Model Level Design Pattern Instance Detection using Answer Set Programming

Gaurab Luitel
Department of Computer Science and Software Engineering
Miami University
Oxford, Ohio, USA
luitelg@miamioh.edu

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

Daniela Inclezan
Department of Computer Science and Software Engineering
Miami University
Oxford, Ohio, USA
inclezd@miamioh.edu

ABSTRACT

Software engineering is becoming increasingly model-centric. Engineers are using models more within projects and their models are growing in complexity. A challenge facing the modeling community is evaluation of these models. One technique for software evaluation is detecting instances of established "good" or "bad" solutions in a system, often termed design patterns or antipatterns, respectively. Most approaches require implemented code for detection. However, this precludes early-stage analysis, and the evaluation of purely or mostly model-centric systems. In this position paper, we introduce a detection technique that uses answer set programming to find occurrences of patterns within sets of structural and behavioral models. We represent the patterns as rules and the structural and behavioral system models as facts, requiring both model types since some patterns specify both. We provide an overview of our proposed approach, contrast existing work, and present discussion points on its impact on model evaluation and anticipated challenges.

CCS Concepts

•**Software and its engineering → Model-driven software engineering; Constraint and logic languages; Software maintenance tools;** *Software reverse engineering; Maintaining software;* Software design tradeoffs;

Keywords

model evaluation; design patterns; antipatterns; model patterns; model quality; answer set programming

1. INTRODUCTION

The use of modeling is becoming increasingly prevalent in the software engineering community [19]. Whether engaged in a pure model-driven process [30] where the models are the primary artifacts in all phases of the project's life cycle, or in a partial model-centric environment, engineers are becoming more comfortable with modeling. As this comfort increases and the complexity of software systems grows, so does the reliance and complexity of the software models. One challenge facing the modeling community is providing the means to analyze models to assess their quality, often termed model evaluation [26, 27].

Traditional software evaluation is quite mature [28] compared to model evaluation. One established method of evaluating software quality is by detecting the existence or absence of accepted positive or negative system implementations [16, 39], known as "design patterns" and "antipatterns", or "patterns" collectively. Finding instances of these in software not only promotes common vocabulary but also provides analysts an immediate indication of quality based on what is found and where. Pattern detection has been realized through both programming code analysis and model analysis. An advantage of the latter is allowing for analysis at any stage during the software engineering process. The majority of model analysis approaches focus solely on structure diagrams, such as UML Class diagrams. However, structural information alone is not always sufficient for software pattern detection [3]. Including semantic information, such as that provided by behavioral diagrams, can allow for more precise detection especially for behavioral-type patterns [12]. However, almost all approaches that include behavioral information in their calculations require implemented code for either static or dynamic analysis. Early on in the software life cycle, or in model-centric environments, code may not be available. In addition, these approaches do not reap the benefits of model-driven pattern detection [36].

One form of reasoning about models involves using logic programming [24], a paradigm that describes a domain as rules and facts. In this position paper, we propose an approach using a form of logic programming called answer set programming (ASP) in order to analyze structural and behavioral UML diagrams in order to detect patterns. In contrast to existing work, this approach does not depend on code and has the potential to detect patterns containing behavioral aspects. A limitation of this work is that it requires both UML Class and Sequence diagrams, however, these diagrams are among the most commonly used in industry [19].

[*]Corresponding Author

MiSE'16, May 16-17, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4164-6/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2896982.2896991

We begin this position paper in Section 2 by presenting background information on software patterns and answer set programming. This is followed by a description of related work and how our work differs to that in the literature. The ideas for our proposed method are described in Section 3, including our choice of answer set programming solver and the various stages in the process. Section 4 presents our validation plan, our expected results, and what we plan on having ready for the workshop. We then conclude in Section 5.

2. BACKGROUND AND RELATED WORK

2.1 Software Patterns

Design patterns are used to describe abstract solutions for frequent problems and desired properties in software engineering. Antipatterns are bad practices and negative properties that engineers should strive to avoid and correct when found. Together, these can be termed software patterns. There are patterns that deal with specific domains, like Java Enterprise systems [8] and multi-agent systems [29]. There are also more general-purpose sets of patterns, like the Gang of Four Patterns [12], or those dealing with performance issues and standards [40]. The definition for a software patterns includes the context of the problem and a general form of the solution, often in model form, allowing it to be applied in multiple ways. From a software evaluation and quality perspective, confirming the presence of design patterns or absence of antipatterns can be used as evaluation metrics, for example, design patterns in architectural evaluation [16] and antipatterns in Java projects [39] or Simulink [35].

