c
         Name "validate_driver"
2:4c
         Name "validate_passenger"
====2
1:8c
3:8c
            Name "validate_driver"
2:8c
            Name "validate_passenger"
====1
1:12c
               Location [65, 299, 664, 654]
2:12c
3:12c
               Location [69, 319, 668, 674]
====2
1:15c
3:15c
               Name "validate_driver"
2:15c
               Name "validate_passenger"
====
1:29c
                  SID "74"
2:29c
                  SID "110"
3:29c
                  SID "83"
====
...
```

Fig. 17 Example Three Way Diff

5.2 Tagging via Graph Algorithms

An alternate approach to discover and tag variability in Simulink model clones is to treat a Simulink system as a directed graph and apply subgraph matching techniques. In this approach, Simulink blocks represent graph nodes and the connections between blocks represent directed graph edges. This graph-based abstraction makes it immune to changes in layout, which is beneficial for finding a set of common blocks between clones but does not help to discover layout-based variability.

The first step in this approach is to discover a set of common blocks between the system clones. Our current algorithm supports an arbitrary number of clones, but we describe it with 2 clones for simplicity. The goal is to map a subset of blocks in clone 1 to a subset of blocks in clone 2. This mapping is accomplished by first

```
Model {
#ifdef FIRST_Name
     Name "m_SimulinkDemoModels_automotive_10_13_so"
#elif SECOND_Name
     Name "m_SimulinkDemoModels_automotive_10_17_so"
#else
     Name "m_SimulinkDemoModels_automotive_10_33_so"
#endif
   System {
#ifdef FIRST_Name || THIRD_Name
        Name "validate_driver"
#elif
        Name "validate_passenger"
#endif
      Location [428, 407, 944, 880]
      Block {
         BlockType SubSystem
#ifdef FIRST_Name || THIRD_Name
           Name "validate_driver"
#elif
           Name "validate_passenger"
#endif
         Ports []
         Position [115, 123, 300, 177]
         System {
#ifdef THIRD_Layout
              Location [65, 299, 664, 654]
#else
              Location [69, 319, 668, 674]
#endif
...
```

Fig. 18 Example Transformed Three Way Diff

mapping a single block from clone 1 to clone 2 known as the root, then recursively matching each roots' neighbours as well as possible.

This algorithm incorporates two types of block matches: strong match (block type and name must both match), and weak match (block type must match but name can differ). The root blocks are chosen by selecting the strongly matched block pair (one from each clone) with the most connections. Since only one connected subgraph is produced from this algorithm, more connections on the root block increases the chances of a larger resulting subgraph. As block matching grows outward from the root blocks, strong matches are prioritized over weak matches to help disambiguate potential match candidates. It is possible for strong matches to exist in the clones that are not found by this algorithm, e.g. if they are separated from the root block by an unmatchable region.

The end result is a connected subgraph G_1 from clone 1 and a connected subgraph G_2 from clone 2, where each node in G_1 is mapped to a corresponding node in G_2. These subgraphs represent the set of common blocks between two clones, as shown in Figure 19.

