Identifying Instances of Model Design Patterns and Antipatterns Using Model Clone Detection

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

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

***Abstract*—A hurdle in the growth of model driven software engineering is our ability to evaluate the quality of models automatically. One perspective is that software quality is a function of the existence, or lack thereof, of good and bad properties, also known as patterns and antipatterns, respectively. In this paper, we introduce the notion of using model clone detection to detect model pattern and antipattern instances by looking for models that are cross clones of pattern models. By detecting patterns at the model level, analysis is accomplished earlier in the engineering process, can be applied to primarily model-based projects, and remains at the same level of abstraction that engineers are used to. We outline the process of using model clone detection for this purpose, including representing the patterns and detection of instances. We present some Simulink examples of pattern representations and discuss future work and research in the area.**

I. Introduction

Model-driven Engineering (MDE) is a relatively new approach to software development that entails higher-level abstractions, or models, being used as the primary artifacts in system development and management. This includes incorporating modeling in all four phases of Software Engineering: requirements, design, implementation, and testing. By utilizing artifacts at an abstraction level closer to the problem space and hiding lower-level details, MDE can yield higher quality systems, facilitate better communication among stakeholders, and make project teams adaptable and flexible. MDE has seen significant levels of adoption in many different domains, including aviation, automotive, aerospace, and other high-reliability embedded systems applications.

While MDE use is growing, and engineers and customers are starting to experience the benefits of using model-based techniques [1], there are still many obstacles to overcome. One example is model quality evaluation, that is, the ability to assess quality of the models and artifacts of interest across the MDE life cycle [2], [3]. When it comes to more traditional software engineering processes, such as those that center around third generation programming languages, quality assurance is a well-researched and established area. In contrast, much less is understood about quality assurance for models in the MDE context. Improving our knowledge and making it easier to reason about model quality is a necessary step in continuing the growth of MDE.

One approach to assessing software quality involves detecting the existence of design patterns [4] and antipatterns [5] in projects. Design patterns and antipatterns describe good and bad, respectively, ways of solving specific design questions in software system development. Detecting them allows for system analysts to automatically, or semi-automatically, identify the presence or absence of desired and undesired properties in their projects. They are typically based on pragmatic experience and involve descriptions of when and where the pattern/antipattern should be applied, implementation variants, and justification. Often these patterns have an abstract representation in model form [6], however, detection is generally accomplished by evaluating code [7] or, in some cases, by writing complex textual rules to evaluate models [8].

In this paper we argue that detection of pattern and antipattern instances in models can be achieved using a specialized form of model comparison [9] known as model clone detection [10]. Model comparison involves comparing models and explicating both similarities and differences. Model clone detection (MCD) involves identifying sets of models that are similar within a given difference threshold. Traditional pattern-detection approaches can be cumbersome, requiring system analysts to switch between abstraction levels, and to delay pattern and antipattern detection until late in the engineering process. By using model clone detection, detection of model patterns and antipatterns (1) can be done earlier in the software life cycle, (2) can be applied to projects that are primarily or purely model-based, and (3) stays at the same (model) level of abstraction that the patterns and systems are defined in.

This paper presents our initial ideas on using MCD for realizing detection of model patterns and antipatterns. Specifically, we look at existing MCD approaches and how they can be employed to detect patterns and antipatterns. Ideally, the notions discussed in this paper will inspire further research into using MCD or MCD-like approaches to improve the quality of MDE software through direct detection of patterns and antipatterns in models.

We begin by providing background information and related work in the areas of model comparison, model clone detection, and design patterns and antipatterns in Section II. We outline the process of using MCD for the purposes of model design pattern and antipattern detection in Section III, using Simulink antipatterns as an example. Section IV highlights some future work and research areas that are within our sights. The paper is then concluded in Section V.

II. Background and Related Work

A. Model Comparison and Model Clone Detection

Model comparison involves comparing and contrasting two or more models in order to identify similarities and differences. There are many tools and techniques for accomplishing model comparison [11], but not all are well suited to assessing model quality through model pattern detection.