2.2 Answer Set Programming

Answer set programming is a form of logic programming that is declarative. It is specifically geared towards complex search problems, including those that are NP-hard [23]. While is very similar in syntax to the popular Prolog language [7], the underlying computational foundation of ASP is quite different. Specifically, ASP's semantics are grounded in the stable logic programming model [14] and it employs answer set solvers to realize its search techniques. While ASP is rooted in first order logic, it belongs to the class of non-monotonic logic in that the addition of new information to an ASP program can cause predicates previously believed to be true to be retracted. This is a powerful feature that allows natural ASP representations of statements found in natural language and exceptions through the use strong negation and default negation.

An answer set [15] program encodes the set of beliefs of an intelligent agent. It consists of rules of the type "*head* ← *body*" read as "believe *head* if you believe *body*". *head* is a literal, such as an atom or its negation in first order logic, and *body* is a set of literals, possibly preceded by "not", which is read as "there is no reason to believe". Atoms and literals express properties of domain objects and relationships between objects. These ASP program models, called answer sets, consist of those literals that are believed to hold. A program may have multiple answer sets, each corresponding to a belief set of the agent. Answer sets are computed by inference systems called solvers.

ASP is especially suitable for representing qualitative knowledge, such as the knowledge we plan on encoding in Class and Sequence diagrams, default statements and their exceptions, dynamic domains in which change is triggered by the occurrence of actions, and uncertainty.

2.3 Related Work

We categorize pattern instance detection approaches into three groups based on how they represent the system under study and the patterns. There are techniques that transform the system to a graph and perform graph matching, techniques representing a system as a matrix and search for matrix representations of patterns, and techniques describing the system as logic facts and the patterns as logic rules.

2.3.1 Graph Representation

A class diagram can be considered a directed graph where classes correspond to nodes and the edges correspond to associations between the classes [38]. A pattern can be represented similarly, that is, the roles in the patterns are represented as nodes and the associations between those roles are represented as edges. These pairs of graphs can then be compared and checked. Tsantalis et al. [38] implement this by calculating the similarity between any two vertices of the system with the pattern. The pair that has the highest similarity score is considered to be a match. These graphs for both the patterns and systems can also be represented as abstract syntax trees (AST) containing information about classes, attributes, methods, and class inheritance [1, 37]. Each node can be unique based on metrics including number of private, public, and protected methods; number of attributes; associations; dependencies; et cetera. These graph representations can be obtained through static analysis or small-scale dynamic program execution [42].

2.3.2 Matrix Representation

Another form of representation involves using matrices. For example, system classes can be represented as an n x n matrix, where n is the number of classes. A pattern with m entities/roles can be represented in an mxm matrix. For example, the adapter design pattern [12] has four roles that would entail a 4 x 4 matrix: Client, Target, Adapter, and Adaptee. Each cell within the matrix has values indicative of factors such as number of methods in the class, if it is inherited, variables, and others. Dong et al. [10] check for similar values in the system matrix and the pattern matrix by calculating if classes are multiples of one another, utilizing prime numbers as the cell values. Tsantalis et al. [38] eventually transform their graphs into matrices for addition metrics. These approaches both require access to source code.

2.3.3 Logic Representation

Mikkonen demonstrated that formalizing aspects of patterns can assist in design pattern reasoning, and provided various insights and examples of formalizing design patterns [25]. In this category of approaches, the system undergoing analysis is formalized as facts and the patterns as rules. The inference engine working on these representations can then find which facts satisfy the rules specified by the patterns. More specifically, the rule variables are the constants, such as a class or an object, that satisfy the rules for the pattern being searched for. Those that satisfy the rules can then be viewed as a candidate pattern instance. Kramer and Prechelt [20], Heuzeroth et al. [17], De Lucia et al. [9], Bergenti et al. [4], and Birkner [5] have all used logic representations to detect patterns. Kramer and Prechelt were the first and their work does not consider behavioral aspects. The remaining three approaches are able to detect structural and behavioral patterns however their work requires access to the source code for that. Zhu et al. [43] have work that closely parallels ours and we discuss it later.

