An Automated Software Tool for Validating Design Patterns

Shane Strasser, Colt Frederickson, Kevin Fenger and Clemente Izurieta
Department of Computer Science
Montana State University
Bozeman, MT 59717
{shane.strasser, coltf, kevin.fenger, clemente.izurieta}@cs.montana.edu

Abstract

Design pattern languages have started to gain more attention by providing the ability to specify instances of patterns. The Role Based Metamodeling Language (RBML) is a visually oriented language defined in terms of a specialization of the UML metamodel that is used to verify and specify generic or domain specific design patterns. The evolution of increasingly complex computer applications is mandating the development of technologies and tools that can validate the intended design and can alert developers of potential changes that were not intended by the original designers. Deviations from design patterns occur as a result of a lack of understanding of intended designs or through evolutionary changes of the software; thus an automated tool is needed to inform developers when design patterns no longer conform to their original intended design. Herein, we present an automated software tool that compares the UML class diagrams of instances of design patterns with their RBML representations and reports back if said UML diagram is compliant. The tool can identify which classifiers (classes or associations) of the design pattern are in violation with their RBML description. Furthermore, the tool produces a score that is indicative of the level of compliance. We describe the algorithm that compares the UML representing a design pattern to its corresponding RBML and provide several scenarios that demonstrate the capabilities of this tool.

1 Introduction

Improving software development cycle-time and quality has been the subject of many research studies. One way this can be done is through the reuse of high quality domain-specific artifacts and design patterns [1]. Kim et al. [2] introduced the Role-Based Metamodeling Language (RBML) for characterizing generic and domain-specific design patterns. RBML allows a user to specify a design pattern which can then be instantiated in UML [3]. The relationship between the RBML specification and UML compliant instances of design patterns is one-to-many. RBML allows for the precise description of high quality domain-specific artifacts and design patterns, which aids with quality control of systems by providing us with the ability to compare UML diagrams of patterns to their RBML specifications and determine if a project is deviating from its intended design.

A problem with using RBML to develop design patterns however; is that it may not always be obvious if a UML implementation completely conforms to its RBML specification. Also, if the UML model of a pattern does not conform to its RBML specification, a developer will want to know which classifiers of the UML model are in violation with the RBML specification.

There has been little work done in creating an automated tool for validating instances of design patterns. In 2005, Kim [4] proposed a basic algorithm but did not provide any test results. In 2008, Kim and Shen [5] presented a divide-and-conquer algorithm along with several case studies. In this paper, we extend their seminal work by implementing an automated testing tool for comparing RBML and UML diagrams. Additionally, our tool produces a score which describes how well a UML diagram matches its RBML specification. We present the divide and conquer algorithm augmented with a scoring engine. Finally, we give some scenarios that demonstrate how the tool can be used to refine UML diagrams that do not completely conform to their RBML specification.

2 RBML

The Role-Based Metamodeling Language, (RBML) was developed in 2003 as a way of expressing domain specific design patterns which can be instantiated as UML diagrams [2]. By having a standard language to specify design patterns, a developer is constrained by a set of rules when creating a UML diagram for a particular design pattern, resulting in

better quality code. For example, during a project a user decides to use a specific design pattern. With RBML, the user is able to specify the one original design pattern and then instantiate the design pattern in many different ways-all compliant with their RBML specification. As projects grow however, they may start to deviate from their original design pattern which results in increases in maintainability. Using the RBML specification of the pattern, developers are able to compare the RBML specification with that of the UML diagram to see if and where the project may be deviating from the original design, thus resulting in higher quality code.

RBML is based upon UML and uses the same syntax as UML. It consists of a number of behavioral and structural diagrams with each one describing different parts of the design pattern. Whereas a UML diagram has classes and interfaces, an RBML diagram has classes (which represent classes) and classifiers (which represent interfaces and abstract classes). Within each class and classifier, the RBML has behaviors and attributes, which represent methods and attributes in a UML model instantiation. RBML also has generalization, association, and dependency relationships between the different classifiers and classes which can also be instantiated in UML model representations. Finally, RBML specifications can have multiplicity constraints on attributes, behaviors, and relationships [2] [5].

