Semi-automatic Identification and Representation of Subsystem Variability in Simulink Models

Manar H. Alalfi, Eric J. Rapos, Andrew Stevenson, Matthew Stephan,
Thomas R. Dean and James R. Cordy
School of Computing, Queen's University
Kingston, Ontario, Canada
{alalfi, eric, andrews, stephan, dean, cordy}@cs.queensu.ca

***Abstract*—This paper presents a semi-automated framework for identifying and representing different kinds of variability in Simulink models. Based on the observed variants found in similar subsystem patterns inferred using Simone, a text-based model clone detection tool, we propose a set of variability operators for Simulink models. By applying these operators to six example systems, we are able to represent the variability in their 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.The framework is aimed at providing assistance to engineers to identify, understand, and visualize patterns of subsystems in a large model set. This understanding may help in reducing maintenance effort and bug identification at an early stage of the software development.**

I. Introduction

Software variability management (SVM) is an important area of research that gained a lot of interest in the last two decades, especially for its vital role in developing reusable and easily maintained software product line (SPL) assets [4]. SVM is a complex, multifaceted problem that intersects with several traditional software engineering topics, including configuration management, run-time dynamism, domain specific languages, model-driven engineering, and software architecture.

One facet of SVM is variability modeling, 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. Variability modeling continues to gain interest from industry, and variability support in modeling tools, including Mathworks' Simulink and IBM's Rhapsody, is one of the 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 variation points in their artifacts is the first and most important step towards enabling variability modeling. Many methods have been proposed for analyzing variability from a requirements point of view, as well as connecting that to the implementation [9], [15]. However, there remains a need for techniques that analyze existing system requirements and implementations for commonality and variability in an automated way.

In this paper we present a framework for identifying variability candidates from existing software intensive systems modeled using one of the most popular modeling languages for hybrid hardware/software systems, Simulink [17]. Our framework uses an efficient model clone detection technique to automatically identify subsystem variants from a large pool of existing Simulink models. It then classifies those variants according to a set of proposed variability operators.

The framework is aimed at providing tool support to automatically represent the recovered subsystem variability directly in the Simulink environment, and thus provide assistance to engineers to identify, understand, and visualize patterns of subsystems in a large model set. This understanding may help in reducing maintenance efforts and enhancing bug identification at an early stage of software development, both on the model level and before model semantics are transformed into actual code. Furthermore, the creation of Variability Models allows for maximum reuse of these well-maintained models, allowing developers to simply insert a Variability Model and select the active variants they desire. We demonstrate our framework on six systems from Mathworks Demo set, and provide initial empirical evidence on the soundness of our proposed variability operators. We describe the stages of our framework using a running example.

The main contributions of this paper are as follows:

- A semi-automated framework for the identification and representation of subsystem variability in Simulink models.
- A set of variability operators for Simulink that have been empirically inferred from the analysis of six Simulink systems representing a range of applications.

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

II. Variability Identification using Simone

To determine an appropriate set of Simulink subsystem variability operators, we used the set of models in 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.

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 submodel clone detector [1]. Simone is a text-based

TABLE I. SIMONE CLONE DETECTION RESULTS AT A DIFFERENCE THRESHOLD OF 20%

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 II. OBSERVED INSTANCES OF THE IDENTIFIED VARIABILITY OPERATORS

System	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
Total	**61**	**74**	**25**	**13**	**31**

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 [1], extended to handle graphical models.

To identify and categorize subsystem variations, we applied Simone to the set of models in each of the six Simulink demonstration systems. From each set of models. We had Simone generate a database of near-miss subsystem clone pairs, representing pairs of model subsystems which are largely similar but 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 uses the efficient exemplar-based algorithm of NICAD 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. This text-based similarity algorithm effectively avoids the subgraph isomorphism problem, which is NP-complete, and the scalability of our approach rests largely on this proven scalable clustering method [7].

In practice the clone classes resulting from this analysis can 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 [6] to directly present similar subsystems in the Simulink model editor.

Table I 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 others in the class. A particular element of each clone class is chosen by our framework as an exemplar, from which the others are considered to be variants. We then classified the nature of these variants to derive the variability operators presented in the next section.