For the purposes of this paper, we are concerned with the first phase of model comparison, calculation. Calculation consists of analyzing models to retrieve the information desired by a particular comparison algorithm. For example, ascertaining which elements are present in one model that are not present in another, and vice versa. Kolovos et. al identify four different categories of calculation, or matching, in model comparison approaches [12]. The one most relevant and useful for detecting model-based patterns in MDE projects is similarity-based matching, which is accomplished by composing the similarity of an element's features with another element in another model. The other categories of calculation would not be ideal for model pattern detection, as they rely on unique element identifiers, static properties/features of models, and/or user pre-calculation configuration.

An example family of similarity-based matching model comparison approaches that is on the rise is model clone detection (MCD) [10]. MCD involves identifying clone pairs or classes of models within an MDE project that are similar up to a specific threshold. There are generally three types of model clones[13]: Type 1, or *exact* clones, Type 2, or *renamed* clones, and Type 3, or *near-miss* clones. MCD can applied to various kinds of models and domains, with the most mature MCD target being Simulink data-flow models [10], [13], [14]. However, recently, there has been advancements in MCD for other model types including various types of UML models [15], [16], Stateflow models [17], [18], and others. Similarly to model comparison approaches, MCD can be realized using a variety of graph-based and text-based approaches, each with their own unique advantages and disadvantages [11].

B. Design Patterns and Antipatterns

Design patterns express generic solutions for common problems or properties in software engineering and design, while anti-patterns encode generic examples of bad practices or properties that we want to avoid. Both can be used in many domains for many different purposes, such as performance prediction [19], multi-agent systems [20], Java enterprise systems [21], and many more. Each pattern or antipattern includes a problem description and an abstraction of the solution that allows it to be reused in various settings.

It is also generally accepted that the existence of patterns and absence of antipatterns can be used as metrics in evaluating the quality of a system [22]. Techniques for pattern and antipattern detection often involve textual rules, for example in Prolog, that investigate systems at either the code level [23], [7], [24], [25] or the design level [26].

Consider the template design pattern [4] in Figure 1. This pattern involves describing an algorithm in a more general

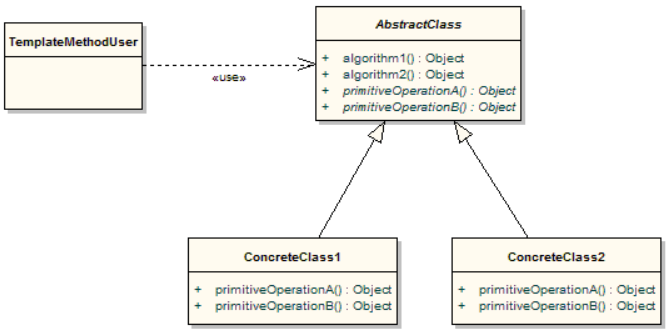
TemplateMethodUser
«use»
AbstractClass
+ algorithm1() : Object
+ algorithm2() : Object
+ primitiveOperationA() : Object
+ primitiveOperationB() : Object
ConcreteClass1
+ primitiveOperationA() : Object
+ primitiveOperationB() : Object
ConcreteClass2
+ primitiveOperationA() : Object
+ primitiveOperationB() : Object

Fig. 1. Template Design Pattern [4]

way, while deferring some specific aspects of the algorithm to subclasses or 'implementers'. As is done in many cases, this design pattern is described and represented at the model level. However, detection is most often done at the textual code level. Some approaches [23], [24] extract textual metadata from C++ classes and compare that metadata to the expected metadata for the template design pattern. One architecture for accomplishing this is presented in Figure 2. In this example, Kramer and Prechelt [23] convert the design patterns into Prolog rules and consider the C++ metadata as Prolog facts. The rules are run as Prolog queries on the extracted facts and instance candidates are presented to the user. Stoianov and Sora [7] also use Prolog rules to analyze code and compare the result to the expected design pattern. These approaches differ from what we propose in this paper in that they use textual rules and work on source code.

Another technique that performs "guideline" checks using codified rule analysis is the Mate Project [27]. Rather than using Prolog rules, they use visual model analysis rules and UML activity diagrams to control the application of the analysis to generate a Java implementation of the rules. Again, this process involves textual analysis, whereas what we are proposing in this paper is strictly model-based.

