Model-Driven Evaluation of Software Architecture Quality 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*—As software architecture methods and tools become increasingly model-driven, evaluating architecture artifacts must adjust correspondingly. Model-driven evaluation of architecture quality has advantages over traditional evaluation techniques, especially when applied in a model-driven context. One approach we found successful in performing model-driven analysis involves using model clone detection, whereby we detect subsystems that are similar to example systems that are positive and negative quality indicators. In this paper we present our ideas on applying model clone detection to realize model-driven evaluation of software architectures, which contain many high-level systems and interactions. We propose having model-based representations of architectural patterns and styles, and employing model clone detection to identify positive and negative architectural aspects for evaluation, including reliability and security. We provide our insights on how this research can be applied to popular architectural paradigms, relation to previous work, and present discussion points on how it will impact software architecture quality evaluation.**

I. Introduction

As the software engineering field and technology mature, software systems grow more complex and larger in size. Correspondingly, the software architectures defining these systems are becoming more complex. This complexity causes challenges both in the design of this architecture and the evaluation of its quality throughout the software life cycle. Seeing as architectural design is more abstract than detailed design or implementation, one technique used in many domains to combat this complexity is model-driven architecture (MDA) [1]–[3]. Pure MDA projects use a variety of high-level abstractions, known as models, and combine these models at design time to perform model execution, verification, and code generation for embedding onto platforms [3]. However, even architects not following a pure MDA approach likely use some higher-level modeling notation to represent their architectures [4], such as using the unified modeling language (UML) to define architectures [5], or even higher-level architecture description languages (ADL) [6].

Evaluating architectures is a well-researched problem with many approaches [7] considering different factors such as reliability, security, maintainability, portability, modularity, and others [8]. For example, most approaches employ a scenario-based technique [7], whereby scenarios emulate activities the architecture should support [9]. Important issues facing existing software architecture evaluation techniques include incorporating them into existing development processes and providing tool support [7]. These techniques require relatively low-level abstractions compared to the architecture models themselves, such as textual formalisms, or require using a new modelling language. In many cases they entail manual evaluation steps, for example, continually developing scenarios and calculating each scenario's impact and weighting [8].

An emerging approach to evaluating software architecture quality is model-driven evaluation, which uses higher-level models to analyze architectures. One form of model analysis yet to be leveraged for this purpose is near-miss model clone detection, which involves finding identical or similar sets of models, up to a given specified difference threshold [10], [11]. Since software architecture patterns and styles can be viewed as "a family of systems in terms of a pattern of structural organization" [12], and the presence of architecture patterns and styles can be employed to measure architecture quality [13], including reliability and security concerns, model clone detection is an ideal candidate to realize model-driven quality evaluation.

One automatic approach we devised, implemented, and validated for model-driven quality evaluation of stand-alone models and subsystems was to use model clone detection to identify instances of individual systems that were similar to established "good" or "bad" examples, known as pattern models [14]. Using an approach featuring model clone detection allows for a completely model-driven evaluation since analysts define and use pattern models that are compared to the models being analyzed [15]. It requires no additional textual or modeling formalisations. A key limitation of our past work is that it was designed and implemented to function at the stand-alone (sub)system level only, that is, in the context of individual systems and their constituents, such as individual Simulink systems [14]. Software architecture, on the other hand, is more high level than individual systems and is a composition of design decisions [16] typically represented as a model that defines elements, form, and rationale [4]. It consists of different views and sets of interconnected models [17]. So while individual systems work with what Buschmann termed "stand-alone patterns" [18], software architecture patterns are higher-level and focus more on interconnections and higher-

level concerns. This presents a challenge in that our previous work in its current state is ill suited to handle these concerns and complex connections.

In this position paper, we propose using model clone detection for semi-automatic model-driven evaluation of software architectures. Specifically, we propose that analysts detect quality indicators in an example-driven manner whereby architectural patterns/styles are represented as examples in the same modeling language notation(s) as the architecture implementations themselves. These example, or "family", models then undergo model clone detection with the systems being analysed, and any clone matches can be used to reason about architecture quality. One merit of this technique is that it is general enough that various types of architectural aspects can be evaluated at once by having the family models represent security, maintainability, reliability, performance, conformance, and other concerns. Additionally, even as architecture instances evolve, the evaluation can take place. Lastly, using this technique provides the same benefits we noted for stand-alone system evaluation including [15] (1) quality and risk assessment early in a project's existence, (2) applicability to projects and teams employing either a pure- or part- MDA philosophy, and (3) no textual or new modeling languages being required.