Once the common set of blocks is established, the remaining blocks in each clone represent some form of variation. In a merged model file, such as that shown in Section 6.7, the common blocks and their connections remain untagged, but the other blocks can be tagged with their clone variant. This can be accomplished by using

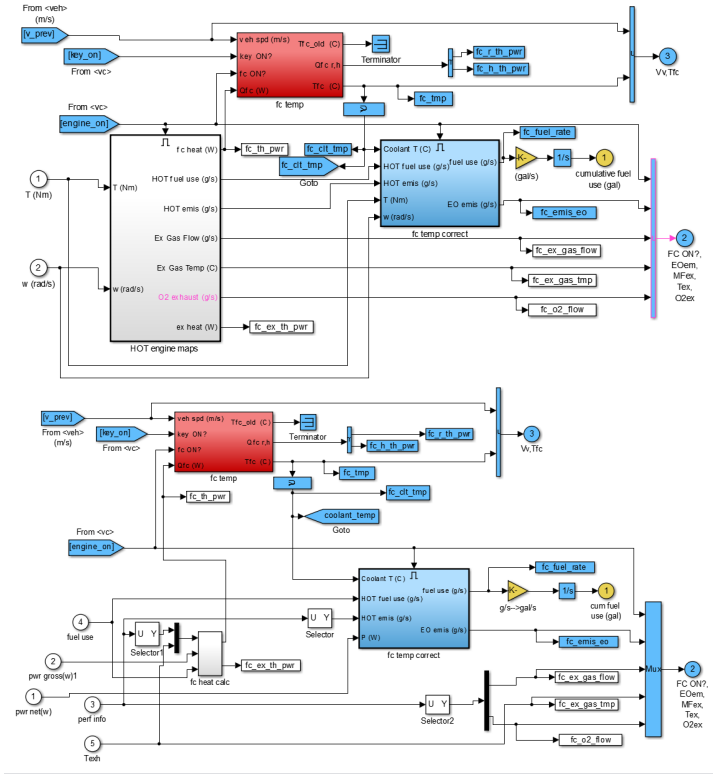


Fig. 19 Common blocks computed by the graph matching algorithm. The root block (red) is determined, then neighbouring blocks are recursively included first by strong match (blue) then by weak match (yellow).

the #ifdef approach from Section 5.1, or by simply adding a new Simulink parameter such as "Variant clone1" to each appropriate variant block. When extending this algorithm to find variation in three or more clones, a tag will specify each clone where the block exists.

6 Representing Variability

Once the variability has been tagged in all instances in a clone class, our goal is to produce a single subsystem model capable of serving for all the instance subsystems of that clone class. To do this, we make use of the Simulink Variant Subsystem Block, a built-in feature of Simulink designed to offer developers the choice between any number of different options for a particular subsystem.

A Variant Subsystem Block can contain any number of different subsystems, as long as they all have the same number of inports. The contained subsystems repre-

sent alternatives for the variant subsystem, and only one of them may be active at any given time. The active subsystem is determined by a logical expression, often making use of a Simulink mode variable. While on the face of it the Variant Subsystem Block seems limited in its expressiveness, being restricted to replacement of entire subsystems, in our work we have leveraged this feature to represent not the subsystem alternatives of the model itself, but rather our variability operators as Variant Subsystem Blocks, allowing us to expose the individual points of variation explicitly in the Simulink environment.

The following subsections outline how we use the Variant Subsystem Block to represent each of our variability operators. We refer back to the example figures in Section 4 as examples of each type of variability.

6.1 Block Variability

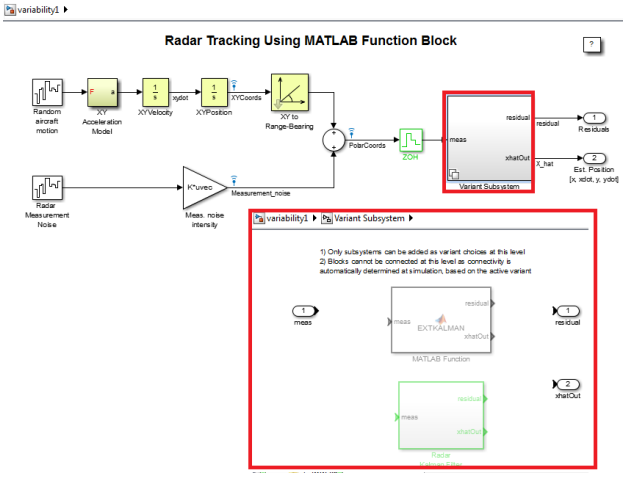
Block variability is perhaps the most intuitive operator to model using the Variant Subsystem Block, especially in the instance of a block being replaced by another similar block. To model this using a Variant Subsystem Block, we simply place each of the alternative blocks in its own subsystem, and place all subsystems in a Variant Subsystem Block. In instances where a block, or group of blocks, is added (or removed, since there is no concept of directionality associated with the block variability operators), the variability is modelled by having the added blocks contained within a subsystem and placed in the Variant Subsystem Block. To represent those instance(s) without those blocks, an empty subsystem, where the inports connect directly to the out ports (or sometimes a terminator), is placed in the Variant Subsystem Block.