While the Object Constraint Language (OCL) seems as if it would be suitable for this task and representation as it can describe rules for UML models, it is unable to represent the absence of class relationships and is not ideal for meta-level descriptions [11].

2.3.4 Summary and Relation to Proposed Method

All work we found in the literature performs some form of structural analysis on models. Some of that work is also capable of behavioral analysis and/or including behavioral diagrams. Of the work that considers behavioral aspects, nearly all of it requires source code. This contrasts the technique we propose in this position paper as we focus entirely on the modeling level, allowing for early analysis in the software engineering process and evaluation of models in a pure model driven environment.

Bergenti et al. [4] take Class diagrams as input like we do. However, to get their behavioral information, they use Collaboration diagrams whereas we use Sequence diagrams. Firstly, Sequence diagrams are more commonly used in industry than Collaboration diagrams [19]. Secondly, Sequence diagrams are more helpful in detecting pattern instances as they are more concerned with temporal aspects. In addition, Sequence diagrams have already been explicitly defined for many patterns [5, 12].

SPASS versus ASP.

The work that is most similar to ours is that of Zhu et al. [43]. They use first-order predicate logic in order to represent patterns as predicates on UML diagrams. Their tool, LAMBDES-DP, translates UML models into first order logic and integrates an automated theorem prover called SPASS [41] for inference. For their experiments, they created a repository of the Gang of Four(GOF) patterns. A key limitation of their work they acknowledge is that SPASS can run forever/time out without any results due to inference being NP-hard for first order logic reasoning. For example, they experienced twenty four time outs in their experiments. The approach we describe in this position paper employs ASP, which always terminates in principle [23] and is tailored to these type of search problems. In addition, since ASP is non-monotonic and uses two negation types, pattern statements, such as "Normally, the state objects in the state pattern are singletons", and exceptions can be represented naturally.

There is an important distinction between ASP and automated theorem provers, like SPASS. Theorem provers focus on what is known and are goal directed, moving forward towards what is being proven. In contrast, ASP has no real notion of "forward" or "backward" inference. An ASP program simply encodes an agent's beliefs. Thus an ASP program may have multiple answer sets, where each answer set stands for a possible set of beliefs of an agent. For example, if specific conditions are met in a diagram, an intelligent agent could infer either that X design pattern is present or Y design pattern is present. If such situations occur where a set of features may indicate more than one design pattern, then ASP will be able to identify all of these possibilities, whereas a theorem prover will not.

Regarding input size of the systems and patterns, we contend that ASP can handle larger inputs and is more efficient. SPASS has a limit on the number of rules that it can handle, while ASP does not. However, ASP computation can be difficult when there are specific cycles, such as default negation cycles, or function symbols producing infinite inputs when grounding. An additional disadvantage of ASP is that it is not suitable for representing quantitative information especially when reasoning with large numerical domains or real numbers. The Class and Sequence diagrams we will be considering should not exhibit these features.

In addition, while Zhu et al. validated their work using the GOF patterns only, we plan on considering those in addition to other patterns, as we discuss in Section 4.1.

3. METHOD OVERVIEW

Once completed our proposed method involves a number of phases and is illustrated in Figure 1. In this section, we step through our vision of the process and how we plan on realizing each step. We include a small contrived example of a State Pattern [12] instance in a system[1] to help demonstrate. Its class diagram is presented in Figure 2. Using the code included in the example system, we were able to reverse engineer a sequence diagram using Visual Paradigm[2], which we present an excerpt of in Figure 3.

3.1 Generation of the Rules and Facts

The first milestone in the process is to generate the ASP facts and rules. Represented by the first set of arrows on the left in Figure 1, this involves a combination of manual and automatic effort.

3.1.1 System Facts

Our goal is to make fact generation an automatic process whereby systems being analyzed have their class diagrams and sequence diagrams represented as ASP facts. The form of models we plan on requiring are those represented in XMI. This is due to XMI's prevalence in the modeling community and, from an implementation and experimentation perspective, we can produce example/test UML models and export them to XMI through available tools such as StarUML [21]. So our main challenge here is writing an appropriate transformation from XMI to ASP. Seeing as Shan and Zhu were able to create analogous mappings from XMI to first order logic [32] that formed the basis of LAMBDES-DP [2, 3], we believe this is feasible. We plan on leveraging their work as well as Mikkonen's [25]. Most of their work focuses on the GOF patterns, but we can still apply the general ideas.