For each of the six systems, we determined the number of instances of each type of variability, and ensured that all

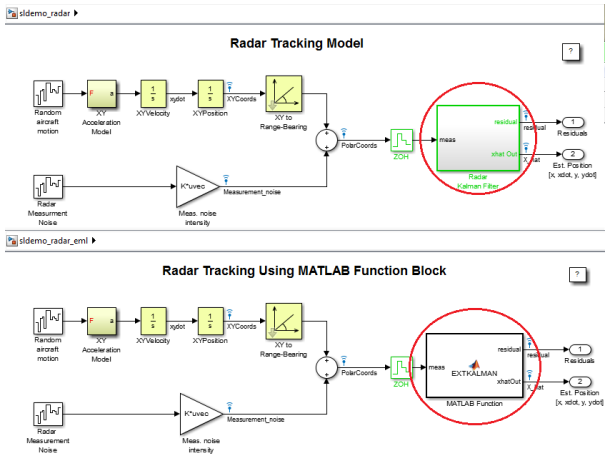

Fig. 1. Block Variability

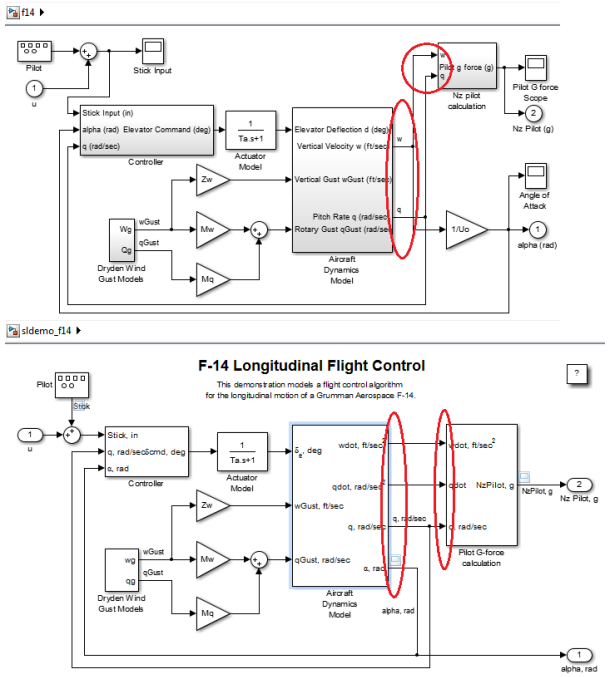

Fig. 2. Input/Output Variability

observed variations could be covered by the set of variability operators. The results of this categorization can be found in Table II. From this, we conclude that on this set of systems the most common types of variability are 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.

III. VARIABILITY OPERATORS

Through 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 1 (circled in red).