To illustrate this operator, recall the example presented in Figure 5. This operator is represented on the main level of the newly created *variability1* model by inserting a Variant Subsystem Block in its place, which then contains two subsystems, one for each of the original options. This can can be seen in Figure 20, which shows the top level model with the Variant Subsystem Block, as well as the two options inside of it (outlined in red).

6.2 Input/Output Variability

Modeling Input/Output Variability is using Variant Subsystem Blocks is somewhat less intuitive. In order to represent this type of variability, the top level subsystem must contain the greatest number of inputs and outputs across all instances. Extra inputs are then dealt with inside the options of the Variant Subsystem, typically by sending them to a terminator in the variants where they are not used. Instances where there are extra outputs are even more difficult to represent, as anything that follows is affected, and will need to be represented inside another Variant Subsystem

(a) Clone pair with block variation



(b) Corresponding variability model

Fig. 20 Representing Block Variability

Block. We can consider the outputs as inputs to the conceptual block "remainder of the model". The extra outputs are then sent to terminators in the instances that do not contain them, and are used as they normally would be in the instances that do contain them.

To illustrate representation of this operator, recall the example of Figure 6, and more specifically the additional outputs from the Aircraft Dynamics Model subsystem in the *sldemo_f14* model, and thus the additional input to the block to the right, which also is in an instance of block variability, as our example. At the top level, there must be a Variant Subsystem Block (Variant Subsystem 1 - outlined in red)

to model the different options for the Aircraft Dynamics Model, which will have four outputs, as this is the maximum number from all of the options. To handle the extra outputs, a second Variant Subsystem Block (Variant Subsystem 2 - outlined in blue) is used. Variant Subsystem 2 also handles the changed subsystem block by offering two options - note one option uses three inputs, and the other uses two. In the instance where only two inputs are required, it is contained in a Container Subsystem (outlined in green), and the third input is sent to a terminator. This can all be seen Figure 21, which shows the top level model (top), as well as the contents of each of the Subsystem Variant Blocks (Variant Subsystem 1 (bottom left) & Variant Subsystem 2 (bottom center)), and the contents of the created Container Subsystem (bottom right). Note that the *variability2* model only accounts for the discussed input/output variability.

6.3 Function Variability

While Function Variability is its own type of operator, it can be modelled in the same manner as Block Variability. Consider that the two blocks with different functions can be thought of as different blocks entirely. Just as with Block Variability, we represent this with a Variant Subsystem Block, with an option for each of the original blocks.

To illustrate this operator, recall the example from Figure 7. Each block is replaced with a Variant Subsystem Block, thus allowing a choice between the two options. Each Variant Subsystem Block can use its own mode variable, thus allowing combinations of options, or a common mode variable, thus only representing the two original observed variants. Figure 22 shows the top level of the *variability3* model, with the eight blocks replaced with Variant Subsystem Blocks.

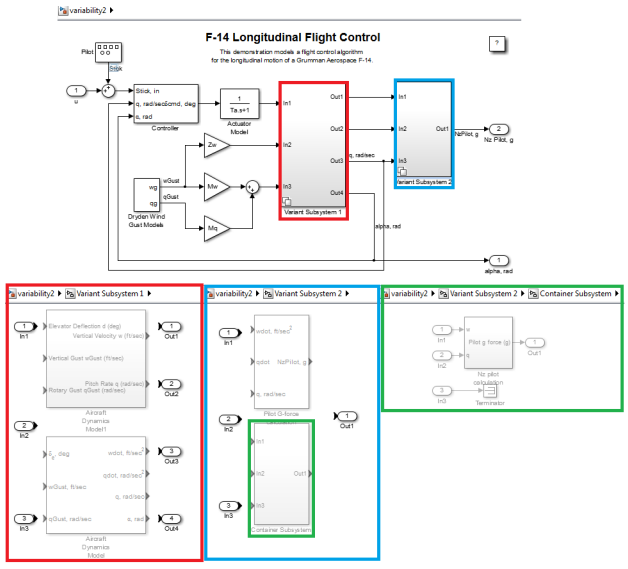
6.4 Layout Variability

Because the layout of the model has no effect on its behaviour, we have chosen to not represent changes in layout in the resulting variability model. Representing the other types of variability in a class that also has layout variability, one layout instance is arbitrarily chosen to represent all models in that class, regardless of their initial layout.