Using our example from Figures 2 and 3, we demonstrate some representations of facts pertinent to state pattern detection. For the class diagram in Figure 2, we present facts we derived, manually at this point, on operations. Specifically, we represent classes in the form of *class(name, [list of attributes], [list of operations], isAbstract)* and operations in the form of *operation(class_Name, operation_Name, return_type, [list of parameters], isAbstract, isConstructor, isStatic)*. Using those meta representations, some example facts about classes and operations from the class diagram

[1] http://www.tutorialspoint.com/
[2] http://www.visual-paradigm.com/

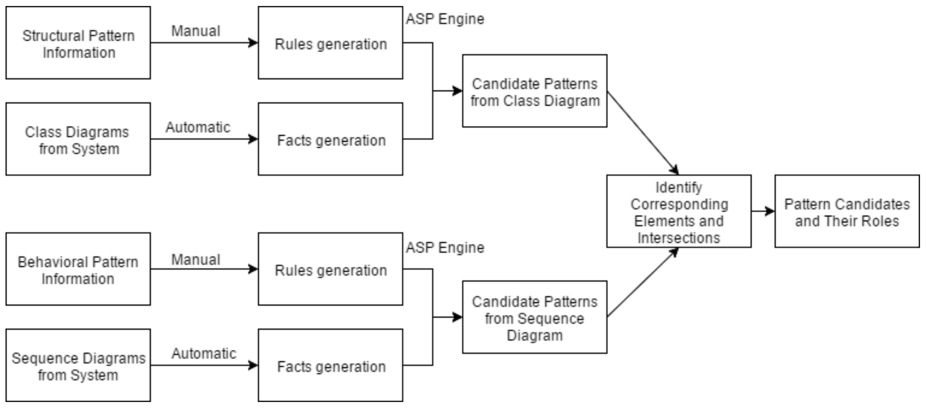
Structural Pattern Information
Manual
Rules generation
ASP Engine
Class Diagrams from System
Automatic
Facts generation
Candidate Patterns from Class Diagram
Identify Corresponding Elements and Intersections
Pattern Candidates and Their Roles
Behavioral Pattern Information
Manual
Rules generation
ASP Engine
Sequence Diagrams from System
Automatic
Facts generation
Candidate Patterns from Sequence Diagram

Figure 1: Proposed Process

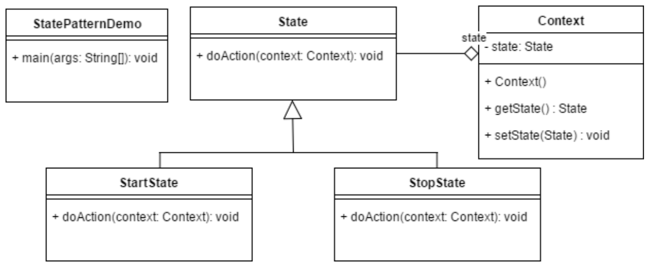
StatePatternDemo
+ main(args: String[]): void
State
+ doAction(context: Context): void
state
Context
- state: State
+ Context()
+ getState() : State
+ setState(State) : void
StartState
+ doAction(context: Context): void
StopState
+ doAction(context: Context): void

Figure 2: Class Diagram of State Pattern Instance

that our proposed process can derive from a diagram's XMI is included in Representation Set 1. It is important to remember that constants begin with a lowercase letter in ASP. The single included class is the *context* concrete class that has an attribute of type *state*, and methods called *context*, *getState*, and *setState*. The first operation represents the *doAction* abstract method of class *state*, which takes a *context* object as a parameter. The second represents the concrete implementation of the first operation, located in the class *startState*. The fact that it is the concrete implementation would be deduced later during the rule evaluation. The last operation involves the *context* class method that sets the state given a provided state.

$$\begin{gathered}
class(context, [state], [context, getState, setState], no).\\
operation(state, doAction, void, [context], yes, no, no).\\
operation(startState, doAction, void, [context], no, no, no).\\
operation(context, setState, void, [startState], no, no, no).
\end{gathered} \tag{1}$$

