Identification of Simulink Model Antipattern Instances using Model Clone Detection

Matthew Stephan
Department of Computer Science and Software Engineering
Miami University
Oxford, Ohio, USA
stephamd@miamioh.edu

James R. Cordy
School of Computing
Queen's University
Kingston, Ontario, Canada
cordy@cs.queensu.ca

***Abstract*—One challenge facing the Model-Driven Engineering community is the need for model quality assurance. Specifically, there should be better facilities for analyzing models automatically. One measure of quality is the presence or absence of good and bad properties, such as patterns and antipatterns, respectively. We elaborate on and validate our earlier idea of detecting patterns in model-based systems using model clone detection by devising a Simulink antipattern instance detector. We chose Simulink because it is prevalent in industry, has mature model clone detection techniques, and interests our industrial partners. We demonstrate our technique using near-miss cross-clone detection to find instances of Simulink antipatterns derived from the literature in four sets of public Simulink projects. We present our detection results, highlight interesting examples, and discuss potential improvements to our approach. We hope this work provides a first step in helping practitioners improve Simulink model quality and further research in the area.**

I. Introduction

While Model-Driven Engineering (MDE) is becoming increasingly prevalent in the Software Engineering community, especially in both business and embedded domains [1], there are still numerous areas that need to be addressed as both model-based development and systems mature. In MDE, higher-level abstractions, or models, are the primary artifacts in all phases of the Software Engineering life cycle, including requirements, design, implementation, testing, and maintenance. Given the importance and longevity of these models, ascertaining and improving quality of the models and other artifacts of interest in MDE projects, known as model quality assurance, becomes imperative [2], [3]. Compared to quality assurance (QA) for more traditional software development paradigms such as code-based development, model quality assurance is much less refined and researched [4]. If MDE is to continue to flourish and allow engineers to fully reap all of its rewards, then evaluation and improvement of model quality is essential.

One method of assessing the quality of Software Engineering systems is to identify and report well-established "good" and "bad" ways of solving specific design questions and constructing the systems' artifacts. These are known as design patterns [5] and antipatterns [6], respectively, and will be referred to collectively simply as "patterns" herein when we are referring to both. Detection of instances of these patterns provides analysts the facility to identify functional and non-functional properties that can be used to reason about the quality of their systems. Typically, patterns are derived by practitioners based on experience. The pattern data includes where and when the pattern is applicable, the various ways of implementing or refactoring the pattern, and a description and justification for the pattern. In many cases, the patterns are accompanied by an abstract representation model [7], with detection being accomplished by code evaluation [8] or, in other cases, complex textual rules that evaluate models [9].

Recently, we argued that instances of model-based patterns can be detected in model-based software using model clone detection [10]. Existing non-model-based software pattern detection can be onerous, requiring those analyzing systems to switch abstraction levels between text and abstract pattern representations, and delaying pattern instance identification to later in the engineering process. Using model clone detection [11], which is a form of model comparison [12], to locate instances of model design patterns and antipatterns (1) can be done early in the Software Engineering process, (2) is applicable to systems that are comprised fully or mostly of model-based software, and (3) keeps the abstraction level consistent between the patterns and the systems under investigation.

This paper builds upon our initial and general ideas of using model clone detection to accomplish model pattern instance detection [10]. In that previous work, we discussed various model clone detection techniques' suitability for detecting pattern instances. In this paper, we flesh out that idea and validate it by building a prototype SIMulink Antipattern Instance Detector (SIMAID) capable of detecting model antipattern instances in Simulink models. We focus on Simulink models specifically because Simulink model clone detection is the most mature of all model clone detection domains [13] and is of interest to our industrial partners. We target Simulink antipatterns because they are more researched and available than Simulink design patterns and have a very immediate and clear impact on model quality in that their detection indicates refactoring opportunities. This is also a likely reason for the increased availability of Simulink antipatterns over design patterns. It is our hope that our work will inspire future model-based pattern instance detectors for other modeling domains, thus improving model quality assurance as a whole.

We begin this paper in Section II with background information and related work on Simulink, Model Clone Detection, pattern detection, and previous research. We then introduce

978-1-4673-6908-4/15 © 2015 IEEE

MODELS 2015, Ottawa, ON, Canada
Foundations

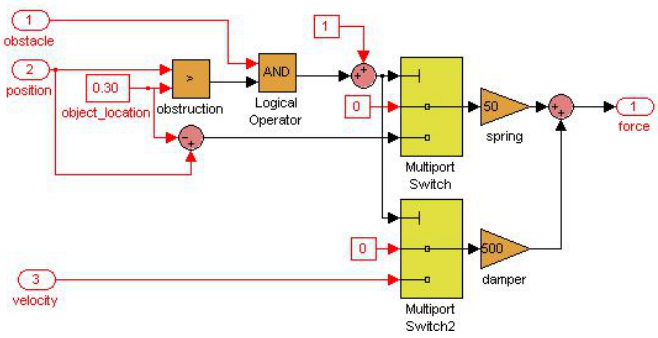

Fig. 1. Window Obstacle Effects Subsystem[1]

SIMAID in Section III including the Simulink antipatterns included in its scope and an overview of its process. Section IV presents our case study using SIMAID on four sets of publicly available Simulink projects. We discuss evaluation, threats to validity, and future work in Section V, and conclude the paper in Section VI.

II. Background and Related Work

This section provides background information on Simulink, model clone detection, and patterns. In addition, we highlight the novelty of our work by comparing it with existing pattern instance detection and analysis techniques. Much of this section is paraphrased and extended from the initial short paper in which we proposed the model clone-based technique [10].

A. Simulink

Simulink [14] is a modeling language in the Matlab development environment that is intended for model-based design and simulation. Simulink models are data-flow models that are comprised of three levels of granularity: whole models, (sub) systems, and blocks. Models are composed of systems, and systems contain other (sub) systems and blocks. Each block type has its own unique semantics and parameters, typically comes from a library, and is connected to other blocks using lines to allow for simulation.