6.5 Subsystem Name Variability

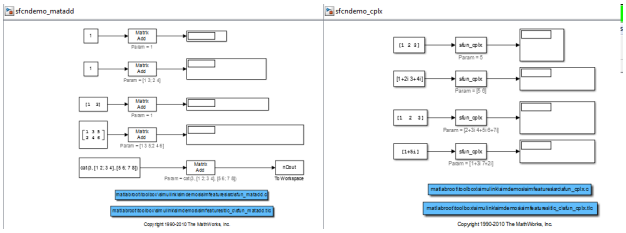
If the contents of a subsystem have not changed, and only the name has changed, there is no behavioural change to the model, however we still wish to represent this

(a) Clone pair with input/output variation

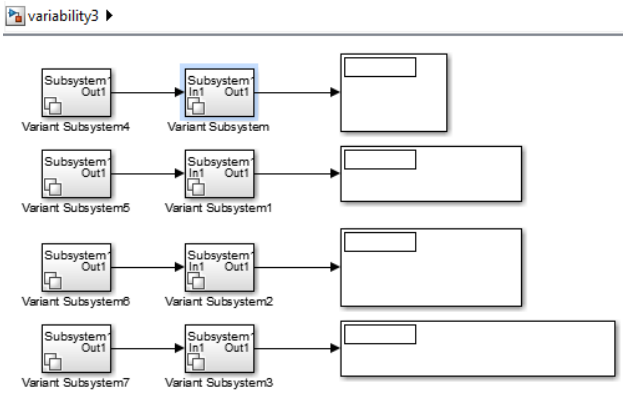


(b) Corresponding variability model

Fig. 21 Representing Input/Output Variability



(a) Clone pair with function variation



(b) Corresponding variability model

Fig. 22 Representing Function Variability

variability as it does have meaning to the developers. This representation would be handled in the exact same way that Block Variability would be; we can just consider the two differently named blocks a different versions of a block, and use them as options in the Variant Subsystem. This would also account for instances where the name has changed, and the actual contents vary slightly, as is the case with our example in Figure 9. Since the implementation for Block Variability has already been demonstrated, there is no need to explicitly illustrate it this paper.

6.6 Combinations of Operators

Through observation of the studied systems, it is evident that each subsystem pattern may require more than one type of variability, and as such, more than one variability operator may need to be applied. Rather than defining combinations of operators as their own unique operator, we have determined that applying any individual operators in succession is sufficient in representing the variability. For example, in an instance where there exists both function variability and block variability, each is handled individually following their respective process.

6.7 Creating Variability Models Directly in Simulink

Currently the process of creating variability models has been automated directly within SimNav for pairs of models. Given two models, the similar blocks and different blocks can be tagged (using a an extension of the algorithm described above), and the different blocks are then merged using the Subsystem Variant Block.

In Section 5.2 we discuss an algorithm to find the common blocks among models in a class of near-miss clones. We use this algorithm to automatically construct a variability model representing the clone class. For this procedure, consider a clone class with n clones ($C_1, C_2, C_3, \ldots, C_n$) where each clone contains a set of blocks ($C_i = \{block_1, block_2, block_3, \ldots\}$).