An example UML diagram and its corresponding RBML specification are shown in Fig. 1. In this example, we have two RBML classes: Subject and Observer. Both have attributes and behaviors and there is an Association Role (relationship) between the two classes. On the right hand side of the diagram we display the UML instantiation, which has three classes, two of which play the observer role. Additionally, we see that each class has the required attributes and behaviors, making this UML diagram a valid instantiation of its corresponding RBML specification.

3 Comparing RBML and UML Diagrams

To compare an RBML and a UML diagram, we used the divide-and-conquer algorithm developed by Kim and Shen [5]. The algorithm works as follows: first the RBML and UML diagrams are broken up into blocks, which are defined as any two classes or classifiers (classes and interfaces in UML) which have a relationship between them. Because there are three kinds of relationships, there are three different kinds of block types: association blocks, generalization blocks, and dependency blocks. In the example in Fig. 1, there is only one RBML block (since there are only two classes). In the UML diagram, there are two blocks: one for the Kiln - TemperatureObs relationship and one for the Kiln - PressureObs relationship.

After all the blocks have been created, the algorithm first performs local conformity checks. Local conformity goes through RBML-UML block pairs and checks to see if the UML block satisfies the RBML block's local requirements by checking all the UML's block behaviors, attributes, and multiplicities to see if they satisfy those constrained in the RBML's block. If the UML block maps to the RBML block, the mapping is added to the set of global mappings. The pseudocode for local conformity is given in Algorithm 1.

Algorithm 1 Check For Local Conformance

```
CheckLocalConform (rbmlBlock R, umlBlock U)
if R.BlockType = U.BlockType then
  if Satisfy(U.Attributes, R.Attributes) then
    if Satisfy(U.Behaviors, R.Behaviors)) then
      return true
    end if
  end if
end if
return false
```

The first if statement checks to make sure both blocks are of the same type (i.e., generalization, association, or dependency). In the second and third if statements, the method "Satisfy" checks to see if the UML attributes (and behaviors) satisfy the RBML attributes. "Satisfy" takes into consideration the multiplicity constraints of the attributes and behaviors from the RBML diagrams when checking to see if the UML diagram matches.

After local conformity has been established, all of the mappings need to be checked for global conformance. Note that during local conformity, a UML diagram may map to several RBML blocks if the UML block satisfies all the RBML requirements. However, the structure of the local neighborhood for that particular UML block might not map to the local neighborhood of the RBML block it was mapped to. Therefore, the algorithm must go through all possible mappings found during local conformance and remove the ones that would violate the global structure.

Global conformity is performed by going through all the RBML to UML mappings found during the local conformity checks. For a possible map, all the RBML blocks that overlap with the selected RBML block in the mapping are found. Likewise, we find all the UML blocks that overlap with the UML block