Figure 1 illustrates a Simulink system modeling a car window that encounters an obstacle in its path[1]. The system and blocks within it represent the spring and damping effects during the window's interaction with an obstacle to calculate force. The blocks on the left of the figure represent the input to the system, while the rightmost block outputs the force to be applied. The Multiport Switch blocks use their first input to determine which of their remaining inputs will be output to the gain blocks (spring and damper) that follow. In many cases, Simulink blocks have corresponding C code that can automatically be generated for embedding on a variety of target platforms. Engineers and analysts edit Simulink models through the Matlab environment by navigating through systems and adding, modifying, and deleting blocks and lines.

[1]http://www.mathworks.com/products/simulink/model-examples.html

B. Model Clone Detection

Model comparison refers to comparing and contrasting sets of models and identifying similarities and differences, respectively. While it is an emerging field, there are many different model comparison techniques and approaches [13], including model clone detection. Model clone detection is a specialization of model comparison that accomplishes similarity-based model matching [11]. It works by finding similar or identical fragments within sets of models in an MDE project and identifying those fragments as clone pairs, which can be clustered as classes. For non-identical, or near-miss (Type 3) clones, this similarity is capped by a specific threshold, for example, clone pairs that are 80% similar. For identical (Type 1) clones and renamed clones (Type 2), no threshold is needed [15].

Model clone detection techniques have been developed for different types of models and domains, the most mature being Simulink [11], [15], [16]. More recently, however, model clone detection approaches have been emerging for other types of models such as Stateflow models [17], [18], UML models [19], [20], and others. Some model clone detectors employ a graph-based analysis approach, while others use a text-based approach, each having their own unique benefits and shortcomings [13].

1) Cross Clones: One extension to model clone detection is the notion of detecting "cross clones". Instead of executing model clone detection on a project and revealing model clone pairs and classes within that project in isolation, cross clone detection involves running a model clone detection tool on two or more projects and ascertaining what clones "cross", or intersect, between the projects. A non-model-based example of this can be seen in our previous work on DebCheck [21], which uses cross cloning to find licensing issues in code-based software systems. Specifically, DebCheck finds near-miss C function clones that cross between a given application's code and the Gnu-licensed Debian source distribution. This same idea can be applied to model clones, as we have proposed [10].

C. Design Patterns and Antipatterns

Design patterns provide tested and generic solutions for prevalent Software Engineering design problems and properties. Antipatterns are general representations of bad practices and properties that software engineers should avoid. Patterns have been utilized in a variety of domains for different purposes, including Java enterprise system design patterns [22], predicting performance [23], and finding design patterns in multi-agent systems [24]. In all cases, the patterns have a description of their applicable contexts and an abstraction of the solution, allowing it to be used in different situations.

1) Related Pattern Instance Detection Techniques: When it comes to assessing software quality, one generally accepted form of metrics is detecting the existence of design patterns and absence of antipatterns [26], [27]. Often, pattern detection involves textual analysis rules, such as Prolog, that examine systems at the source code level [8], [25], [28], [29], or design level [30]. Textual code level pattern detection is most

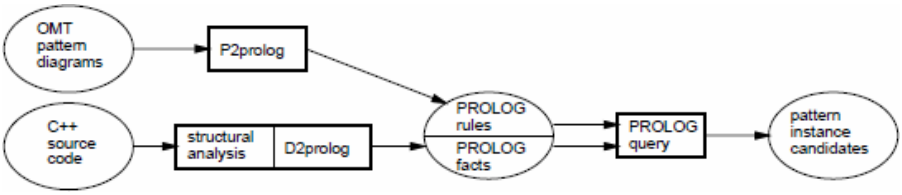

Fig. 2. Sample C++ Prolog Analyzer [25]

common. One textual approach involves extracting meta data from C++ classes and comparing it to meta data for known patterns [25], [28]. As illustrated in Figure 2, Kramer and Prechelt [25] accomplish this by converting patterns into Prolog rules and considering system properties, like C++ meta data, as Prolog facts. We include this figure as it is representative of the general process taken by many existing model pattern detection approaches and helps contrast them with what we do with SIMAID. In this example, they run Prolog queries using the pattern rules on the extracted system facts, and then present instance candidates to the user. Stoianov and Sora [8] also use Prolog rules to analyze code. These approaches differ from the research we present in this paper in that they use textual rules and work on source code.

The Mate Project [31] performs "guideline" checks using codified rules. In contrast to Prolog approaches, they use visual model analysis rules and activity diagrams to generate an implementation of their guideline rules in Java. This process performs textual analysis using those Java rules, while our approach and SIMAID tool are strictly model-based. A similar reverse engineering approach to detecting design patterns is realized by Tsantalis et al. [29]. They develop a matrix representing the pattern properties they are looking for and compare these matrices to the graph-based version of the code. Some notable shortcomings of this technique include "convergence of the similarity algorithm (depending) on the system graph size" and "the time needed for the calculation of similarity scores . . . can be prohibitive for large systems." We conducted similar research where we expressed Java Enterprise Edition (J2EE) framework implementations as framework specific models (FSM) [32]. We were able to detect antipattern instances by reverse engineering Java code, obtaining a model, and performing analysis on that model's properties. Unlike the approach in this paper, both these approaches require and reverse engineer the code, and include a textual codification of the patterns. In addition, the FSM approach works only with software frameworks.

Most closely related to our work is that of Wenzel and Kelter [33]. They also note the problems with translating patterns into "non-familiar formalisms", such as Prolog. Their approach employs model comparison by defining patterns as UML models and comparing those UML pattern models to systems of interest. They specifically evaluate characteristics unique to the patterns being sought after. This evaluation involves converting all the UML models to attributed type graphs and associating a weight to model-language specific properties, like having the same super class or sharing a package. While their approach is tailored to detecting design pattern occurrences, antipattern instance detection could be accomplished in the same manner. Similar to our work, this is an example of detection of model-based patterns using model comparison. However, it is intended for UML class models only, can experience performance problems with the heavily cyclic graphs that can appear in data-flow and behavioral models, and can be accessed only within the FUJABA development tool[2]. The approach we present in this paper uses the Simone model clone detector [15], which avoids cyclic graph and subgraph isomorphism issues and is not restricted to any particular tool environment. Simone reports its results as both an HTML web page and an XML database, and can be viewed directly in the Simulink environment using the SimNav interface [34]. While SIMAID works with Simulink specifically, the general technique we use for SIMAID is applicable to any modelling language for which a mature model clone detection technique exists.