This position paper begins in Section II with some background on model clone detection and architectural patterns and styles. We then overview the process in Section III and provide our insights on how it can be realized for popular architectural paradigms in Section IV. We present related work in Section V and points for discussion in Section VI, followed by our conclusion in Section VII.

II. Background

A. Model Clone Detection

A software clone is an element of software that is similar or identical to another element. Traditionally, clone research has focused on the notion of "code clones", which refers to clones in textual code representations [19]. More recently, techniques are being developed for model clone detection, whereby analysis focuses on finding identical or similar model elements or sets of elements [10]. The majority of research thus far has focused on Simulink [10], [11], but work is being done on other modeling formalisms like UML [20], [21] and Stateflow [22]. Model clone detection approaches can employ different methods for comparison including graph-based techniques, textual analysis of the models' representations, or a variety of heuristic approaches [23]. These model clone detection approaches have been validated and proven to be effective through experimentation, and industrial validation [10], [21], [24].

There are four types of model clones [11]. A type 1 model clone pair represents two models that are completely identical, ignoring any formatting or layout information. Type 2 model clones are models that are structurally identical to one another but allow for renamed elements or changed element values. Type 3 model clones, or near-miss model clones, are pairs/sets of models that are similar up to a certain similarity threshold, such as 75% structural similarity. Type 4 model clone pairs are those are significantly different structurally, beyond a Type 3 threshold, but are semantically equivalent. More technical details on model clone detection can be found in our previous work and others [10], [11], [21].

1) Model Clone Detection as an Evaluation Tool: In recent work, we described how model clone detection can be used as an evaluation tool for stand-alone systems [15]. Utilizing "cross clones" [25], we suggested having analysts run model clone detection on the union of their systems and sets of "pattern" models, representing either established good or bad solutions. Any model clones that cross these sets indicate that an individual system is an instance of that pattern. This is made possible through the detection of Type 3 model clones. We implemented and validated this approach in a tool for analyzing Simulink systems by detecting instances of antipatterns [14]. This tool is able to detect five different types of antipatterns in Simulink model sets, demonstrating that structure is enough to identify potential pattern instances. In this position paper, we propose adapting this work to cater it towards software architecture quality analysis, addressing architectural-specific aspects and features.

B. Architectural Patterns and Styles

One of the requirements of our proposed approach is there must be descriptions/examples of architecture that represent some notion of either a "good" or "bad" way of doing things. Fortunately, this is fairly prevalent for software architecture in the form of software architecture patterns [13], [18]. These software architecture patterns are architectural solutions to commonly occurring questions or problems. They can be incorporated into software architecture design and development approaches and, when it comes to quality, Harrison and Avgeriou speak explicitly to the idea that architectural patterns can be used to satisfy and measure architectural quality aspects [13]. There are examples in the literature of concurrency patterns [18], [26], [27], scalability patterns [18], context-processing middleware patterns [28], usability patterns [29], and more.

A synonymous term to architecture patterns is software architectural styles, where an architectural style "abstracts elements and formal aspects from various specific architectures" [30]. For example, there are architectural styles that address GUI construction [31] and network-based system layouts [32]. Architectural styles can be formalized [33] and there is even tool support for performing architectural style-centered creation [34].

III. Overview of Proposed Process

Since an architectural pattern/style can be viewed as "a family of systems in terms of a pattern of structural organization" [12], and the presence of these can be used to measure quality [13] it is the goal of our process to ascertain if a specific architecture undergoing quality evaluation belongs to that "family of systems." Thus, we can employ Type 3 (near-miss)

model clone detection with an tuned similarity threshold to see if each specific architectural instance is structurally similar to an architecture "family model" that is both representative of that family and is in the same form as the architecture itself. Software architecture syntax/structure dictates and defines the semantics since the syntactic domain maps to the semantic domain [33], so any model clones containing a "family model" will demonstrate similar positive or negative quality indicators.

The general process involves three stages, which we adapt from our earlier work [15].

A. Stage 1: Creating Architecture Style/Pattern Family Models

The first step involves defining the family models that will be the basis of comparison. This is a manual step that needs to be performed only once.

This stage in the process can include having analysts consult a domain expert, conduct domain analysis, and incrementally refine the family models through experimentation. The family models must be created by the system analyst using the same modeling language/tools used to create the architecture models that are to be analyzed later. For example, in our previous work involving stand-alone Simulink models [14], we created family models in Simulink by leveraging existing research on cataloging patterns, adapting standards from advisory boards, and utilizing company specific sources, like domain experts and documentation. Sources such as these are likely to exist and be viable for architectural modeling languages, in addition to other sources.