Sequence diagrams consist of lifelines describing specific entities, such as classes or class instances; messages; and a relative order that the messages are delivered. For sequence diagrams messages, we might represent our ASP facts in the form *message(messageID, sender, receiver, operation_invoked)*. Using this form, and the concrete sequence diagram of our system in Figure 3, we can synthesize facts such as those we present in Representation Set 2. The first message represents the *doAction* call from a *statePatternDemo* object to a *startState* object. The second message is triggered by the first as indicated by the identical first two numbers. This information is important because it allows us to keep track of what requests get called by which handlers and helps build a hierarchy of message calls. The third message describes the similar call of *doAction* after the state has been switched to a *stopState* object. The fourth follows the same idea as the second message, allowing us to represent requests, triggers, and handlers.

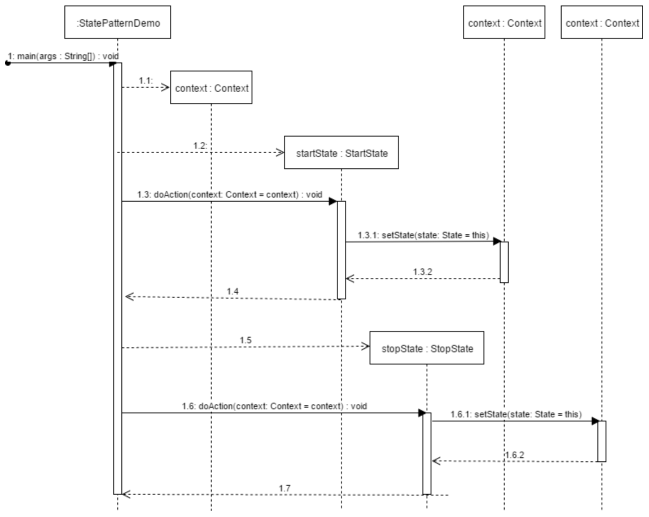
:StatePatternDemo
context : Context
context : Context
1: main(args : String[]) : void
1.1:
context : Context
1.2:
startState : StartState
1.3: doAction(context: Context = context) : void
1.3.1: setState(state: State = this)
1.3.2
1.4
1.5
stopState : StopState
1.6: doAction(context: Context = context) : void
1.6.1: setState(state: State = this)
1.6.2
1.7

Figure 3: Sequence Diagram of State Pattern Instance

$$\begin{gathered}
message(13, statePatternDemo, startState, doAction).\\
message(131, startState, context, setState).\\
message(16, statePatternDemo, stopState, doAction).\\
message(161, stopState, context, setState.
\end{gathered} \tag{2}$$

3.1.2 Pattern Rules

Developing the structural and behavioral pattern rules is a manual process. This is because the specific patterns in scope have to be selected by analysts and then transformed into the ASP rules. While it is analogous to fact generation, generating the rules is much more complex and would involve pattern inference. The only way this inference could be automated was if there existed instantiations of general pattern models [36] or one established general model that can be mapped and transformed into ASP rules. Since rule generation needs to be done only once for each rule of interest, having this process be manual is acceptable. In addition, rules can be updated and improved manually over time.

We continue with our example of the State pattern, which has both structural and behavioral aspects. It has a number of requirements [3], as outlined by Bayley and Zhu. For example, requirements on components include 1) Contexts and states are classes, 2) Requests are the operations of the context, and 3) Handlers are the operations of the state. Static conditions identified by Bayley and Zhu include 4) There is an aggregation between Context and State and 5) All handlers must be abstract. Dynamically, 6) Every request is handled by a handler and 7) Every handler either returns or sends a message including the new state as a parameter. We provide some example ASP rules we devised that encompass a selection of these requirements in Representation Set 3, where the numbers on the left represent the requirements being addressed and the variables are capitalized as necessitated in ASP. In these ASP rules the "_" symbol represents a wild card, meaning the ASP solver accepts anything in that place. The rule addressing 2) ensures that if there is a class that is a context class and it has a list of operations, then each of those operations are viewed to be "requests". For requirement 3), the rule is similar to the rule from requirement 2), in that all of the operations of a state class are called handlers. The last example rule, addressing requirement 5), ensures that every handler must be an abstract operation, hence the "no" in the abstract operation parameter.