Mapelsden et al. have devised a domain specific modelling language intended for design patterns called Design Pattern Modelling Language (DPML) [35]. Their language allows for linking design pattern solution elements to UML model elements in order to verify that design patterns have been adhered too. In contrast to our model clone detection based process and the work of Wenzel and Kelter, this also requires learning and using the "non-familiar formalism" that is their language. In addition, DPML is intended to be linked to the model elements at design time and is used more for specification and adherence, rather than pattern instance detection.

D. Identifying Model-Based Pattern Instances using Model Clone Detection

In our previous short paper we proposed a general strategy for detecting pattern instances using model clone detection [10] in order to improve model quality. The process, which is based on the same principles as DebCheck [21], is centered on cross cloning. Specifically, it begins by defining and modelling each pattern of interest as a concrete (sub) model, storing those model-based pattern representations as their own library, and performing model cross clone detection between the system being analyzed and the library of patterns. Model clone pairs between a pattern model and the system can then be viewed by a QA tester as an instance of that pattern in

[2]http://www.fujaba.de/

the system, and used to reason about properties of the system in model quality assessment.

1) Representation of Model Patterns: In a way similar to Wenzel and Kelter's [33] UML model comparison approach, concrete instances of the patterns of interest must be represented as (sub) models to be compared to the models in the system undergoing analysis. A key consideration is which model patterns are to be represented and how this will be done.

The question of which model patterns to include depends on what quality engineers are interested in and what is available in the research and industrial literature. Regarding the "how", many patterns are already defined as examples at the design/modelling level, such as those in Gamma et al.'s "Gang of Four" design pattern textbook [5] and other works on design patterns [36] and antipatterns [37]. In some cases patterns are presented as a set of explicit model instances or examples, and in other cases, as a single generalized or abstract representation of the pattern model. In both situations, all that is required is to represent these patterns and their defining properties in a suitable concrete model form so that the clone detection tool can identify clone pairs between pattern models and target models. We elaborate on this notion in our previous research [10].

2) Detecting Model Pattern Instances using Model Clone Detection: When it comes to detecting model pattern instances using model clone detection, earlier approaches that are capable of detecting Type 1 (exact) model clones only [11], [20] may not be useful in their current form. This is because target projects to be analyzed would have to have identical models, in both structure and type naming, to those in our model pattern library in order to be detected.

Model clone detectors capable of detecting Type 3 (near-miss) and Type 2 (renamed) model clones [15]–[17], [19] are suitable candidates. Detection of Type 2 model clones will allow engineers to detect simple structure-based model patterns that have identical structure to, but different element names than, the systems being evaluated. Type 3 model clone detection allows for more flexible and abstract definitions of model patterns. Any model clones identified as Type 3 clones to a model in pattern libraries are near-miss matches that indicate potential pattern instances that can be verified by the system analyst or domain expert. More discussion about model clone types, with hypothetical examples, can be found in our introductory paper [10].

III. Simulink Antipattern Instance Detector (SIMAID)

In order to validate our idea of using model clone detection for model pattern detection, we have developed a Simulink antipattern instance detector (SIMAID) that employs Simulink model clone detection. We chose this specific domain and function because (1) Simulink is one of the most widely used MDE methods in industry, (2) Simulink model clone detection methods are mature and reliable [11], [15], [16], [38], [39], and (3) our industrial partners are extremely interested in quality assurance for Simulink. SIMAID uses the publicly available Simulink model clone detection tool Simone [15], which is both mature and has been thoroughly evaluated [38], [40].

A. Simulink Antipatterns Considered

As yet there is no established corpus of Simulink antipatterns, and for the antipatterns that we were able to find, there are few clear antipattern definitions or models. However, one source of Simulink antipatterns we found very helpful was the preliminary list provided by Tran and Kreuz [41] based on their experiences at Daimler. In their list, they briefly describe in a sentence or two, without Simulink examples or models, antipatterns that are analogues of traditional code-based antipatterns. They simply enumerate the antipatterns and are concerned solely with refactoring techniques for them, rather than detecting instances in Simulink projects.

Another source of antipatterns is the MathWorks Automotive Advisory Board's (MAAB) modeling style guidelines [42], which we used to find good examples of Simulink antipatterns. The style guidelines were originally composed by Mathworks with Ford, Daimler Benz, and Toyota, and now has contributions from "many of the major automotive OEMs and suppliers". Not all of the guidelines are directly transferable to meaningful model antipatterns, for example those dealing with aesthetics or documentation, but there are a handful that immediately appeared ideal for model antipattern representation.

The following subsections enumerate the Simulink antipatterns we have currently implemented in SIMAID. In each case we extend the existing definitions for them, and provide examples of our model pattern representations. All of our Simulink antipattern models are publicly accessible at our website[3].

1) Primitive Obsession: The Primitive Obsession (PO) antipattern [41] exists when there are small subsystems in a Simulink project that encapsulate very simple or primitive operations. The problem with this is there is no need the impose a hierarchical subsystem structure for a system that has one or two simple blocks in it. Simulink subsystems are meant to "Keep functionally related blocks together" in order to simplify Simulink models [4].

In order to represent this Simulink antipattern, we created a set of subsystems each containing a single block with a sink and a source. Figure 3 presents four example PO antipattern models from our Simulink antipattern library. Each of them represents a subsystem suffering from PO and, given the right similarity threshold, can facilitate detection of subsystems that have very few, likely one to two, operations. When it came to deciding which blocks to use and how many subsystems to create in the PO pattern models, we simply relied on Simulink's own "Commonly Used Block" library. The blocks have Simulink default names, which will be ignored by any model clone detector capable of detecting Type 2 or 3 clones. In addition, our PO antipattern representations are such that