The family models must be general enough such that they can be matched to any potential implemented candidates, that is, facilitate high recall. This is especially important as many patterns are configurable or have many variation points. The family models must not be too general, however, so as to not impact the precision of the matches. The family models' definitions should be completed and tweaked by analysts, domain experts, and researchers. This step is the main challenge in adapting our existing work as software architectures are very abstract high-level end-to-end representations, focusing on multiple systems, and having an emphasis on organization and interconnections, which contrasts our previous work that focuses on individual systems and stand-alone patterns.

In Section IV we discuss the feasibility of creating family models for popular architectural representations.

B. Stage 2: Model Clone Analysis

The second stage involves running model clone detection on the union of the family models created in the first stage and the architecture models being analyzed. This step is automated in that all this work and analysis is performed by the clone detector once it is setup. The only manual aspect is developing a suitable model clone detector, which needs to be done only once.

A key prerequisite of this proposed evaluation approach is that there exists a model clone detector for the architectural models/formalisms being used. New model clone detection techniques and tools can be realized using a variety of approaches, and in our previous work we outlined the steps we took in creating a text-based model clone detector [11], SIMONE. These steps included 1) creating a grammar to represent the modeling language describing the models and their structural information, as SIMONE is parser-based and language-sensitive, and 2) normalizing the underlying modeling representations through a combination of filtering and sorting. To apply this same approach to allow for model clone analysis of any of the architectural languages we describe in this paper, both of these steps will be important and can be achieved through domain analysis, speaking to experts, and refinement. Seeing as these architectural languages describe structural information and aspects, we believe that using textual analysis is sufficient, especially if we build it on the same framework as SIMONE was built on. Whatever model clone detector approach is used, a fairly large (around 33-50%) Type-3 difference threshold will be necessary to find instances of design patterns in order to allow for a lot of pattern variations.

After development of a model clone detector and creation of the family models, this stage is automatic and can occur at any time after the architectures to be analyzed have been created or designed. Thus, if the family models exist before the project begins, analysts can perform early model-driven evaluation of their architectures and continue to do so throughout the software life cycle simply by executing model clone detection. Execution entails running model clone detection on the union of the family models and architectures models being analyzed for quality, which is passed as input to the next stage: reasoning about software architecture quality.

In Section IV, we postulate about using or creating model clone detectors for the chosen architectural representations based on our experiences.

C. Stage 3: Reasoning about Software Architecture Quality

After model clone detection analysis is complete, architecture quality analysts can identify the model clones of pattern family models in the system architectures under study. Identifying cross clones is automatic as many clone detection tools can explicitly indicate cross clones [14], [25].

Once cross clones are identified automatically, analysts are made aware which systems under study are instances of which design patterns or antipatterns. Each architectural model cross clone, illustrated as the "X" in Figure 1, represents a categorization of that system's architecture as an instance of the architectural pattern/style family. In other words, we are looking for architectural model clone pairs that intersect between the set of pattern family models and the system architectures being evaluated. Tools like SIMONE identify intersections automatically [11].

At this point, analysts can record all identified instances and begin to draw qualitative and quantitative quality conclusions. For example, Harrison and Avgeriou [13] indicate positive and negative quality implications of various architectural pattern usages. For each pattern and quality aspect, they assign a

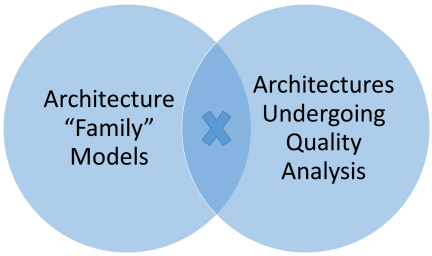

Fig. 1. Identification of Architectural Model Cross Clones

TABLE I
EXAMPLE OF QUANTITATIVE QUALITY VALUES FOR PATTERNS

Pattern	Security	Reliability	Efficiency	Portability
Layers	+2	+1	-1	+1
Pipes & Filters	-1	-2	+1	+2
Blackboard	-1	0	-1	0

label of "Key Strength", "Strength", "Neutral", "Liability", or "Key Liability". As we demonstrate in Table I, we can ascribe quantitative values to these labels based on their findings. For example, having a range of values from "Key Strength" equaling two to "Key Liability" equaling negative two. So, if we were to detect an instance of the Blackboard pattern in a project, then we ascribe a -1 score to security and efficiency, with no impact on reliability or portability. If we find architectural instances of the Layers pattern, then we give security a +2 and reliability +1. Other example quality aspects that we can consider including are usability, maintainability, and implementability [13].

A nice aspect to this approach is that the analysis can be compositional. That is, it should be possible to assess quality attributes of an architecture based on the assessment of its parts, facilitating both a high-level composite quantitative score to present to decision makers, and a low-level detailed score for system developers and engineers. From a traceability perspective, the score sources themselves can be linked directly to the systems exhibiting the relevant qualities as the cross clone information will include the location of the architectural models demonstrating the properties.