To detect design patterns, Tsantalis et al. [25] reverse engineer code-based projects and develop a matrix representing the properties they are looking for. These matrices are compared to a graph-based version of the code. They note that the "convergence of the similarity algorithm depends on the system graph size" and that "the time needed for the calculation of similarity scores . . . can be prohibitive for large systems." Similarly, we conducted research using framework-specific models [28] where we expressed Java Enterprise Edition (J2EE) antipatterns as a framework specific model (FSM) and checked for antipattern instances in specific J2EE framework instances obtained through reverse engineering code to obtain a model. In contrast to what we propose in this paper, both these approaches reverse engineer the code, and use a textual codification of the patterns or antipatterns. The FSM approach works only with software frameworks.

The work of Wenzel and Kelter [29] relates most closely to the techniques we propose in this paper. They also highlight the issues with the translation of patterns into "non-familiar formalisms" such as Prolog. To combat this, they employ a form of similarity-based model comparison by defining

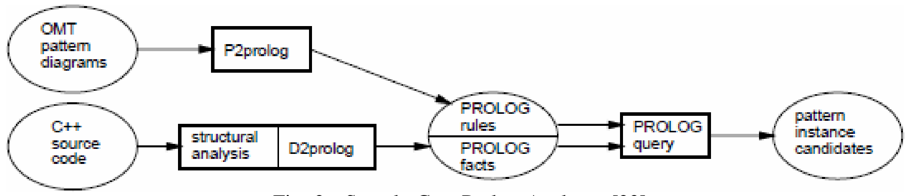

Fig. 2. Sample C++ Prolog Analyzer [23]

patterns as UML models and comparing those to existing models, comparing the characteristics specific to the pattern being detected. To do this, they convert all the models to attributed type graphs and associate a weight to specific model type properties, such as sharing the same package or super class. They do not address antipatterns, but it is likely their approach would still apply. While this work is an example of detection of model-based patterns through model comparison, it is defined for use on UML class models only; can see notable performance hits when encountering heavily cyclic graphs, which can appear in data-flow and behavioral models; and can be accessed only within the FUJABA development tool[1]. The MCD approaches we discuss in this paper have been demonstrated for a variety of model types, can avoid cyclic graph and sub-graph isomorphism issues [13], and can have their results 'reported' in a variety of formats to be used by analysts using different tools.

For the remainder of the paper we use the terms *pattern* and *model pattern* to refer to both model design patterns and antipatterns, unless explicitly indicated.

III. Using MCD for Model Design Pattern and Antipattern Detection

Our proposed process involves defining and representing each pattern of interest as a model or set of models, storing those patterns in their own project and running a MCD tool to detect cross clones between that pattern project and the projects of interest. This follows a similar framework to our DebCheck [30] tool, which uses cross cloning to find licensing issues in code-based software systems. For our modelling purposes, this can be accomplished using an appropriate and validated similarity threshold and viewing cross-clones of each respective pattern model as instances of that pattern. This section summarizes that process and gives examples of Simulink antipatterns and how they can be represented in model form.

A. Design Pattern and Antipattern Model Representation

Similarly to Wenzel and Kelter's [29] model comparison method, using MCD to assess quality through model pattern detection can be realized by expressing the patterns or antipatterns as a model to be compared to other models.

In order for this notion to be feasible, we first must consider how the patterns will be represented. To do so, we require a corpus of model representations for known patterns. This is quite practical, as many patterns are defined and can be identified at the design/modeling level. Some patterns have high-level general model definitions that cover all cases of the respective pattern, while others are defined in an example-driven manner. In the case of the former, a single, but representative, instance of that general model can treated as the model to compare to the system's models and used for detection and quality evaluation.

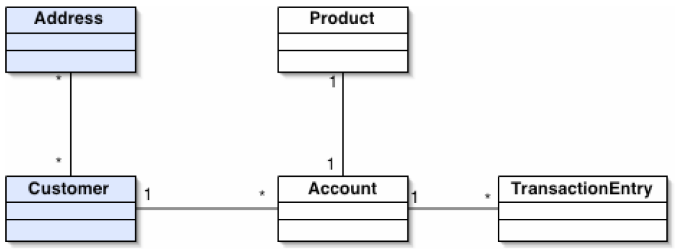

Fig. 3. Example of the Dredge Antipattern [31]

Consider the patterns defined in Gamma et al.'s design pattern book [4]. In many cases patterns are presented using a set of model instances rather than a a single generalized or abstract model pattern. All that is required is to represent these instances and their defining properties in the appropriate model form for the specific MCD tool. In cases where the patterns are defined using example models rather than a higher-level general model, most of the work is already done.