[3] http://www.users.miamioh.edu/stephamd/projects/simaid/

[4] http://www.mathworks.com/help/simulink/ug/creating-subsystems.html

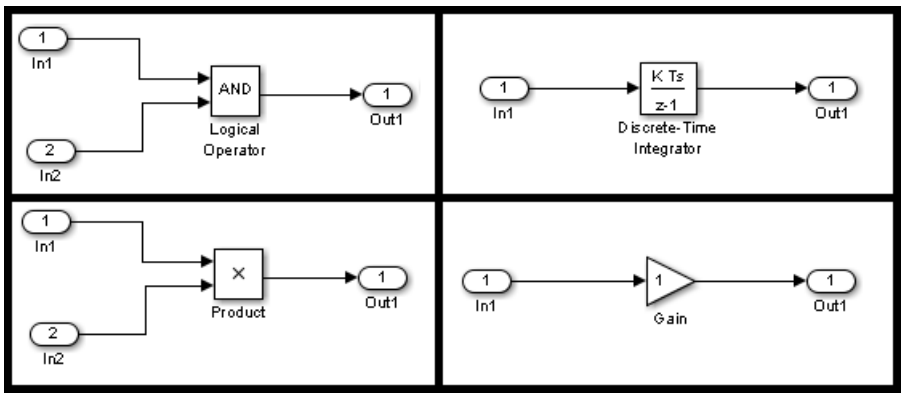

Fig. 3. Example Primitive Obsession (PO) Antipattern Models

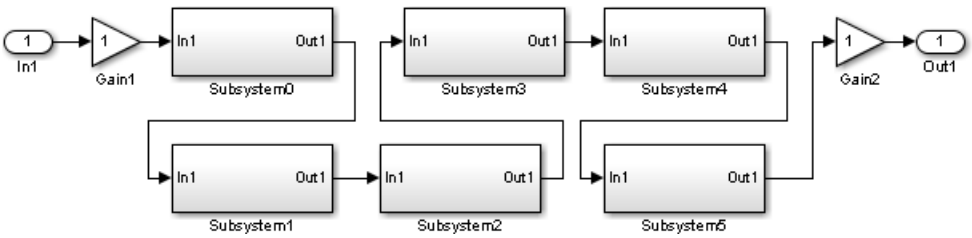

Fig. 4. Example Middle Man (MM) Antipattern Model with 2 Blocks and 6 Subsystems

Type 3 model clone pairs of systems with 2 function blocks are also identified.

2) Middle Man: The Middle Man (MM) Simulink antipattern describes a case where a (sub) system is doing too much delegation and not contributing significantly to the behavior of the overall system [41]. This is a problem because it adds to the complexity of the system hierarchy unnecessarily.

An identifiable symptom of an MM antipattern instance is having a system contain very few operational blocks and many subsystems. To model this antipattern, our Simulink antipattern library includes a model with a set of subsystems representing a variety of cases of blocks and subsystems. One example variant includes the one depicted in Figure 4. This subsystem contains two blocks and six subsystems and, with near-miss model clone detection, will catch similar and related antipattern instances. To complete the MM antipattern model set, we include systems covering the cross product of zero to three blocks and zero to six subsystems. We could have easily included more variations of these systems, but chose three blocks and six subsystems based on our Simulink experience and to demonstrate the feasibility of SIMAID. SIMAID could handle a larger cross-product set if the antipattern libraries included one.

3) Feature Envy: The Feature Envy (FE) Simulink antipattern refers to an instance where a subsystem is "more interested" in another subsystem than the one it is actually in [41]. It is similar to the Middle Man antipattern in that it will have very few blocks, but differs in that it is overly fixated on a single subsystem.

The properties of antipattern instances of those suffering from this antipattern are having very few blocks and having one or more calls/signals sent to a single (not many, like MM) subsystem. For brevity we have left out the FE model representation as it is similar to MM, except with a single subsystem.

4) Block/Signal Clumps: The Block/Signal Clumps (BSC) Simulink antipattern, analogous to the "Data Clumps" code smell [27], describes instances where a set of primitive blocks or signals appear often together throughout a Simulink system or project. Since these are not encapsulated, in either a bus or subsystem, this antipattern increases the numbers of ports and the sizes of subsystems.

Instances of this antipattern can be detected directly using model clone detection, and does not require cross cloning. Essentially, the definition of this antipattern corresponds with that of Type 2 model clones if there are many occurrences of them throughout a project. So, in SIMAID, we simply perform Simulink model clone detection looking for identical, but potentially renamed clones. Because types are factored in and not filtered out for the comparison by Simone [15], this is an ideal approach. In addition, we want to limit ourselves to roughly five or less blocks for it to be considered a BSC instance as the clumps are small in nature and more spread to permeation as a result.

We arbitrarily chose clone classes with five or more model clone instances to be those that are instances of the BSC antipattern and viable targets for subsystem creation. We could have easily chosen different numbers for our clump sizes and clone pair counts, but five seemed to make sense for our purposes.

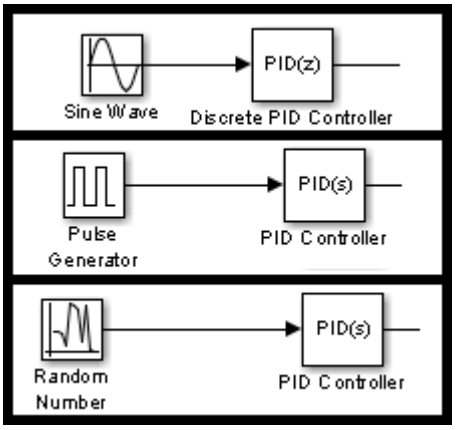

Fig. 5. Sample of JN_0001 Guideline Antipattern Models

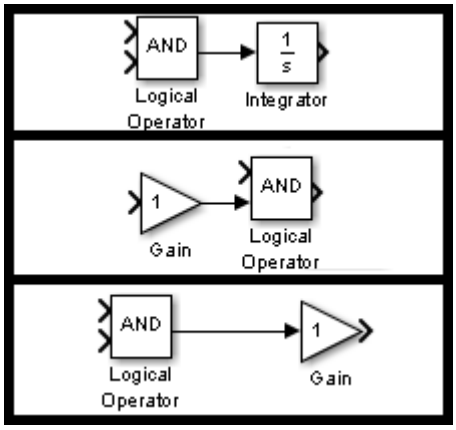

Fig. 6. Sample of NA_0002 Guideline Antipattern Models

5) MAAB Guidelines: There were three MAAB guidelines [42] that seemed appropriate to treat as antipatterns for quality evaluation in Simulink models. We outline these guidelines and provide some of our Simulink model-based representations of them.