There are other sources of quality implications of patterns that can be used to create quantitative measures such as these, including work by Shaw and Garlan [12]. However, this is an emerging area that will grow as the field of software architecture quality research matures.

IV. APPLICABILITY TO POPULAR ARCHITECTURAL LANGUAGES

In order to consider the plausibility of using model clone detection for model-driven software architecture quality evaluation, we consider the three most widely used software architectural languages. In 2013, Malavolta et al. performed an industrial survey on ADL use and understanding [6]. When asked which ADLs they use, 86% of the participants said they use UML, 16.3% said they use the Architecture Analysis & Design Language (AADL) [35], and 11.63% use ArchiMate [36].

A. UML

The most widely used tool for describing software architecture is UML. UML can describe multiple software architecture views through different modeling representations. For example, UML Component diagrams can be used for a code architecture view, UML Package diagrams for module architecture views, and UML Class diagrams with stereotypes along with UML State or Sequence diagrams for execution architecture views [5]. Furthermore, UML 2 added model concepts that improve the ability to represent architectures [37].

Regarding the plausibility of Stage 1, creating family models for UML, architectural patterns and styles for UML are often represented as the general form of models themselves. In the simple case, the UML architectural pattern will be a single generalized model in a specific type of diagram, such as a Component diagram. This can be instantiated as a family model and will be a Type 3 model clone to anything similar. In other scenarios, there may be multiple diagram types that, together, best identify the pattern. It would be prudent for the family models to include all the corresponding diagrams to increase the precision of the pattern identification, thus improving the quality analysis. This is especially true in the case of UML 2.0 since the connectors allow for many variations of architectural element representations [37]. Thus, using Type 3 model clones can be used in conjunction with a refined and validated set of family models.

Take, for example, the Reactor Software Architecture Pattern [26], which handles server requests concurrently to make projects better suited to multi-threading and modular development. It can be described through a UML Component diagram, such as the one in Figure 2 showing the internal components of the Logging Implementation and its relation to external components. It also has an associated UML Class diagram and UML Sequence diagram, which we omit. All three of these diagrams should be represented as family models and associated together in order to best identify instances of the Reactor Pattern.

From a model clone analysis perspective for Stage 2, there exists model clone detectors for UML. The MQlone tool [21] detects UML model clones through a combination of graph analysis and UML-specific heuristics, such as element names and indexes. There is also a model clone detector specifically for UML behavioral models [20] that uses the models' underlying textual representations. Because UML profiles are prevented from breaking the UML semantics and structural rules, these model clone detection techniques can either account or be updated to account for any UML profiles in a specific domain or company. In the simple case of having a single type of UML diagram, model clone detection can be used as per usual. In the more complex case of having multiple UML diagram types together form the architectural pattern or style, the model clone detector would have to be aware of this. While this capability does not yet exist, we

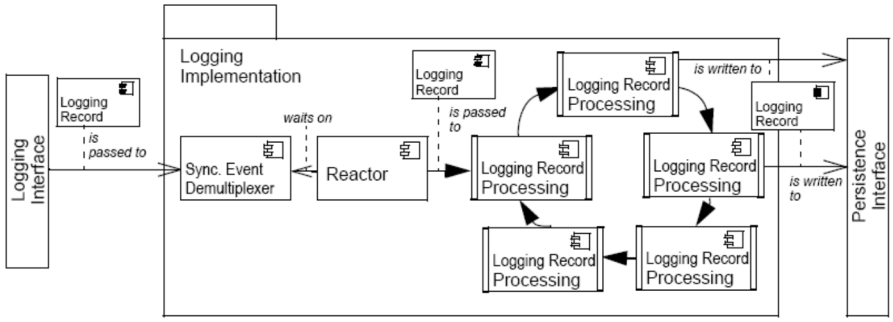

Fig. 2. UML Component Diagram for Reactor Architectural Pattern [18]

see two potential extensions to model clone detectors that can accomplish it. (1) In the case that the components in the architectural implementations are explicitly linked [5], the family models can be annotated in such a way that they are associated with one another and the model clone detector would use that information to report only potential instances where all, or the majority of, the linked diagrams have a clone to that family model group. (2) A more general extension is to have the model clone detector pre-process both the family models and architecture implementations being analyzed. Specifically, the detector could combine the family model group of diagrams into one diagram/representation and compare that against the combination, or cross product, of all of the relevant implemented architectural diagrams being analyzed. So, for example, the Reactor architecture family model group from earlier would have a Component, Class, and Sequence diagram. These three would be combined into one family model and compared against all combinations of Component, Class, and Sequence diagrams implemented in the architectures being analyzed. This would obviously be more computationally intensive, however, it would be a much more general and applicable approach. One that would be computationally feasible if we were to use the underlying textual representations of these diagrams instead of doing graph matching [11].