1. Compute the blocks in common between all clones in the clone class using the algorithm from Section 5.2:

$$C_{common} = \bigcap_{i=1}^{n} C_i$$

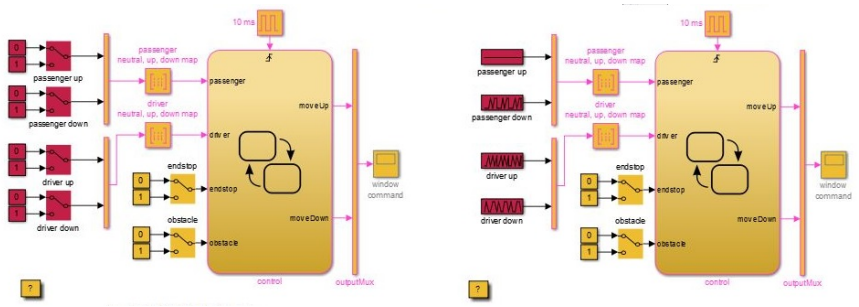
2. Take the complement of the common blocks to find the blocks that vary between clones, one variant set per clone: $V_i = C_i - C_{common}$
3. For each variant set V_i, place its blocks into a subsystem S_i
4. Create a variant system containing subsystems $S_1, S_2, \ldots, S_n$
5. The final variability model contains the common blocks C_{common} and the variant subsystem from the previous step

The connections between blocks are kept wherever possible (i.e. within the common blocks and within each individual set of variant blocks). Connections that traverse the boundary between a variant set and the common set are severed and replaced by input/output ports in the resulting subsystems. The blocks within each subsystem are then connected to these ports such that the meaning of the original clone is restored when its corresponding Variant Subsystem Block is activated. Figure 23 shows a clone pair both before and after the variability procedure has been applied.

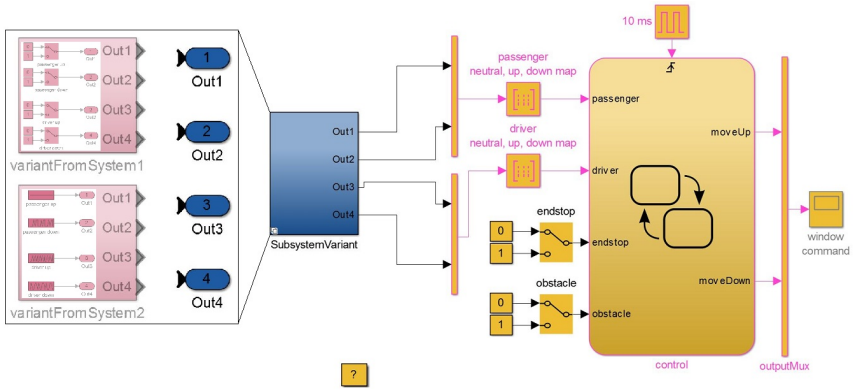
Due to the scalability limitations of graph algorithms, we also plan to continue exploring other possibilities for automating the creation of variability models. We plan on extending the work from Section 5.1 to allow us to directly manipulate the textual representation of subsystem variants into a single Simuulink model capable of expressing variability in a similar manner to that described in this section.

7 Related Work

Model variability is a richly researched area. There have been a number of techniques developed for many different domains [9]. Typically, variability is looked at from a management perspective [5], in that it is an essential property of projects that



(a) Clone pair with variation



(b) Variability model

Fig. 23 (a) A clone pair with common blocks in yellow and different blocks in red, and (b) the corresponding variability model.

needs to managed. There have also been steps taken to semi-automatically extract variability in code-based projects [13] and model-based projects [20] in order to manage it. The difference between our work and the latter is we use model clone detection, via Simone, as the starting point for finding variability among, and grouping into classes/patterns, sets of models, whereas they compare systems recursively by mapping similar components of the same type based on different criteria; like name similarity, number of identical parameter values, connections, and more; in order to get a weighted similarity sum between zero to one. Similar to one of our two presented approaches for tagging variability, they identify variation points using a graph-based approach. It is our contention that using classes, patterns, and clustering provides a better basis for representing variability and more useful information then basic name and property similarity. In addition, graph-based approaches for matching typically do not perform well on larger model sets due to subgraph isomorphism [6]. Our approach avoids this issue because our model clone detection algorithm uses the textual representations of the models [1]. While one of our pro-

posed tagging approaches uses graph algorithms, it is applied only to sets of single models/graphs that have already been identified as similiar, so the subgraph isomorphism complexity will not be an issue on this small scale.