a) Prohibited Simulink Blocks for use with Controllers: Guideline JM_0001 in the MAAB guidelines notes that control algorithm models must be designed from discrete blocks. So, the Simulink modeling implication of this is there is a list of sources that should not be used in conjunction with control blocks. Figure 5 provides three of our model antipattern examples including three discouraged sources and various PID controller blocks.

b) Mismatch of Logical and Numerical Operations: This MAAB guideline, NA_0002, indicates that blocks performing numerical operations must not be used in conjunction with logic blocks. The two manifestations of this guideline that we model include (1) numerical output should not be connected as input to logic blocks and (2) logical output should never be directly connected to numerical blocks. We once again took the cross-product of the logic block, Logical Operator, with the common numerical Simulink blocks, like gain, product, sum, and others. Figure 6 illustrates three examples of our antipattern model representations including one numerical operator as a source to logic block and two examples of a logical operator as input to a numerical block.

c) Number of Inputs to Variant Subsystems: Variant subsystems allow different and dynamic signal paths to be executed at run time. A variant subsystem has multiple subsystems and one set of in ports and one set of out ports. What makes a variant subsystem unique is those ports are not connected until the Simulation begins. As such, it is possible for subsystems to have a different number of in ports than their containing variant subsystem. The MAAB guideline NA_0020 says this is fine as long as the unused inputs are terminated using the terminator block. Our antipattern Simulink representations model the case where they are not terminated. Specifically, we model the cases covering the cross product of variant subsystems with one to four inports and contained subsystems with one to five inports. For example, we have a set of four variant subsystems with four inports and inner subsystems with inports of one, two, three, and five. Beyond that, these antipattern models are quite straightforward not necessitating an example.

B. Overview of the SIMAID Process

The first step in the SIMAID process involves modelling the Simulink antipatterns and storing them in a library. We outlined this step in the previous subsection. When ready, this library and the projects undergoing model quality assurance must be placed in an appropriate location such that they can be subject to model clone detection. So, in our case, we simply placed the folders containing the antipattern library alongside each respective project to analyze in the same higher-level containing folder.

Once the library and project are ready, Simone has to be configured for antipattern detection. We established from evaluation [38] and consultation with our industrial partners a near-miss threshold between 20% and 30% yields the most applicable clones. We require near-miss flexibility for antipattern instance detection since they will rarely, if ever, be exact matches, but also have some smaller patterns. Thus, for all antipatterns but BSC, we configure Simone to use blind renaming, with a 25% difference threshold. For BSC clones, we set Simone to use blind renaming, but with a 0% difference threshold. This will give us clumps of blocks, allowing for changes in names, but not types. We were able to utilize Simone as is, without changing its implementation.

The last step in the SIMAID process involves extracting the antipattern instances. At this point, we accomplish this by simply noting any clones present in cross-clone detection, that is, those that intersect our antipattern libraries and the projects undergoing evaluation. The exception to this is for BSC clones where we look at the Type 2 model clone classes and note any model clone instances contained in classes with a size of five or more clones. In the future, we will work on fully automating this process to present this information to the model QA tester

TABLE I
PUBLIC SIMULINK PROJECTS

Project	# of Simulink Systems
Simulink Demonstration Examples (SDE)	1852
Matlab Central (MC) - 5 Projects	723
ConQAT Example Models (CM)	638
Advanced Vehicle Simulator (AVS) Version 3	1876

directly in the native Simulink environment using the SimNav interface [34].

IV. SIMAID CASE STUDY

In order to validate our SIMAID process, and the idea of using model clone detection to detect model patterns in general, we performed a case study using SIMAID to detect Simulink antipatterns in four public Simulink projects. We make these projects publicly available on the same web page as our antipattern models[3]. The only exception to this is the Simulink example demonstration set, which are public, but are owned by Mathworks and available for download on their website[5].

Table I lists these projects and the number of Simulink systems they are composed of. The first project includes the set of all the Simulink example demonstration models, including aerospace, automotive, and other types of models. The second project is a collection of five Simulink systems from Matlab Central, which is an online repository of Matlab artifacts. The third project is one consisting of open-source models provided to us by the developers of ConQAT [11] that they had used for evaluation in the past. That set includes systems related to "Aircraft Library Source" and "Surveillance System Design Source". The last system is the third version of a large open-source Simulink application called Advanced Vehicle Simulator, called Advisor, which allows for simulation of various properties found in automobiles. From a run time performance perspective, the tool performed nearly identically on the projects in the case study to running basic Simone on them. That is, running SIMAID did not significantly impact the run time of the model clone analysis.

Table II summarizes the results from our case study on using SIMAID to perform antipattern instance detection on each respective Simulink project. Each row presents the findings of SIMAID when analyzing the four systems presented in Table I. Each column indicates the number of Simulink antipattern instances detected for our antipatterns under consideration. For PO anti pattern clone pairs, there was a lot of overlap (similarity) in our model representations because they were such simple systems. As such, we divided the number of PO clone pairs by a reduction factor of 2.5 to better represent the recall. This factor was based on our experience with two of the smaller open source systems: MC and CM. Unfortunately, none of our analysis of the projects revealed any MAAB antipattern instances, leaving us unable to conclude anything about SIMAID's ability to detect occurrences of those antipatterns. In each of the following subsections, we discuss the results for each project and present example model antipattern instances we found interesting.

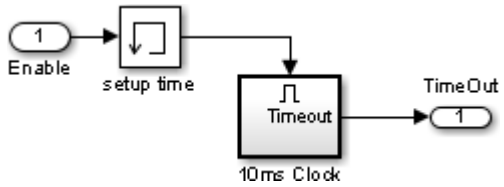

Fig. 7. Feature Envy Antipattern Instance found in SDE

A. Simulink Demonstration Examples

SIMAID discovered a nice range of different antipattern occurrences in the SDE systems. Despite being a smaller project than the Advanced Vehicle Simulator, this project has a higher number of primitive obsession antipattern instances. The implication of this from a model quality perspective is that this project has a lot of simple subsystems that should be refactored into larger ones. Because of the larger amount of primitive systems, this project, as a whole, has a lot of unnecessary hierarchy and would be very difficult to navigate through manually.