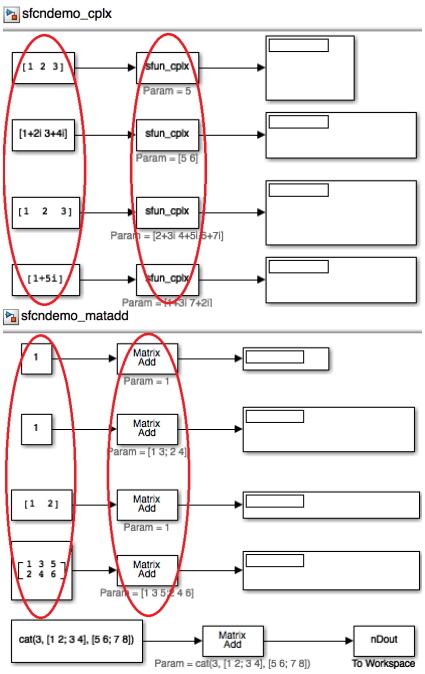

Fig. 3. Function Variability

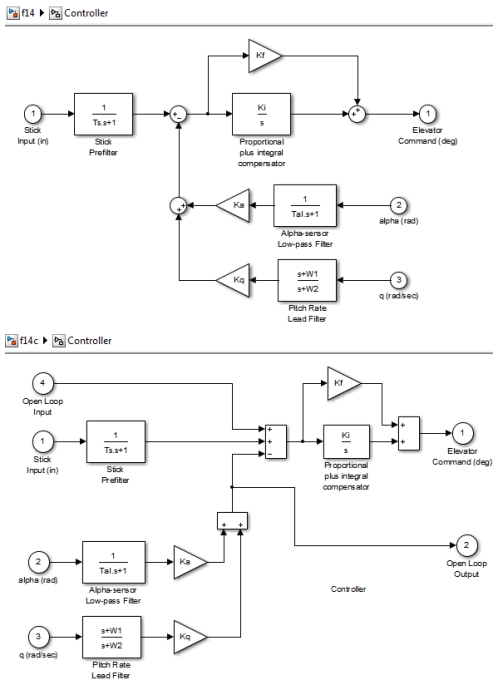

Fig. 4. Layout Variability

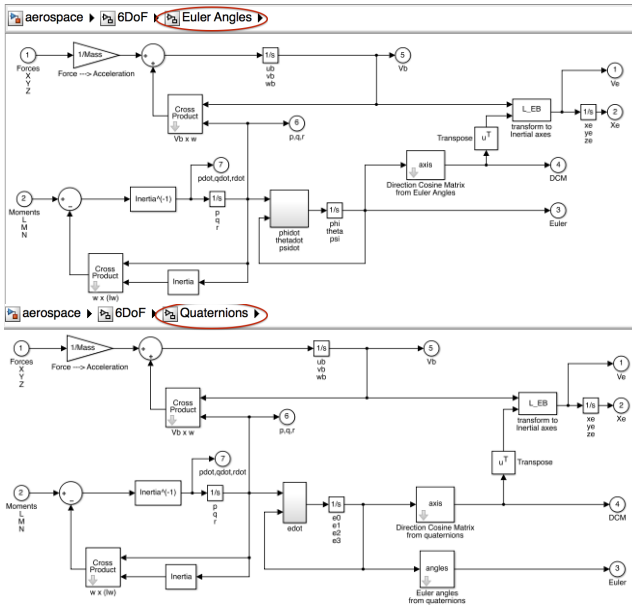

Fig. 5. Subsystem Name Variability

Input/Output Variability: Changes in the number of input/output ports for a specific block. This type of variability is shown in Figure 2 (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 3 (different functions and constants in many blocks - all circled in red).

Layout Variability: Changes to the layout information of the model elements, such as block position. This type of variability is shown in Figure 4 (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 5 (circled in red).

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

In this paper, we explored an approach for the initial tagging of variation using graph matching algorithms to determine identical sub-graphs, and we are currently experimenting with another approach using *diff* and *#ifdef* on the normalized textual representation of the models, however, we have not included details of the second approach due to pages limits.

An approach to discover and tag variability across Simulink subsystem clones is to treat the subsystems as directed graphs 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 subsystem clones. The goal is to map a subset of blocks in clone 1 to a subset of blocks in clone 2. Our current implementation does this for only two clones, but we are currently extending it to clone classes of any size. This mapping is accomplished by first mapping a single block from clone 1 to clone 2 known as the root, then recursively matching each root's neighbours as well as possible.