For example, consider the "Dredge" J2EE antipattern [31] in Figure 3. The general form of this antipattern involves a long list of Enterprise Java Bean (EJB) entities contained within deep graphs. The problem with this is that there will be many database fetches. Figure 3 contains an example model adhering to the general description of this antipattern, where we have many EJB entities with relationships to others. The refactoring of this antipattern, not shown, involves combining the EJBs in the Figure into two "lighter" EJBs, "AddressLight" and "CustomerLight", in order to reduce fetches. From a MCD perspective, the model representations for this antipattern can be expressed as a set of models representing potential "bad sizes" of lists of connected entities, or, alternatively, the required similarity to one representative model can be tuned to find potential antipattern instances.

B. Detecting Instances using MCD

Early model clone detection approaches [10], [16] may not be useful in their current form as they can detect Type 1 (exact) clones only. This means that our defined pattern models would have to be identical matches to potential pattern instances. Although, as shown in the extension done by Al-Batran et al. [32], it may be possible to use semantics-based

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

normalization techniques to establish a semantic correspondence between known patterns and a normalized representation of the models under study. For example, if we wanted to detect the *Composite Pattern*[4] or other design patterns, we could represent the pattern in a normalized form and look for both syntactic or semantic matches in our models.

On the other hand, newer model clone detection approaches that are able to detect Type 2 (renamed) and Type 3 (near-miss) clones [13], [15], [17], [14] are ideally suited for this task. For simpler, more purely structure-based patterns, the detected instances will be examples of Type 2 clones of the pattern, in that the structure of both the pattern model and the models being analyzed will be identical, but the element names or types will be different. In more complex model patterns that have more flexibility in their definitions and relationships, Type 3 (near-miss) clones of the pattern models will indicate potential pattern instances that can be verified by a specialist. Thus, we conclude that only Type 2 and Type 3 clones should be used for model pattern detection.

C. Example: Simulink Antipatterns

In this section, we present examples of Simulink antipattern representations and discuss how detection of instances of these can be implemented using MCD. We focus specifically on Simulink because (1) no one has yet achieved model pattern or antipattern detection for it; (2) Simulink is of particular interest to our industrial partners; and (3) we have previously developed a Simulink near-miss MCD tool, Simone [13], that we are familiar with and has been quantitatively evaluated [33]. In the future, we plan on implementing these model antipattern representations in Simulink and executing our detection process on our industrial and open-source models.

Based on their industrial experience and extending existing antipattern research on traditional code-based antipatterns, Tran and Kreuz [34] have developed a corpus of Simulink model antipatterns. They outline briefly, but without actual models or examples in Simulink, analogies of corresponding code-based antipatterns, and proposals for how to refactor them. They do not discuss or present any way of detecting them, rather they focus strictly on refactoring the models once instances of the antipatterns are discovered. Here we present a small set of potential Simulink model antipattern representations corresponding to a small selection of antipatterns in Tran and Kreuz's corpus.

1) Primitive Obsession: The Primitive Obsession antipattern is characterized by a project that has small subsystems encapsulating simple, or even primitive, calculations. In Figure 4, we present four potential Simulink representations, each its own subsystem, that we could use for cross cloning to detect primitive obsession antipattern instances. In our examples, the blocks have Simulink default names, which will be ignored by any model clone detector capable of detecting Type 2 or 3 clones.

2) Block/Signal Clumps: The Block/Signal Clumps antipattern involves a set of primitive blocks or signals that often appear together in various locations in a project. Since these are not encapsulated, in either a bus or subsystem, this increases the numbers of ports and the sizes of subsystems. Model clone detection detects this antipattern, in the generic sense, by default in that commonly recurring groups of blocks are always identified by MCD. However, if there were a specific clump of blocks/signals that we wanted to identify, we could include a clump in our pattern project/library, such as the one presented in Figure 5, and find all instances of that clump by detecting near-miss model clones at the block level.