Our analysis of this project also yielded many Middle Man and Feature Envy antipattern instances. One example of the latter is presented in Figure 7, displaying the subsystem "10ms Recurring H/W Timer". Here we see the system is "fixated" on the subsystem "10ms Clock" and adds to the functionality only by adding one integration step delay for "setup time". Even though there is that additional delay, it is clear that this system really just performs the clock operation.

B. Matlab Central

The Matlab Central project had a large number of Block/Signal Clump instances relative to its size: 73 instances spread throughout 723 systems. One such clump, illustrated in Figure 8 contained 6 blocks that appeared together 20 times in different places. Seeing as BSC are Type 2 clones, this means the types of clones and structure were identical, but the elements may have been renamed. From a model quality assurance perspective, it would be in the interest of the QA team to create a subsystem in a library containing those blocks and replacing the 20 instances with a call to that subsystem. Simulink even has a built-in feature to do it automatically in one step in the Simulink UI simply by selecting a group of blocks.[6].

C. ConQAT Example Models

According to our analysis with SIMAID, the ConQAT example models, containing aircraft library and surveillance design systems, were relatively free of antipattern instances. There were a notable number of block clumps that could be refactored into libraries. In addition, there

[5]http://www.mathworks.com/products/simulink/model-examples.html

[6]mathworks.com/help/simulink/ug/creating-subsystems.html#f4-7371

TABLE II
SIMULINK ANTIPATTERN INSTANCES DETECTED USING SIMAID

Antipattern / Project	Middle Man	Primitive Obsession	Feature Envy	Block/Signal Clumps	Guideline
Simulink Demonstration Examples	28	170	10	7 Clumps; 83 Instances	0
Matlab Central	0	23	0	5 Clumps; 73 Instances	0
ConQAT Models	0	9	0	6 Clumps; 44 Instances	0
Advanced Vehicle Simulator	1	149	6	19 Clumps; 198 Instances	0

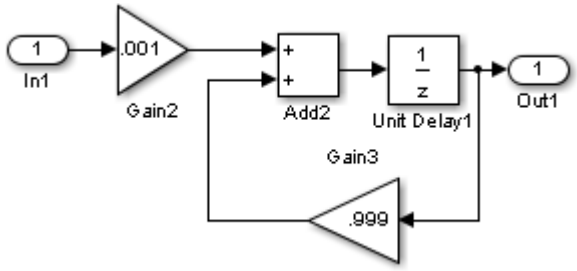

Fig. 8. Example of Block Clump with 20 Instances in MC

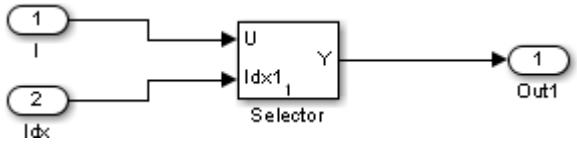

Fig. 9. Primitive Obsession Antipattern Instance from CQ

were a handful of primitive obsession antipattern instances. One such example is presented in Figure 9. Here we see a subsystem, called "Enabled Subsystem" within the "Pre_SysGen_Post_sh_fifo_nohwcs" model, that is "obsessed" with a primitive operation. Specifically, it has only a single selector block. A selector block is a simple Simulink block that takes in an input vector, matrix, or signal, and outputs selected elements[7]. Model quality can be improved immediately by refactoring that single block to the upper level.

D. Advanced Vehicle Simulator

Figure 10 presents the lone Middle Man antipattern instance found in the Advanced Vehicle Simulator. The system, "electric acc loads <acc>" in the model "BD_PAR_SA_AUTO" has no real purpose other than acting as a Middle Man to the subsystem handling electrical accessory loads, the lightning-bolt subsystem in the diagram. This Middle Man antipattern indicates a case where this system has no purpose other than passing information to another system and unnecessarily complicates the hierarchy of the Simulink AVS project.

Our analysis shows that this system has an inordinate number of Block/Signal Clumps: 19 classes totally 198 instances. This model quality metric found during our quality assurance analysis indicates that this system has serious opportunity for refactoring and that there is a ton of repeated block clumps that should be extracted into subsystems or buses.

[7]http://www.mathworks.com/help/simulink/slref/selector.html

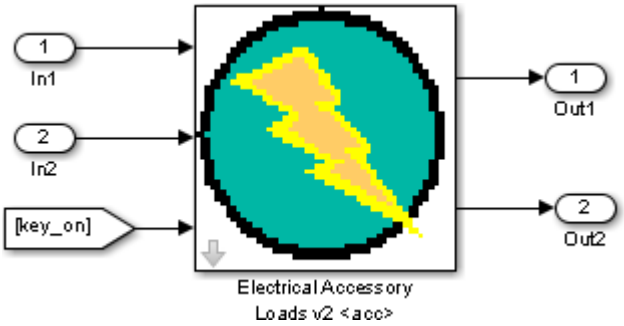

Fig. 10. Middle Man Antipattern Instance found in AVS

V. DISCUSSION AND FUTURE WORK

When it comes to detection tools, an important and obvious question is that of recall and precision, that is, the ability to account for all theoretical instances and the accuracy of the result set, respectively. From a model clone detection perspective, SIMAID is built on Simone. In the past, we evaluated Simone's recall and precision [38], [40] and found it to have a recall in the mid-to-high nineties and a precision in the high nineties, with both values varying depending on the similarity threshold. So, the overall question of recall and precision for SIMAID is not related to model clone detection in itself, but rather its ability to detect antipattern instances. The run time performance of SIMAID is dependent on the run time performance of Simone, which is quite efficient as it uses text analysis and transformation technologies [15].

Pettersson et al. [43] point out the difficulties in evaluating pattern instance detectors. One problem is that there is typically no "Gold Standard" for accuracy metrics that can be used, or it takes a long time to construct correctly and completely. Similar to our problems in constructing abstract representations, potential over- and under- approximations can occur, meaning the representations are too restrictive or not restrictive enough, respectively. Another related issue is the actual measure for precision and recall. They note "For pattern detection applications involving human clients, ... one would tolerate checking a list of pattern candidates manually if one could be sure that the list contains all pattern occurrences." and also provide a weighted harmonic mean of precision and recall. Because this needs to be done manually and by someone qualified, our long-term plan involves enlisting an experienced Simulink domain engineer and having them validate our detected antipattern instances. In addition, they could assist us in refining our model representations.