B. AADL

An important feature of the AADL is that it is both a graphical and textual modelling language [35]. For example, take the architectural style of having "mutually informing components" [38] facilitating increased dependability by having components determine their and their partners' modes. It can be generalized in AADL graphical form, as shown in Figure 3 demonstrating two identical sub-components and their various modes. There is also an AADL textual model representation of this architectural style [38], which we omit.

For Stage 1 of our proposed process, creating the family models for AADL would involve instantiating the AADL graphical and/or textual model defining the architectural pattern or style. While AADL models can represent components at different levels of abstraction, they are all the same diagram type and adhere to the same meta model/grammar. They can be defined textually, graphically, or in a combination of the two [35]. So, the choice between graphical versus textual representation of the family models would correspond to both the representation of the AADL architecture models being analyzed and the AADL architectural pattern/style definition models. If the models to analyze used only graphical or textual representations, then the family models would need contain only graphical or textual elements, respectively.

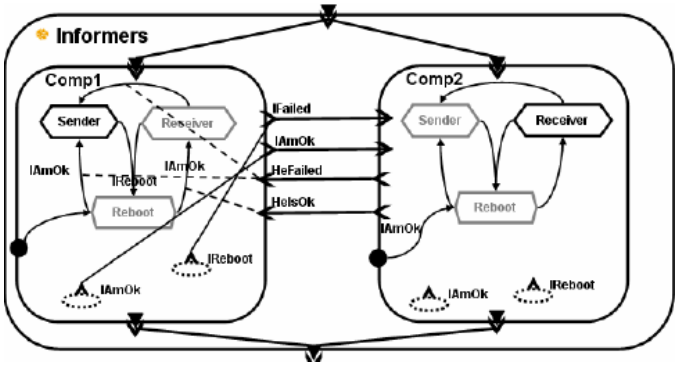

Fig. 3. AADL Model for Mutually Informing Components [38]

From a model clone detection, Stage 2, perspective, there are no tools to our knowledge that detect model clones in AADL. However, because AADL models have textual representations, it is likely that one could develop an AADL textual model clone detector the same way we created a model clone detector for Simulink [11]. This is what we will do if we choose to implement this process for AADL.

C. ArchiMate

ArchiMate is an ADL modeling language specifically targeted towards modeling enterprise architecture [36]. It includes a set of meta models accounting for various levels of architecture abstraction including, in order of specificity, domain models, dynamic systems, enterprise architectures, and project-level concepts. ArchiMate accounts for business, applications, and technology all within one language [36].

Stage 1 family models intended to work with ArchiMate languages would need to be created using the target ArchiMate tool implementation used to design the architectures being analyzed. Because ArchiMate models have many inter-domain relationships, the family models would need to capture that

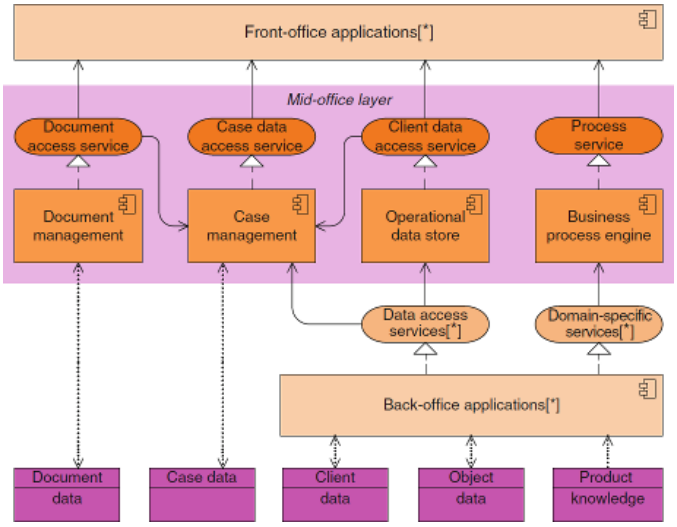

Fig. 4. ArchiMate Mid-Office Architecture Pattern [39]

behavior from associated styles and patterns. Take for example the Mid-Office Architectural Pattern [39] in Figure 4. It provides a unified external view to clients while a system is transitioning to a service-oriented architecture from a traditional back-office one. It has other patterns nested within it, including the Document Management and Business Process Engine elements, so the family models would need to include the lower level elements as well.