$$\begin{aligned}
2)\mathit{isRequest}(\mathit{Op}) : -\mathit{class}(C,\_,\mathit{OpList},\_), \mathit{isContext}(C),\\
\#\mathit{member}(\mathit{Op},\mathit{OpList}), \mathit{operation}(\_,\mathit{Op},\_,\_,\_,\_,\_).\\
3)\mathit{isHandler}(\mathit{Op},C) : -\mathit{class}(C,\_,\mathit{OpList},\_),\\
\mathit{isState}(C),\\
\mathit{member}(\mathit{Op},\mathit{OpList}),\\
\mathit{operation}(\_,\mathit{Op},\_,\_,\_,\_,\_).\\
5)\mathit{isHandler}(\mathit{Op},C) : -\mathit{operation}(C,\mathit{Op},\_,\_,\mathit{no},\_,\_).
\end{aligned} \tag{3}$$

3.2 ASP Engine

3.2.1 Choice of ASP Solver

There are a number of implementations of ASP solvers, but two have consistently performed better than others in ASP competitions: Clingo [13] and DLV [22]. Both are inspired by the Davis-Putnam-Logemann-Loveland (DPLL) SAT solving backtracking algorithm and use other techniques such as back-jumping, restarts, et cetera.

In terms of the specific ASP dialect that each solver accepts, there are important differences. In DLV, there are built-in functions for lists and the Prolog notation for lists is applicable. Moreover, DLV has a more natural and intuitive notation for aggregate functions that is closer to mathematical notation. Aggregates also exist in Clingo, but they are somewhat cumbersome and less straightforward. Queries can be specified in a DLV program, but not in Clingo. DLV supports two types of query answering capabilities: (1) cautious, that is, something is true if it is true in all answer sets of a program and (2) brave, that is, something is true if it is true in at least one answer set of the program.

Writing mathematical expressions in the language of DLV is difficult, as literals consisting of arithmetic expressions are required to contain only one arithmetic operation. Clingo has much better support for arithmetic expressions, for example, arithmetic operations can be concatenated. Finally, Clingo introduces the concept of "choice rules" that allow for easy specification for indicating that answer sets should contain a certain number of literals of a given set. The same is not always as easy to encode in DLV.

At this stage, we are planning on using the DLV solver because of its support for lists, queries, and its natural syntax for aggregates. All of which will be important for our purposes.

3.3 Element Alignment and Candidate Intersection

This stage of the process involves interpreting the output of the ASP solver. The output will indicate structural and behavioral elements that adhere to rules describing structural patterns and behavioral patterns, respectively. One challenge here is to align the Class diagram elements with the corresponding Sequence diagram elements. In cases where they are explicitly linked or cross-referenced, as is done in tools like Rational Software Architect[3] this is relatively easy. In other cases signature and element mappings can be developed to link the appropriate diagram elements [43]. Once elements are linked, we need to calculate the intersection of the structural pattern instances and behavioral ones. For example, if there is a structural pattern instance of a specific pattern, Y, found among a set of classes, X, we need to determine if there is a behavioral pattern instance for pattern Y found among the X set of classes. Since the rules specify roles, we will be able to explicitly identify the roles of the pattern participants.

3.4 Presentation of the Pattern Candidates and Roles

Presentation to the analysts of the potential pattern candidates and the corresponding roles is an important consideration as it can dictate how the results are used. The simplest approach would be a textual representation, like XML, that has a <pattern> tag that indicates the type of pattern found, and an <element> tag that can indicate the universally unique ID (UUID) of the UML element, and its role(s) within the pattern.

A long term goal would be to integrate the results into an existing UML graphical viewer. For example, an analyst could click on specific pattern instances, and using colorization, the applicable UML elements can be highlighted, indicating different roles through labels or shading. This is possible because of UML's use of UUIDs. We could potentially leverage existing tools such as VizzAnalyzer that help visualize structural and behavioral architectural analysis [18].