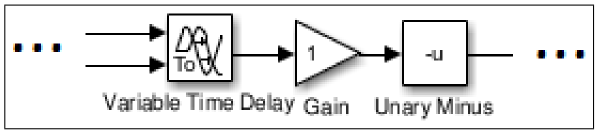

Fig. 5. Block/Signal Clumps Antipattern Simulink Example

3) Other Examples: Other interesting examples of Simulink antipatterns identified by Tran and Kreuz include the *Middle Man* and *Inappropriate Intimacy*. Middle Man occurs when a subsystem is essentially a delegate and not contributing significantly to the overall behavior of the system. A Simulink representation of it for MCD purposes might include a subsystem containing many inports connected directly to outports that are not part of some virtual or non-virtual bus. Inappropriate Intimacy involves a scenario where two systems are too highly coupled together. For detecting Inappropriate Intimacy, pattern models containing subsystems connected to one another through a varying number of connections can be developed.

IV. Areas of Research and Future Work

A. Simulink Antipattern Detection Using MCD

In the near future, we will be conducting research on detecting antipatterns in Simulink using Simone and developing an evaluated proof of concept prototype. This involves extending the preliminary list provided by Tran and Kreuz [34], devising and refining Simulink antipattern representations, and speaking to our industrial partners and other domain experts about other Simulink antipatterns of interest. The refinement of model antipattern representations involves developing potential pattern models and attempting to minimize the number of false positives encountered when performing MCD on those pattern models cross cloned with open-source and industrial projects. We will then present these representations to the modeling community at large for feedback, and formalize and package the Simulink antipattern detection process for adoption in both industry and research. We focus on model antipatterns for Simulink because these appear to be an important topic to both researchers and industry.

Another source of antipatterns that we can leverage for potential model clone targets is the MathWorks Automotive Advisory Board's modeling style guidelines[2]. These style guidelines include roughly eighty guidelines, some of which can be modeled as antipatterns.

[2]http://www.mathworks.com/solutions/automotive/standards/maab.html

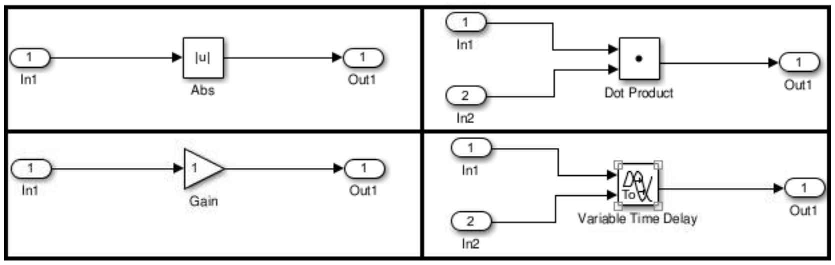

Fig. 4. Primitive Obsession Antipattern Simulink Examples

B. Semantically Equivalent Pattern Instances

The majority of MCD research thus far identifies structural clones only. While structure in these models often describe behavior, there is also the new notion of Type 4, *semantically equivalent*, clones. These clones occur when two very structurally different models exhibit identical behavior. From a pattern detection perspective, this is an interesting area of research because there may be models that meet the criteria of a pattern from a semantic perspective but have very little structural similarity to the definition of the pattern. Al-Batran et al. [32] present forty semantics-preserving Simulink transformations that illustrate this possibility. Evaluation using current MCD technologies or any type of model comparison would miss instances of both patterns and antipatterns that were semantically equivalent but structurally different beyond a defined threshold.

C. Using MCD for Model Pattern and Antipattern Mining

Model pattern inference involves discovering new libraries of patterns by looking at a large number of existing model-driven systems, extracting instances of common reuse, and evaluating these instances to determine if they are "good" or "bad" according to a domain expert. It may be hard to define what one is looking for in terms of "new" model patterns. A good starting point could be to think of model patterns as the model representation of traditional patterns similar to those associated with software code [5], [4], [35] for those that do not yet have a model-based definition.

For pattern inference, MCD approaches capable of detecting Type 1 clones only are not likely to be of use. This is because we are not looking for identical elements, but rather for those with similar intention in being constructed a certain way. What is most promising are MCD approaches that detect Type 2 and Type 3 clones. The similarity threshold and sample patterns can be tailored to fit the general types of patterns or sub-structures desired, possibly by extending techniques to consider semantic information. For behavioral models, model clone detection techniques could yield some interesting results as they are already designed to match the largest common subsequences of elements. If we look at the semantics of the clones extracted, likely through the assistance of a domain expert and the techniques discussed by Al-Batran et al. [32], we may be able to deduce model pattern- or antipattern-like sub graphs. A first approximation to this idea has been implemented by Antony et al. [15], who used model clone detection to find instances of potential security threats in design-recovered behavioral models of web application interactions.