We selected Simulink model antipattern detection because antipatterns were more available than design patterns and we considered them instantly applicable to model quality since refactoring them immediately improves system quality. In contrast, for Simulink model design patterns, we would first have to determine a lack of design patterns and additionally detect places where design patterns can be exploited. In terms of the effectiveness of detecting the presence of antipatterns in judging model quality, that is something we are investigating as future work. We believe that it will be just as effective as it has been in the analogous code domain [26], [27]. Important questions to answer on that front include how "bad" each antipattern is compared to other ones when it comes to overall model quality for a project. Some of that may be able to be calculated by looking at the connections and impact factor of the systems containing antipattern instances. So, if an antipattern is a more "severe" one and is in a "key" system, then that would have a larger quality impact value.

There are a number of threats to validity to our case study. Firstly, as pointed out by Pettersson et al. [43], choice of projects is always a factor when evaluating detection tools. For example, it is unfortunate that either none of our projects contained MAAB guideline violations or SIMAID failed to detect them, and we are unable to discern which it is. Because of the accuracy of Simone and the relative simplicity of the MAAB guidelines we modelled, we believe it is the former. Similarly, we had a limited set of antipatterns and the ones we had were relatively simple. In regards to our overall objectives however, these publicly available projects and this set of antipatterns allowed us to illustrate detection of Simulink antipattern instances. Another threat to our case study is that we semi-manually counted and extracted the antipattern instances. While we were very careful and used automated string search tools to guide us, any manual interaction always introduces the possibility of error. For future work, we plan on automating the extraction of the SIMAID results into an easy to read form and, ideally, one that is natural and integrated into the model engineers' development and quality assurance environments. This will include us consulting with Simulink engineers from our industrial partners, and may, for example, include presenting results using Simone's SimNav interface [34].

Furthermore, we will be focusing on applying our SIMAID technique in industrial environments through collaboration with our partners. Seeing as they already use Simulink in much of their development, we want to introduce SIMAID to their test engineers, refine it for their purposes, and help them use it to further their quality assurance measures. Through observation and interaction, we can continue to evolve SIMAID as a model quality assurance technique. In terms of Simulink antipattern selection criteria for further research, we will be discussing our existing corpus with our industrial partners and also investigating what other ones are of known and of interest to modellers.

Lastly, our continual future work involves improving both the quantity and quality of Simulink Antipatterns covered by SIMAID. There is limited research on Simulink antipatterns, and Simulink patterns in general. In this research, we did our best to leverage what is out there, extracting the antipatterns from both Tran and Kreuz's work [41] and the MAAB guidelines [42] that fit well with our proposed approach. While we considered all the antipatterns from both those sources, the antipatterns we chose were relatively low-hanging fruit. It may be possible to encode more antipatterns from those sources, and we hope that publicizing our work and future collaborations with Simulink experts will help increase the quantity of Simulink antipatterns. In addition, we will continue to look for additional sources of Simulink antipatterns, like forums, and even considering transforming antipatterns from other related domains, like Modelica [44], to Simulink.

VI. Conclusion

Improving model quality assurance is a crucial step in ensuring the continued growth and success of Model-Driven Engineering. As model artifacts permeate through all phases of the Software Engineering development process and mature over time, the need for analyzing and evaluating properties of these models becomes increasingly important. One established approach to assessing quality in software projects in general is to detect known design solutions, or patterns, and poor solutions, or antipatterns, to common problems. Traditionally, this has been done by either evaluating source code or using textual rules to evaluate design models. Recently, we proposed the idea of detecting model patterns, at the model level, by using model representations and performing a type of model comparison known as model clone detection. This has the benefit of (1) allowing earlier analysis in product creation, (2) being able to be used on projects that are built mainly or entirely using model-driven practices, and (3) allowing testers to maintain the same level of abstraction between the patterns and the projects undergoing quality assurance.

In this paper, we validated this idea by developing a Simulink antipattern instance detector, named SIMAID. Using existing corpora of Simulink antipatterns, we considered four antipatterns and three guidelines to include within SIMAID. SIMAID involves representing the Simulink antipatterns as Simulink models and placing them in libraries. These libraries and desired projects then undergo cross-clone detection using our near-miss model clone detector, Simone. We evaluated SIMAID through a case study targeted at four publicly available projects of varying natures and sizes, finding instances of the antipatterns throughout all the projects. No instances of MAAB Simulink guideline violations were detected in any of our projects.

Better calculating recall and precision for SIMAID's antipattern detection is one area of future work we are pursuing. Precision and recall for antipattern detection is best achieved through manual interaction with domain experts and is an avenue we will explore. Other future work includes providing SIMAID's data in a form conducive to engineers by seeking feedback from our industrial partners on their desired quality assurance capabilities they want to see from the tool. As

research and our industrial collaborations continue in this area, we plan on improving the quality and quantity of Simulink antipatterns within SIMAID's scope.

It is our belief that research on model-based pattern detection has much promise in the area of model quality assurance and analysis. Through SIMAID, we have improved Simulink model quality evaluation by allowing analysts the ability to detect "bad" system properties within their Simulink projects. We hope that this gives way to more advances in the field of Model-Driven Engineering, not just for Simulink but for modelling across the board.

ACKNOWLEDGMENT

This work is supported in part by 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.

REFERENCES

[1] M. Volter, T. Stahl, J. Bettin, A. Haase, and S. Helsen, *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.

[2] P. Mohagheghi and J. Aagedal, "Evaluating quality in model-driven engineering," in *International Workshop on Modeling in Software Engineering*, 2007, p. 6 pp.

[3] T. Punter, J. Voeten, and J. Huang, "Quality of model driven engineering," *Model-Driven Software Development: Integrating Quality Assurance*, 2009.

[4] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *2007 Future of Software Engineering*, 2007, pp. 37–54.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[6] W. J. Brown, H. W. McCormick, T. J. Mowbray, and R. C. Malveau, "Antipatterns: refactoring software, architectures, and projects in crisis," 1998.