4. DISCUSSION

4.1 Validation Plan

For general validation, we will inject some pattern instances in larger systems, potentially mutating them slightly for variation as we have done in other validation and evaluation frameworks [33, 34]. The main validation will come from comparing our results against results from existing approaches. In many cases we can use the systems and patterns described in their work. While our technique focuses solely

[3]www-03.ibm.com/software/products/en/ratisoftarch

on the model-level, we will be able to contrast our results to those using source code by reverse engineering the Class and Sequence diagrams from the code used in their experiments using transformation tools, such as Visual Paradigm. In addition to incorporating the Gang of Four patterns [12] set, we plan on formalizing some of Brown's antipatterns [6] and UML architectural patterns [31].

4.2 Expected Results

For approaches performing model evaluation that focus solely on structural diagrams, we expect that our tool will have higher precision detecting instances of behavioral patterns. That is, of all the potential pattern instances identified, more of them will be "correct". This is because we are considering more information and increasing the requirements for potential matches. Ideally, the recall, meaning number of true pattern instances identified from the systems under study, will be relatively consistent. However, this will be a challenge and is a matter of tuning the pattern specification rules.

Contrasting with LAMBDES-DP, our goal is to improve on their false-positive error rate, which they attributed to their time outs. So, we anticipate higher recall than their tool, accounting for any potential pattern instances that they would miss. The precision should be comparable, assuming the formalizations are similar. Because we both apply and extend the predicate logic formalisms provided by Bayley and Zhu [2] for the UML systems and the patterns, we believe this will be the case.

In contrast to source-code approaches, there is a good chance they will have better precision. This is understandable due to source code containing much more detailed information and being able to perform a much more discerning analysis. From a recall perspective, depending on the realization of the patterns in the projects, it is possible that patterns that were intended to be included by engineers were not due to implementation error. Models are more abstract than source code, so the concrete source code artifacts being compared have more room for differences/noise. But, for the most part, the recall of source code approaches should be comparable to our work employing ASP.

4.3 Preliminary Results for the Workshop

By the time of the workshop, we plan on having implemented this process on a small scale. Specifically, we want to implement fact generation for UML models, and develop tested and tuned pattern rules for a handful of patterns. Some low-hanging fruit that we are starting with is the Strategy and State pattern as structural analysis approaches are unable to discern between the two due to their similar structure.

5. CONCLUSIONS

Being able to evaluate software at the modeling level has many benefits. One way of evaluating models is by detecting positive and negative quality indicators in models in the form of design patterns and antipatterns, respectively. Much of the existing work in pattern detection and analysis requires completed source code in order to generate models or to fill in informational gaps, and focuses solely on structural pattern aspects. In this position paper, we presented our initial ideas on a technique using answer set programming in order to perform analysis directly on the models themselves in lieu of source code.

Representing the structural and behavioral pattern aspects being searched for as ASP rules and systems to be analyzed as facts, we plan on identifying potential pattern instances efficiently and with more accuracy, especially compared to approaches that consider only structure. The work most similar to ours uses first order logic and SPASS for inference. Using ASP has three advantages over SPASS: ASP will always terminate; ASP will be able to identify multiple and all formalized possibilities, whereas a theorem prover will not; and ASP can handle larger input and more rules than SPASS.

Interesting research aspects and milestones of this work include formalizing the patterns as ASP rules, automating the transformation of UML Class and Sequence diagrams as ASP facts, aligning elements after inference, and presenting the results to analysts. To validate our work, we plan on comparing against both model-based techniques, and those that require source code by reverse engineering models from the code. It is our belief that using ASP in this manner will stimulate interesting modeling research and help facilitate modeling in software engineering by improving model analysis and evaluation.

6. ACKNOWLEDGMENTS

We would like to thank Dr. Gerald (Jerry) Gannod and Dr. Dhananjai (DJ) Rao for their guidance on this work, both on the related work aspects and for their suggestions on the approach.

7. REFERENCES

[1] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *International Workshop on Program Comprehension*, pages 153–160. IEEE, 1998.

[2] I. Bayley and H. Zhu. Formalising design patterns in predicate logic. In *International Conference on Software Engineering and Formal Methods*, pages 25–36. IEEE, 2007.

[3] I. Bayley and H. Zhu. Formal specification of the variants and behavioural features of design patterns. *Journal of Systems and Software*, 83(2):209–221, 2010.

[4] F. Bergenti and A. Poggi. IDEA: A design assistant based on automatic design pattern detection. In *International conference on Software Engineering and Knowledge Engineering*, pages 336–343, 2000.