Albeit a relatively new sub-area, there is some existing work on variability in Simulink models. Weiland and Manhart [27] argue the necessity for modelling variability in Simulink. They introduce a classification of possible concepts that can be employed in order to represent Simulink variability: Model elements for model adaptation, conditional model elements, and model elements for data variability. In our paper, we realize the first of these concepts proposed by them in order to explicate variability among Simulink clones. While Weiland and Manhart note that using the variant subsystem block does not perform well in regards to their binding time, we have found, from speaking to our industrial partners and engineers, that this solution is most preferable for them. They want the variability to be demonstrated and usable within their native Simulink environments. In addition, we have yet to witness any binding time limitations or concerns in using the variant subsystem block, but this is something we will continue to monitor as we employ this solution in our methods.

One Simulink-specific approach to encoding variability is accomplished by Haber et al. [11], who note that functional-modelling approaches for representing Simulink variability are often complex and do not scale well to larger systems. Thus, they propose Delta Simulink, which is a first-class language that includes single step operations like add, remove, modify, and replace. While it is an operational approach, it is also graphical in that users can illustrate their deltas in a separate, non-Simulink, viewer. Because their approach is operational, they note that "some modification operations have to be split up into several deltas to be applied in a sequence." Our representation avoids this in that it is declaritive. The sets of models belonging to a related cluster indicate, all at once, exactly how the models differ. This declaritive representation is understandably simplier, while still being precise, and has been recieved well thus far by our industrial partners. Another concrete deficiency is that Delta Simulink is another modelling language and the tools they develop are external from Simulink. During our conversations with the industrial engineers, it was made clear from the beginning that a key priority was to have an approach that works within Simulink and can be as least disruptive as possible to their processes. Seeing as Delta Simulink is a new language, albeit an extension to Simulink, and exists outside the Simulink editor environment, their solution was not ideal for our purposes.

Steiner et al. [21] manage Simulink variability by using and contrasting Pure::variants, which has a Simulink connector that uses "point of change" information; and Hephaestus, which has a graphical interface that allows developers to select system elements to be used to generate specific product line instances. Their approach uses conditional model elements in order to represent Simulink variability, which, as we discussed previously in this paper, would not be ideal for Simulink clone variants. In addition, the learning curve for using their technique is quite high as engineers, would have to familiarize themselves with Pure::variants and Hephaestus. It uses the Hephaestus graphical interface, which is external to the Simulink

editor native environment and is another reason this approach was not well suited for us nor our partners.

Managing clones in product lines involves cases where systems using product lines or feature models have exact duplicates or similar segments of a related product line. Rubin et al. [18, 19] provide a framework for handling such systems that includes abstract operators that allow engineers to reason and manage clones detected in these systems. Thier work is focused on the product line, higher-level of abstraction, level while our work is intended explicitly for Simulink models. In addition, our approach is declarative, while their's is operational.

While variability involves looking at how systems differ at a somewhat larger scale, model mutations focus on step-wise changes to a model in order to perform various types of analysis. Recently, we proposed and validated a taxonomy of Simulink model mutations [25] for the purposes of injecting various types of Simulink model clones [24]. There is also work on Simulink model mutations that describe mutation instances that explicitly try to mutate a model's run-time properties [4, 12, 28]. While this mutation analysis work was helpful in guiding how we viewed Simulink variability, we essentially were focused on a higher and more-feature-oriented level.

Basit and Dajsuren [3] use a constraint language in order to model mutations among Simulink clones with the purpose of allowing clone management that is entirely separate from the models. Thier work is concerned with a different, but related, task. Their work looks at the model clones as simply clones, while we focus on the (sub)system level in order to to identify candidates for (sub)system variants. Our approach is more geared towards tool support and working directly in the Simulink environment to assist engineers. In addition, they do not have a tool at this point and engineers would have to their constraint language.