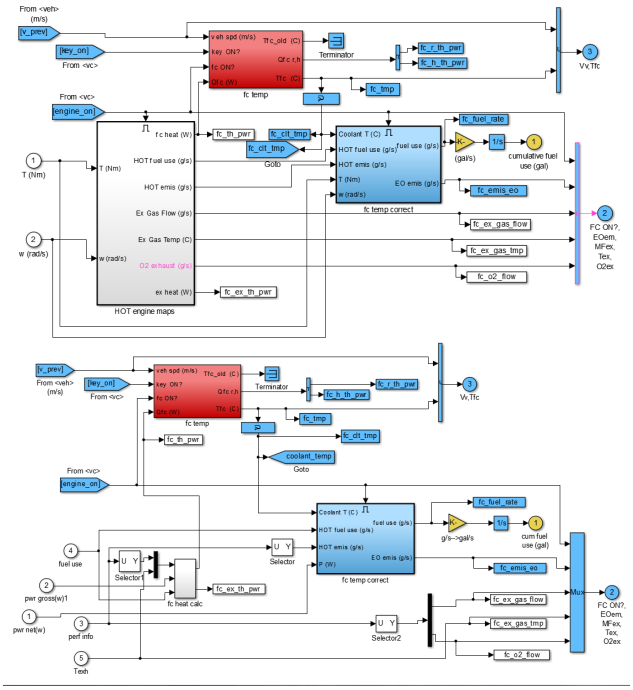

Fig. 6. Common blocks computed by the graph matching algorithm. The root block (red) is determined, then neighbouring blocks are recursively included first by strong match (blue) then by weak match (yellow).

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, for example, 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 the two clones, as shown in Figure 6.

Once the common set of blocks is established, the remaining blocks in each clone represent the elements of variation. In a merged subsystem file, the common blocks and their connections remain untagged, but the other blocks can be tagged with their clone variant. This can be accomplished 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.

V. Representing Variability

Once the variability has been tagged in all instances in a clone class, we must produce a single subsystem file capable of serving for all the instances of that clone class. To do this, we make use of the Simulink Variant Subsystem Block, a built-in feature designed to offer developers the choice between a 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 represent 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.

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.

VI. Related Work

Model variability is a richly researched area. There have been a number of techniques developed for many different domains [8]. Typically, variability is looked at from a management perspective [4], in that it is an essential property of projects that needs to managed. There have also been steps taken to semi-automatically extract variability in code-based projects [11] and model-based projects [14] 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. In contrast, 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 our two presented approaches for tagging variability, they identify variation points using a graph-based approach.

Albeit a relatively new sub-area, there is some existing work on variability in Simulink models. Weiland and Manhart [18] argue the necessity for modeling variability in Simulink. They introduce a classification of approaches that can be employed to represent Simulink variability: Model elements for model adaptation, conditional model elements, and model elements for data variability. In our paper, we utilize the first of these in order to realize variability among Simulink clones. Wille et al. [19] use architecture design languages (ADL) and metrics pertaining to components' names, functions, inports and outports to explicate variability into family models. Their work differs from ours in that they parse and convert non-hierarchical Simulink models to an XML ADL format and represent variation as Pure::variants. By contrast, we use direct clone detection on the Simulink source, and represent variation directly in editable Simulink models.

An example Simulink-specific approach to encoding variability is accomplished by Haber et al. [9], who note that functional-modeling 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. Our work differs in that it focuses specifically on clones and is situated entirely within the Simulink environment. Steiner et al. [15] 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, would not be ideal for Simulink clone variants.

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. [12], [13] provide a framework for handling such systems that includes abstract operators that allow engineers to reason and manage clones detected in these systems. Our work differs in that it is explicitly focused on Simulink models and has more of a declarative representation of the variability that can be used within the Simulink environment.

We represent our clone variability using the built-in Simulink "Model Variants" blocks. In contrast, Basit and Dajsuren [2] used a constraint language in order to model variability among Simulink clones with the purpose of allowing clone management that is entirely separate from the models. While we also keep clone management separate from the Simulink models themselves, we attempt to leave the management in the native Matlab Simulink domain using *SimNav* as we believe this is a much more natural environment for the engineers.

While variability involves looking at how systems differ at a somewhat larger scale, model mutations focus on stepwise changes to a model in order to perform various types of analysis. Recently, we proposed and validated a taxonomy of Simulink model mutations [16] for the purposes of injecting various types of Simulink model clones. There is also work on Simulink model mutations that describe mutation instances that explicitly try to mutate a model's run-time properties [3], [10], [20]. 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.

VII. Conclusion

Based on the six example systems of the Simulink Demonstration set, we have identified 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 text-based model clone detector. We have presented a method for tagging variability across a set of similar Simulink models based on graph matching, and we are currently experimenting with another one based on text differencing. Both of these processes have been automated for pairs of similar subsystems, and we are currently extending them to N-way differencing.