[5] M. Birkner. Objected-oriented design pattern detection using static and dynamic analysis in Java software. Master's thesis, University of Applied Sciences Bonn-Rhein-Sieg, 2007.

[6] W. H. Brown, R. C. Malveau, and T. J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley, 1998.

[7] W. Clocksin and C. S. Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003.

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

[9] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi. Improving behavioral design pattern detection through model checking. In *European Conference on*

Software Maintenance and Reengineering (CSMR), pages 176–185. IEEE, 2010.

[10] J. Dong, D. S. Lad, and Y. Zhao. Dp-miner: Design pattern discovery using matrix. In *International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 371–380. IEEE, 2007.

[11] R. B. France, D.-K. Kim, S. Ghosh, and E. Song. A UML-based pattern specification technique. *Transactions on Software Engineering*, 30(3):193–206, 2004.

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

[13] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Answer set solving in practice. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(3):1–238, 2012.

[14] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.

[15] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New generation computing*, 9(3-4):365–385, 1991.

[16] N. B. Harrison and P. Avgeriou. Leveraging architecture patterns to satisfy quality attributes. In *Software Architecture*, pages 263–270. Springer, 2007.

[17] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe. Automatic design pattern detection. In *International Workshop on Program Comprehension*, pages 94–103. IEEE, 2003.

[18] D. Heuzeroth and W. Löwe. Understanding architecture through structure and behavior visualization. In *Software Visualization*, pages 243–286. Springer, 2003.

[19] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *International Conference on Software Engineering*, pages 471–480. ACM, 2011.

[20] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Working Conference on Reverse Engineering*, pages 208–215. IEEE, 1996.

[21] M. Lee, H. Kim, J. Kim, and J. Lee. StarUML 5.0 developer guide. *The Open Source UML/MDA Platform*, 2005.

[22] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *Transactions on Computational Logic*, 7(3):499–562, 2006.

[23] V. Lifschitz. What is answer set programming?. In *AAAI*, volume 8, pages 1594–1597, 2008.

[24] J. W. Lloyd. *Foundations of logic programming*. Springer Science & Business Media, 2012.

[25] T. Mikkonen. Formalizing design patterns. In *International conference on Software engineering*, pages 115–124. IEEE Computer Society, 1998.

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

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

[28] C. V. Ramamoorthy and S.-b. F. Ho. Testing large software with automated software evaluation systems. In *ACM SIGPLAN Notices*, volume 10, pages 382–394. ACM, 1975.

[29] S. Sauvage. Design patterns for multiagent systems design. In *MICAI: Advances in Artificial Intelligence*, pages 352–361. 2004.

[30] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):0025–31, 2006.

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

[32] L. Shan and H. Zhu. A formal descriptive semantics of UML. In *Formal Methods and Software Engineering*, pages 375–396. Springer, 2008.

[33] M. Stephan. Model clone detector evaluation using mutation analysis. In *International Conference on Software Maintenance and Evolution*, pages 633–638. IEEE, 2014.

[34] M. Stephan. *A Mutation Analysis Based Model Clone Detector Evaluation Framework*. PhD thesis, Queen's University, August 2014. http://qspace.library.queensu.ca/handle/1974/12376.

[35] 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*, pages 276 – 285, 2015.

[36] 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*, pages 48–53, 2015.

[37] P. Tonella and G. Antoniol. Object oriented design pattern inference. In *International Conference on Software Maintenance*, pages 230–238. IEEE, 1999.

[38] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *Transactions on Software Engineering*, 32(11):896–909, 2006.

[39] E. Van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Working Conference on Reverse Engineering*, pages 97–106, 2002.

[40] A. I. Verkamo, J. Gustafsson, L. Nenonen, and J. Paakki. Design patterns in performance prediction. In *Workshop on Software and Performance*, volume 2000, pages 143–144, 2000.

[41] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In *International Conference on Automated Deduction*, pages 140–145. Springer, 2009.

[42] L. Wendehals. Improving design pattern instance recognition by dynamic analysis. In *Workshop on Dynamic Analysis*, pages 29–32, 2003.

[43] H. Zhu, I. Bayley, L. Shan, and R. Amphlett. Tool support for design pattern recognition at model level. In *International Computer Software and Applications Conference*, volume 1, pages 228–233. IEEE, 2009.