Calculating and representing variability in models is analogous to calculation and representation phases of Model Comparison [23]. The first phase, calculation, involves discovering what is the same and what is different, while the second phase of model comparison, representation, addresses the form that the differences and similarities among models take. There are many different ways of achieving model comparison calculation [22], but nothing was specifically suited to identifying variability among a set of Simulink models as outlined in our framework. As such, we presented both a Unix-diff and graph-based approach in this paper. Model comparison representation can be realized in an operational fashion, such as through the use of edit scripts; or in a more declarative fashion like those that represent the differences in a model-based or abstract-syntax-like form. The representation we present in this paper falls more in the declarative category as we are representing the variability in model form by having multiple model-implementation options linked to a specific variation point.

For our graph-based variation analysis, where we explicate the similarities and differences within a set of already identified "similar" models, we based our approach on what Deissenboeck et al. [10] did in their ConQAT graph-based model-clone algorithm. The difference here is that we are using a graph-based approach on only the micro level in order to compare and constrast a small set of models. The

algorithm ConQAT uses can not detect near-miss clones, while our model-clone detection approach can [1]. Using an approach that does not identify near-miss clones from the graph-based variation analysis perspective is sufficient, as we need only to identify variation points.

8 Conclusion

Based on the six example systems of the Simulink demonstration set, we have empirically derived five variability operators for Simulink Models. These five operators encompass all of the different types of variability observed from the initial analysis of similar subsystem variance provided by Simone, a hybrid sub-model clone detector. We have presented two methods for tagging variability across a set of similar Simulink models, one based on text differencing and one on graph matching. Both of these processes have been automated for pairs of similar subsystems, and we are currently extending them to handle N-way differencing. We have shown how each of the five variability operators can be represented directly in the Simulink environment through a novel use of the Variant Subsystem Block by extending our SimNav tool to support this feature. While the variability representation using the graph matching approach showed good results in representing variability for small subsystems, we are still experimenting our tool for larger subsystems and expect to face some scalability issues related to our graph matching algorithm. For that reason we continue to explore the alternative text-based implementation carrying on our tagging approach using #ifdef . This alternative approach has the advantages that it automatically tags variations based with the types of variability operators and is likely to scale better than the graph based approach.

9 Acknowledgments

This work is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) as part of the NECSIS Automotive Partnership with General Motors, IBM Canada and Malina Software Corp., and by an Ontario Research Fund Research Excellence grant.

References

[1] Alalfi MH, Cordy JR, Dean TR, Stephan M, Stevenson A (2012) Models are code too: Near-miss clone detection for Simulink models. In: ICSM'12 - 28th Int. Conf. on Software Maintenance, pp 295–304

[2] Alalfi MH, Rapos EJ, Stevenson A, Stephan M, Dean TR, Cordy JR (2014) Semi-automatic identification and representation of subsystem variability in Simulink models. In: ICSME'14 - 30th Int. Conf. on Software Maintenance and Evolution, pp 486–490
[3] Basit HA, Dajsuren Y (2014) Handling clone mutations in Simulink models with VCL. In: IWSC'14 - 8th Int. Works. on Software Clones, pp 1–8
[4] Binh NT, et al (2012) Mutation operators for Simulink models. In: KSE'12 - 4th Int. Conf. on Knowledge and Systems Engineering, pp 54–59