D. Challenges

One challenge relates to representing the patterns in model form for clone detection and evaluating the process we lay out in this paper. It is important that both the model being chosen to represent the model pattern and the clone similarity threshold being used by an MCD tool retains the essence of the pattern, while still capturing as many pattern instances as possible. In other words, we want our model pattern representations and tool configurations to yield both high precision and high recall for each pattern. Early on, this can be accomplished informally and manually by experimentation and consulting with Simulink users. Eventually, the evaluation of precision and recall for model pattern detection, and evaluation of our proposed approach may be achieved quantitatively and automatically, as we have done in the past for MCD tool evaluation [33].

There may be some patterns that are more challenging to represent in a single model or set of models. For example, the "Divergent Change" Simulink antipattern [34] is when there is a single subsystem that "needs to be modified by many different types of changes". From a structural perspective, it is not immediately clear what set or sets of models could be used to encapsulate this antipattern. It may dictate a situation where a model pattern representation should employ a Simulink variation block[3] to account for multiple possibilities and block types.

Lastly, MCD for model types other than Simulink are only now beginning to emerge and mature. This lack of maturity is an inhibiting factor to realizing MCD-based model pattern detection for other model types.

V. Conclusion

Establishing model quality is an important step in cultivating the growth of MDE. One way of accomplishing this step is to have a facility for detecting both patterns and antipatterns at the modelling level. Almost all approaches thus far perform pattern detection on code or use textual rules, like Prolog or Java, to analyze models. In this paper, we propose using model clone detection (MCD), a form of model comparison, to detect model patterns directly using models only. This allows for analysis on primarily model-based projects, earlier quality assessment in the software engineering process, and keeps the level of abstraction consistent for analyzers and engineers.

[3]http://www.mathworks.com/help/simulink/slref/variantsubsystem.html

The process of using MCD for detecting instances of model patterns begins by devising a representative form of each model pattern being searched for. When a design pattern is defined in a general/high-level sense, we need to instantiate a concrete instance of that pattern and ensure both the instance and similarity threshold of the MCD tool afford us high recall. If the pattern definition is a specific example model or set of models, we need only to create those models in our modeling notation. After we set up our "pattern models" for cross cloning with the systems under analysis, we can then configure and execute an MCD tool to look for Type 2 and Type 3 clones only. All model clones in the same clone class as a pattern model can be interpreted to be an instance of that pattern. After outlining this process, we presented examples of potential Simulink antipattern models.

Our immediate plan involves realizing Simulink antipattern detection by constructing Simulink antipattern representations, configuring and executing our Simulink MCD tool, evaluating both our proposed process and the results, and repeating and refining that process. We plan on having much of this completed by the time of the workshop and presenting it in future papers. Areas of future work and research include detecting structurally different pattern instances, antipattern detection of other model types as their respective MCD tools mature, and addressing the research challenges we mentioned above. Overall, it is our position that using MCD for model pattern and antipattern detection can help further our ability to automatically evaluate model quality, thus helping to evolve the field of MDE.

ACKNOWLEDGMENT

This work is supported 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 Engineerin*, 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] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] M. Stephan and J. R. Cordy, "A survey of model comparison approaches and applications," in *International Conference on Model-Driven Engineering and Software Development*, 2013, pp. 265–277.

[12] D. Kolovos, R. Paige, and F. Polack, "Model comparison: a foundation for model composition and model transformation testing," in *international workshop on Global integrated model management*, 2006, pp. 13–20.

[13] 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.

[14] 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.

[15] 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.

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

[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] 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.

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

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

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

[23] 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.

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

[25] 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.

[26] 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.

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

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

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

[30] 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.

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

[32] B. Al-Batran, B. Schatz, and B. Hummel, "Semantic clone detection for model-based development of embedded systems," *Model Driven Engineering Languages and Systems*, pp. 258–272, 2011.

[33] 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.

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

[35] D. Schmidt, M. Fayad, and R. Johnson, "Software patterns," *Communications of the ACM*, vol. 39, no. 10, pp. 37–39, 1996.