Seeing as ArchiMate has various tools implementing its language, the model clone detection analysis necessary for Stage 2 of our proposed process would depend on the specific ArchiMate tool. In cases where only the graphical representation was available, then graph-based model clone detection would be the direction to take. If the textual representation was also available for analysis, than a text-based model clone detection approach could be employed. Since "each level inherits the concepts from the previous level, while also providing specialisations of the existing concepts" [36], the grammar/syntactic comparison would need to account for all concepts from all meta levels in the scope. Applying and extending the work we have done for SIMONE would likely allow us to create a model clone detector for ArchiMate tools, assuming that the specification of the textual representation is available.

V. Related Work

There are many techniques for evaluating software architecture [7], [8]. These are often geared towards specific quality attributes such as modifiability, design suitability, reliability, security, flexibility, and others [7]. The most popular evaluation approach uses scenarios that describe cases and forms that the architecture must account for and take, respectively [9]. All crucial uses of a system are covered by the scenarios, allowing analysts to monitor scenario interactions and reactions. As we presented in this paper, architectural quality evaluation can also be model-driven. Some examples include performance models [40], [41], consistency checking through analysis graphs [42], pattern instance detection using logic reasoners [43], and non-functional analysis through formal models [44]. In these examples, and others, analysts are required to learn and use a new modelling language or formalism, and cater their analysis to specific quality aspects. This contrasts the technique we propose in that analysts can use the same modeling language(s) that their architectures are described in and, as long as the quality aspects can be modelled in an example driven-manner, analysts can evaluate different architectural quality aspects, including reliability and security.

Zhu et al. extract information from established patterns to help an architect make and evaluate decisions in their domain [45]. This work contrasts ours in that it does not perform analysis directly on the architecture models, but rather helps in the forward engineering direction. However, the information they extract from patterns can definitely be useful when drawing conclusions about the pattern instances detected through our process. Similarly, Reference Architectures are generic architectures with associated quality benefits for engineers to use when designing their systems [46]. It is possible to leverage some of these reference architectures as sources of patterns. Kim [47] devised a meta-modeling approach to specify patterns. While our proposed family models are based directly on the style and pattern models, it is possible that the family models can be enhanced by adding information from the meta models.

VI. Discussion Points and Future Work

Using model clone detection for architecture quality evaluation is widely applicable in that various types of modelling quality measures and properties can be defined as pattern family models. From a familiarity and learning perspective, it has the advantage that analysts can specify desired architectural properties directly as examples in the same modelling formalism as the architectures to be evaluated. Using and extending existing model clone detectors should reduce the amount of additional scientific work required to realize Stage 2 in our proposed process for the various ADLs. We would likely leverage our existing work on SIMONE, while exploiting various grammar inference techniques [48] to analyze the structural information described by these ADLs. We are quite confident that we would be able to find clones in these domains based on the success we and others have had in achieving model clone detection. Some of these architecture models may have many elements and lots of potential combinations of structure. However, we believe that employing an approach using the same textual analyzer that we built our tool to detect Simulink patterns on will counter this nicely, as it sufficiently considers structural information described in the respective modeling languages, avoids the sub-graph isomorphism problem [11], and is built upon a very efficient and tested text clone analyzer [49].

The correlation between patterns and quality has been discussed by Harrison and Avgeriou [13] specifically for ar-

chitecture patterns and by many others for software in general including traditional programming code domains [50], [51]. However, the exact correlation between patterns and quality is not clear. While Harrison and Avgeriou provide labels for the impact of a pattern on its various quality aspects that we proposed converting numerically earlier in Table I, an important area of future work is to better define how "good" or "bad" various architecture patterns are and to validate this in regards to overall architecture quality. Another quality factor to consider is looking at the context of the architecture pattern to help assess its impact to system quality on the whole. For example, "crucial" systems showcasing architecture patterns would have greater implications than instances found in less crucial systems. Of course, quality is relative to the business context. As we presented in our table, the decision to use a pattern or the remedying of an antipattern will likely have trade offs. But it is important to remember that our proposed process will only identify potential pattern instances and provide some automated quality insights. It is not meant, at least at this point, to automatically refactor the architectural models. While an overall quality assessment may be difficult, even the compositional evaluation will have value in itself and can lead to a suggested overall quality assessment based on various aspects of the architecture, such as security, maintainability, and reliability. It may be easier and more useful to focus on antipatterns because an architecture that matches a bad pattern is bad, whereas, it may be difficult to conclude anything from a partial match with good patterns.

Through this proposed process, architecture quality can be checked continually throughout the software's lifetime despite architectural evolution. This can ensure conformance during a project to help with architectural management. Traceability is enhanced as analysts can track styles or patterns occurrences over time and the impact on overall quality. This is an improvement over the current practice, whereby a recent study showed that architecture design decisions are often mistaken and rarely documented [52].