[5] Capilla R, Bosch J, Kang KC (2013) Systems and Software Variability Management. Springer
[6] Cook SA (1971) The complexity of theorem-proving procedures. In: 3rd ACM Symposium on the Theory of Computing, ACM, pp 151–158
[7] Cordy JR (2006) The TXL source transformation language. Science of Computer Programming 61(3):190–210
[8] Cordy JR (2013) Submodel pattern extraction for Simulink models. In: SPLC'13 - 17th Int. Conf. on Software Product Lines, pp 7–10
[9] Czarnecki K, Grunbacher P, Rabiser R, Schmid K, Wąsowski A (2012) Cool features and tough decisions: a comparison of variability modeling approaches. In: VaMoS'12 - 6th Int. Works. on Variability Modelling of Software-Intensive Systems, pp 173–182
[10] Deissenboeck F, Hummel B, Jürgens E, Schätz B, Wagner S, Girard JF, Teuchert S (2008) Clone detection in automotive model-based development. In: Proceedings of the 30th international conference on Software engineering, ACM, pp 603–612
[11] Haber A, Kolassa C, Manhart P, Nazari PMS, Rumpe B, Schaefer I (2013) First-class variability modeling in Matlab/Simulink. In: VaMoS'13 - 7th Int. Works. on Variability Modelling of Software-intensive Systems, pp 11–18
[12] He N, Rümmer P, Kroening D (2011) Test-case generation for embedded Simulink via formal concept analysis. In: DAC'11 - 48th Design Automation Conf., pp 224–229
[13] Kastner C, Dreiling A, Ostermann K (2013) Variability mining: Consistent semiautomatic detection of product-line features. IEEE Trans Software Engineering 40(2)
[14] Malton A, Schneider K, Cordy J, Dean T, Cousineau D, Reynolds J (2001) Processing software source text in automated design recovery and transformation. In: IWPC'01 - 9th Int. Works. on Program Comprehension, pp 127–134, DOI 10.1109/WPC.2001.921724
[15] Object Management Group (2009) Variability modeling. *http://www.omgwiki.org/variability/doku.php?id=introduction_to_variability_modeling*
[16] Rapos EJ, Stevenson A, Alalfi MH, Cordy JR (2015) SimNav: Simulink navigation of model clone classes. In: 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp 241–246, DOI 10.1109/SCAM.2015.7335420

[17] Roy CK, Cordy JR (2008) NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: ICPC'08, 16th IEEE Int. Conf. on Program Comprehension, pp 172–181
[18] Rubin J, Chechik M (2013) A framework for managing cloned product variants. In: ICSE'13 - 35th Int. Conf. on Software Engineering, pp 1233–1236
[19] Rubin J, Czarnecki K, Chechik M (2013) Managing cloned variants: a framework and experience. In: SPLC'13 - 17th Int. Conf. on Software Product Lines, pp 101–110
[20] Ryssel U, Ploennigs J, Kabitzsch K (2010) Automatic variation-point identification in function-block-based models. In: GPCE'10 - 9th Int. Conf. on Generative Programming and Component Engineering, ACM, pp 23–32
[21] Steiner E, Masiero P, Bonifácio R (2013) Managing SPL variabilities in UAV Simulink models with Pure:variants and Hephaestus. CLEI Electronic Journal 16(1):1–7
[22] Stephan M, Cordy JR (2012) A Survey of Methods and Applications of Model Comparison. Tech. Rep. 2011-582, Queen's University, revision 3
[23] Stephan M, Cordy JR (2013) A Survey of Model Comparison Approaches and Applications. In: International Conference on Model-Driven Engineering and Software Development (Modelsward), SCITEPRESS, pp 265–277
[24] Stephan M, Cordy JR (2018) MuMonDE: A framework for evaluating model clone detectors using model mutation analysis. Software Testing, Verification and Reliability 0(0):e1669, DOI 10.1002/stvr.1669, https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1669
[25] Stephan M, Alalfi MH, Cordy JR (2014) Towards a Taxonomy for Simulink Model Mutations. In: International Conference on Software Testing, Verification and Validation Workshops (ICSTVVW), pp 206–215, DOI 10.1109/ICSTW.2014.17
[26] The Mathworks Inc (2014) Simulink version 8. *http://www.mathworks.com/products /simulink/*
[27] Weiland J, Manhart P (2014) A classification of modeling variability in Simulink. In: VaMoS'14 - 8th Int. Works. on Variability Modelling of Software-Intensive Systems, pp 1–7
[28] Zhan Y, Clark J (2005) Search-based mutation testing for Simulink models. In: GECCO'05 - Genetic and Evolutionary Computation Conf., pp 1061–1068