[7] J. K. Mak, C. S. Choy, and D. P. Lun, "Precise modeling of design patterns in UML," in *International Conference on Software Engineering*, 2004, pp. 252–261.

[8] A. Stoianov and I. Sora, "Detecting patterns and antipatterns in software using Prolog rules," in *International Joint Conference on Computational Cybernetics and Technical Informatics*, 2010, pp. 253–258.

[9] D. Ballis, A. Baruzzo, and M. Comini, "A rule-based method to match software patterns against UML models," *Electronic Notes in Theoretical Computer Science*, vol. 219, pp. 51–66, 2008.

[10] M. Stephan and J. R. Cordy, "Identifying instances of model design patterns and antipatterns using model clone detection," in *International Workshop on Modelling in Software Engineering*, 2015, p. 6 pp., to appear.

[11] F. Deissenboeck, B. Hummel, E. Jurgens, B. Schatz, S. Wagner, J. Girard, and S. Teuchert, "Clone detection in automotive model-based development," in *ICSE*, 2009, pp. 603–612.

[12] M. Stephan and J. R. Cordy, "A survey of methods and applications of model comparison," Queen's University, Tech. Rep. 2011-582, 2012.

[13] ——, "A survey of model comparison approaches and applications," in *International Conference on Model-Driven Engineering and Software Development*, 2013, pp. 265–277.

[14] C.-M. Ong, *Dynamic simulation of electric machinery: using MATLAB/SIMULINK*. Prentice Hall, 1998, vol. 5.

[15] 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 *International Conference on Software Maintenance*, 2012, pp. 295–304.

[16] 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 *ICSE*, 2009, pp. 276–286.

[17] T. R. Dean, J. Chen, and M. H. Alalfi, "Clone detection in Matlab Stateflow models," *Electronic Communications of the EASST*, vol. 63, 2014.

[18] M. A. Kumar, "Efficient weight assignment method for detection of clones in state flow diagrams," *Journal of Software Engineering Research and Practices*, vol. 4, no. 2, pp. 12–16, 2014.

[19] E. Antony, M. H. Alalfi, and J. R. Cordy, "An Approach to Clone Detection in Behavioural Models," in *International Working Conference in Reverse Engineering*, 2013, pp. 472–476.

[20] H. Storrle, "Towards clone detection in UML domain models," *Software & Systems Modeling*, vol. 12, no. 2, pp. 307–329, 2013.

[21] J. R. Cordy and C. K. Roy, "Debcheck: Efficient checking for open source code clones in software systems," in *International Conference on Program Comprehension*, 2011, pp. 217–218.

[22] W. Crawford and J. Kaplan, *J2EE design patterns*. O'Reilly Media, Inc., 2003.

[23] A. I. Verkamo, J. Gustafsson, L. Nenonen, and J. Paakki, "Design patterns in performance prediction," in *Workshop on Software and Performance*, vol. 2000, 2000, pp. 143–144.

[24] S. Sauvage, "Design patterns for multiagent systems design," in *MICAI: Advances in Artificial Intelligence*, 2004, pp. 352–361.

[25] C. Kramer and L. Prechelt, "Design recovery by automated search for structural design patterns in object-oriented software," in *Working Conference on Reverse Engineering, 1996*, 1996, pp. 208–215.

[26] B. Huston, "The effects of design pattern application on metric scores," *Journal of Systems and Software*, vol. 58, no. 3, pp. 261–269, 2001.

[27] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Working Conference on Reverse Engineering 2002*, 2002, pp. 97–106.

[28] M. Vokac, "An efficient tool for recovering design patterns from C++ code." *Journal of Object Technology*, vol. 5, no. 1, pp. 139–157, 2006.

[29] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *Transactions on Software Engineering*, vol. 32, no. 11, pp. 896–909, 2006.

[30] F. Bergenti and A. Poggi, "Improving UML designs using automatic design pattern detection," in *International Conference on Software Engineering and Knowledge Engineering*, 2000, pp. 336–343.

[31] I. Sturmer, I. Kreuz, W. Schafer, and A. Schurr, "The MATE approach: Enhanced Simulink and Stateflow model transformation," in *MathWorks Automotive Conference*, 2007.

[32] M. Stephan, "Detection of Java EE EJB antipattern instances using framework-specific models," Master's thesis, University of Waterloo, 2009.

[33] S. Wenzel and U. Kelter, "Model-driven design pattern detection using difference calculation," in *Workshop on Pattern Detection for Reverse Engineering*, 2006.

[34] J. R. Cordy, "Submodel pattern extraction for simulink models," in *International Software Product Line Conference*, 2013, pp. 7–10.

[35] D. Mapelsden, J. Hosking, and J. Grundy, "Design pattern modelling and instantiation using dpml," in *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, 2002, pp. 3–11.

[36] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[37] B. Dudney, S. Asbury, J. K. Krozak, and K. Wittkopf, *J2EE antipatterns*. John Wiley & Sons, 2003.

[38] M. Stephan and J. R. Cordy, "Model clone detector evaluation using mutation analysis," in *International Conference on Software Maintenance and Evolution*, 2014, pp. 633–638.

[39] H. Petersen, "Clone detection in Matlab Simulink models," Master's thesis, Technical University of Denmark, 2012, iMM-M. Sc.-2012-02, 2012.

[40] M. Stephan, "A mutation analysis based model clone detector evaluation framework," Ph.D. dissertation, Queen's University, 2014.

[41] Q. Minh Tran and I. Kreuz, "Refactoring of Simulink models," in *MathWorks Automotive Conference, Stuttgart*, 2012.

[42] MathWorks Automotive Advisory Board, *Control Algorithm Modeling Guidelines using MatLab, Simulink, and Stateflow Version 3.0*. Mathworks Inc., 2012, http://www.mathworks.com/solutions/automotive/standards/maab.html.

[43] N. Pettersson, W. Lowe, and J. Nivre, "Evaluation of accuracy in design pattern occurrence detection," *Transactions on Software Engineering*, vol. 36, no. 4, pp. 575–590, 2010.

[44] M. M. Tiller, "Patterns and anti-patterns in modelica," in *Proceedings of 6th International Modelica Conference*, 2008, pp. 647–656.