So far the identification of variant subsystems, as Simone subsystem clone classes, and the differencing of pairs of similar subsystems has been fully automated. Based on the textual representation of a set of instance subsystems, we are working on the final step of fully automating the representation of variability based on the Variant Subsystem Block representation of our variability operators using a TXL[5] source transformation to build a variance model.

References

[1] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson. Models are code too: Near-miss clone detection for Simulink models. In *ICSM'12 - 28th Int. Conf. on Softw. Maint.*, pages 295–304, 2012.

[2] H. A. Basit and Y. Dajsuren. Handling clone mutations in Simulink models with VCL. In *IWSC'14 - 8th Int. Works. on Softw. Clones*, pages 1–8, 2014.

[3] N. T. Binh et al. Mutation operators for Simulink models. In *KSE'12 - 4th Int. Conf. on Knowledge and Systems Eng.*, pages 54–59, 2012.

[4] R. Capilla, J. Bosch, and K.-C. Kang. *Systems and Software Variability Management*. Springer, 2013.

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

[6] J. R. Cordy. Submodel pattern extraction for Simulink models. In *SPLC'13 - 17th Int. Conf. on Softw. Product Lines*, pages 7–10, 2013.

[7] J. R. Cordy and C. K. Roy. Tuning research tools for scalability and performance: The NICAD experience. *Sci. Comput. Program.*, 79:158–171, 2014.

[8] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski. Cool features and tough decisions: a comparison of variability modeling approaches. In *VaMoS'12 - 6th Int. Works. on Variability Modelling of Software-Intensive Systems*, pages 173–182, 2012.

[9] A. Haber, C. Kolassa, P. Manhart, P. M. S. Nazari, B. Rumpe, and I. Schaefer. First-class variability modeling in Matlab/Simulink. In *VaMoS'13 - 7th Int. Works. on Variability Modelling of Software-intensive Systems*, pages 11–18, 2013.

[10] N. He, P. Rümmer, and D. Kroening. Test-case generation for embedded Simulink via formal concept analysis. In *DAC'11 - 48th Design Automation Conf.*, pages 224–229, 2011.

[11] C. Kastner, A. Dreiling, and K. Ostermann. Variability mining: Consistent semiautomatic detection of product-line features. *IEEE Trans. Softw. Eng.*, 40(2), 2013.

[12] J. Rubin and M. Chechik. A framework for managing cloned product variants. In *ICSE'13 - 35th Int. Conf. on Softw. Eng.*, pages 1233–1236, 2013.

[13] J. Rubin, K. Czarnecki, and M. Chechik. Managing cloned variants: a framework and experience. In *SPLC'13 - 17th Int. Conf. on Softw. Product Lines*, pages 101–110, 2013.

[14] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Automatic variation-point identification in function-block-based models. In *GPCE'10 - 9th Int. Conf. on Generative Prog. and Component Eng.*, pages 23–32, 2010.

[15] E. Steiner, P. Masiero, and R. Bonifácio. Managing SPL variabilities in UAV Simulink models with Pure:variants and Hephaestus. *CLEI Electronic Journal*, 16(1):1–7, 2013.

[16] M. Stephan, M. Alalfi, and J. R. Cordy. Towards a taxonomy for Simulink model mutations. In *Mutation'14 - 9th ICST Int. Works. on Mutation Analysis*, pages 206–215., 2014.

[17] The Mathworks Inc. Simulink version 8. *http://http://www.mathworks.com/products/simulink/*, 2014.

[18] J. Weiland and P. Manhart. A classification of modeling variability in Simulink. In *VaMoS'14 - 8th Int. Works. on Variability Modelling of Software-Intensive Systems*, pages 1–7, 2014.

[19] D. Wille, S. Holthusen, S. Schulze, and I. Schaefer. Interface variability in family model mining. In *SPLC'13 - 17th Int. Conf. on Softw. Product Lines, co-located workshops*, pages 44–51, 2013.

[20] Y. Zhan and J. Clark. Search-based mutation testing for Simulink models. In *GECCO'05 - Genetic and Evolutionary Computation Conf.*, pages 1061–1068, 2005.