This process, once realized and implemented, is semi-automatic. While Stage 1: the creation of family models, is a one-time manual process, it is theoretically possible to automate it through the notion of pattern inference. This is separate research area [53], and one that would enrich the model-driven evaluation of architectures, but is not critical. The second stage, model clone analysis, is an automatic process. The reasoning stage, or Stage 3, is done semi-automatically, as cross clones can be identified by the tools and quantitative scoring can be programmed and automated. On the other hand, qualitative assessments and decisions would be manual. A long term goal would be to incorporate automatic or guided refactoring based on the analysis, as done in other domains [54].

A notable limitation of this process is that there must be examples of styles or patterns for models to emulate. While each domain and industry can develop them, it is not always the case that this information is shared with the community as a whole. Also, tweaking the family models to achieve a desired measure of recall and precision is challenging. Typically because "correct" instances would have to be verified manually by an expert. We may have to start with a large difference threshold, like 50%, and refine the models incrementally. In addition, there is the possibility that architects may deliberately have anti-patterns, thus, this approach would lead to false alarms. However, since this is simply an analysis process, refactoring these architectures is something that can be left as a decision for project stakeholders.

Our first step will be to focus on a particular domain, decided on by gauging the industrial and academic communities to see what is interesting and feasible. We will then determine if it is possible to extract useful architectural structures that represent instances of positive or negative qualities based on the literature and domain experts. Once we have completed implementing this proposed process for the decided architectural language, an important aspect of work involves evaluating and validating its merits. This includes determining if a variety of architectural aspects can be evaluated, likely by calculating recall and precision of pattern instance detection for the various types architecture patterns. It should be clear if an implementation of the process will allow early project evaluation as we would test it on architectures representing projects at various levels of detail and maturity.

VII. Conclusion

In this position paper we presented an approach for model-driven evaluation of software architecture quality that uses model clone detection by having example architectures associated with quality measures and properties as the basis for comparison. In addition to being completely model-driven, a key benefit is that analysts need not use any modeling formalisms other than the ones they use already in their architecture designs and implementations. We presented our ideas on how the process would work for three popular architecture description languages, including examples based on our experiences and the literature. We believe this work has much promise in the area of software architecture quality because its ability to assess various quality aspects, help analysts understand their systems better, and its capability of having evaluation performed any time during the software life cycle.

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] S. J. Mellor, K. Scott, A. Uhl, and D. Weise, "Model-driven architecture," in *Advances in Object-Oriented Information Systems*. Springer, 2002, pp. 290–297.

[2] A. G. Kleppe, J. Warmer, and W. Bast, *MDA explained: The model driven architecture: practice and promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2003.

[3] S. J. Mellor, *MDA distilled: principles of model-driven architecture*. Addison-Wesley Professional, 2004.

[4] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[5] C. Hofmeister, R. L. Nord, and D. Soni, "Describing software architecture with UML," in *Software Architecture*. Springer, 1999, pp. 145–159.

[6] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What industry needs from architectural languages: A survey," *Transactions on Software Engineering*, vol. 39, no. 6, pp. 869–891, 2013.

[7] M. A. Babar, L. Zhu, and R. Jeffery, "A framework for classifying and comparing software architecture evaluation methods," in *Australian Software Engineering Conference*. IEEE, 2004, pp. 309–318.

[8] L. Dobrica and E. Niemelä, "A survey on software architecture analysis methods," *Transactions on Software Engineering*, vol. 28, no. 7, pp. 638–653, 2002.

[9] R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario-based analysis of software architecture," *Software*, vol. 13, no. 6, pp. 47–55, 1996.

[10] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert, "Clone detection in automotive model-based development," in *International Conference on Software Engineering*, 2008, pp. 603–612.

[11] 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.

[12] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Prentice Hall Englewood Cliffs, 1996, vol. 1.

[13] N. B. Harrison and P. Avgeriou, "Leveraging architecture patterns to satisfy quality attributes," in *Software Architecture*. Springer, 2007, pp. 263–270.

[14] M. Stephan and J. R. Cordy, "Identification of Simulink Model Antipattern Instances using Model Clone Detection," in *International Conference on Model Driven Engineering Languages and Systems*, 2015, pp. 276 – 285.

[15] ——, "Identifying Instances of Model Design Patterns and Antipatterns Using Model Clone Detection," in *International Workshop on Modeling in Software Engineering*, 2015, pp. 48 – 53.

[16] J. Bosch, "Software architecture: The next step," in *Software architecture*. Springer, 2004, pp. 194–199.

[17] P. Avgeriou and U. Zdun, "Architectural patterns revisited–a pattern," in *European Conference on Pattern Languages of Programs*, 2005.

[18] F. Buschmann, K. Henney, and D. Schimdt, *Pattern-oriented Software Architecture: On Patterns and Pattern Language*. John wiley & sons, 2007, vol. 5.

[19] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[20] 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.

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

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

[23] 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.

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

[25] 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.

[26] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2013, vol. 2.

[27] R. Mzid, C. Mraidha, J.-P. Babau, and M. Abid, "SRMP: a software pattern for deadlocks prevention inreal-time concurrency models," in *International Conference on Quality of Software Architectures*, 2014, pp. 139–144.

[28] R. Rouvoy, D. Conan, and L. Seinturier, "Software architecture patterns for a context-processing middleware framework," *Distributed Systems Online*, vol. 9, no. 6, pp. 1–1, 2008.

[29] E. Folmer and J. Bosch, "Usability patterns in software architecture," in *International Conference on Human-Computer Interaction*, 2003, pp. 93–97.

[30] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[31] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr, J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow, "A component-and message-based architectural style for GUI software," *Transactions on Software Engineering*, vol. 22, no. 6, pp. 390–406, 1996.

[32] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[33] G. D. Abowd, R. Allen, and D. Garlan, "Formalizing style to understand descriptions of software architecture," *Transactions on Software Engineering and Methodology*, vol. 4, no. 4, pp. 319–364, 1995.

[34] B. Schmerl and D. Garlan, "Acmestudio: Supporting style-centered architecture development," in *International Conference on Software Engineering*. IEEE, 2004, pp. 704–705.

[35] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis & design language (AADL): An introduction," DTIC Document, Tech. Rep., 2006.

[36] M. M. Lankhorst, H. A. Proper, and H. Jonkers, "The architecture of the Archimate language," in *Enterprise, Business-Process and Information Systems Modeling*. Springer, 2009, pp. 367–380.

[37] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little, *Documenting software architectures: views and beyond*. Pearson Education, 2002.

[38] P. Feiler and A. Rugina, "Dependability modeling with the architecture analysis & design language (AADL)," DTIC Document, Tech. Rep., 2007.

[39] M. Iacob, M. Lankhorst, and A. Schrier, "Patterns for agility," in *Agile Service Development*. Springer, 2012, pp. 95–110.

[40] A. Brunnert, K. Wischer, and H. Krcmar, "Using architecture-level performance models as resource profiles for enterprise applications," in *International Conference on the Quality of Software-Architecturess*, 2014, pp. 53–62.

[41] G. A. Moreno and P. Merson, "Model-driven performance analysis," in *International Conference on the Quality of Software-Architectures*. Springer, 2008, pp. 135–151.

[42] M. Biehl and W. Löwe, "Automated architecture consistency checking for model driven software development," in *Architectures for Adaptive Software Systems*. Springer, 2009, pp. 36–51.

[43] G. Luitel, M. Stephan, and D. Inclezan, "Model level design pattern instance detection using answer set programming," in *International Workshop on Modeling in Software Engineering*, 2016, pp. 13–19.

[44] L. Berardinelli, P. Langer, and T. Mayerhofer, "Combining fUML and profiles for non-functional analysis based on model execution traces," in *International Conference on Quality of Software Architectures*, 2013, pp. 79–88.

[45] L. Zhu, M. A. Babar, and R. Jeffery, "Mining patterns to support software architecture evaluation," in *Working IEEE Conference on Software Architecture*. IEEE, 2004, pp. 25–34.

[46] S. Angelov, P. Grefen, and D. Greefhorst, "A framework for analysis and design of software reference architectures," *Information and Software Technology*, vol. 54, no. 4, pp. 417–431, 2012.

[47] D.-K. Kim, "A meta-modeling approach to specifying patterns," Ph.D. dissertation, Colorado State University, 2004.

[48] A. Stevenson and J. R. Cordy, "Grammatical inference in software engineering: an overview of the state of the art," in *Software Language Engineering*. Springer, 2012, pp. 204–223.

[49] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *International Conference on Software Maintenance and Evolution*, 2015, pp. 131–140.

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

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

[52] Z. Durdik and R. Reussner, "On the appropriate rationale for using design patterns and pattern documentation," in *International Conference on Quality of software architectures*. ACM, 2013, pp. 107–116.

[53] P. Tonella and G. Antoniol, "Object oriented design pattern inference," in *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 1999, pp. 230–238.

[54] M. O. Cinnéide, "Automated application of design patterns: a refactoring approach," Ph.D. dissertation, Trinity College, 2001.