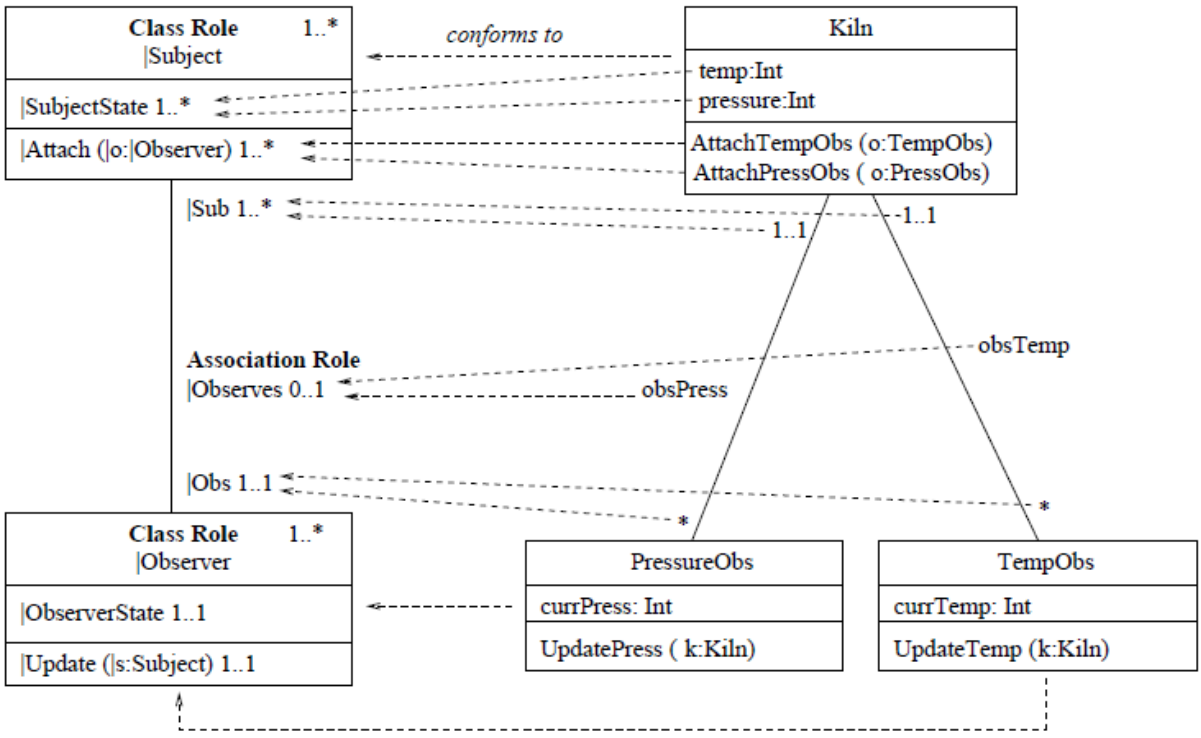

Fig. 1: An example of a RBML diagram on the left and a UML instance on the right [6].

selected from the mapping being observed.

For each UML block in the local neighborhood, we check to see if that block can be mapped to an RBML block from the local neighborhood of the RBML block in the mapping. If all of the UML blocks in the local neighborhood of the UML block in the mapping can be mapped to an RBML block in the local neighborhood of the RBML block in the mapping, we return true and the mapping is kept. If false, then the mapping of the RBML block to the UML block is removed. The pseudocode for the algorithm is given below in Algorithm 2.

Algorithm 2 Check For Global Conformance

```
CheckGlobalConform (rbmlBlock R, umlBlock U)
if   MapTo (OverLapBlcks (U) ,OverLapBlcks (R))
then
  return true
else
  return false
end if
```

In the algorithm shown above, *rbmlBock* and *umlBlock* are two blocks from a possible mapping found in local conformity. Overlapping blocks (OverLapBlcks) returns the local neighborhood of the *umlBlock* (or *rbmlBlock*). Method "MapTo" checks to see if all overlapping UML blocks can be mapped to an overlapping RBML block.

The final part of the algorithm checks the multiplicity of the various RBML classes. This is done by going through each RBML block and seeing how many UML blocks map to it. If the number of UML blocks that map to the RBML block is not the correct number or does not fall within the value allowed by the specified multiplicity, then the mapping is set as false. The algorithm takes in a RBML block and checks to see if the number of UML diagrams that map to it are within the specified range. If true, then the algorithm returns true, otherwise it returns false.

The entire conformance algorithm can be found directly below. The algorithm goes through each RBML UML block pair and calls "CheckLocalConform" on the block pairs. If the local conformity algorithm returns true, then a mapping of the RBML to UML block is added to the set of mappings. Next, each mapping is iterated through and global conformity is checked by "CheckGlobalConform". The last step iterates over all the RBML blocks and checks the mapping to see if the multiplicity constraints are violated ("CheckMulti"). If the multiplicity is

violated, the mapping is set as false. The mapping is returned along with whether or not the mapping is set as false.

Algorithm 3 Conformance Check for UML to RBML

```
Conformance (RbmlDiag RBML, UmlDiag UML)
for all rbmlBlck R ∈ RBML do
  for all umlBlck U ∈ UML do
    if CheckLocalConform(R, U) == true then
      Mapping.Add (R → U)
    end if
  end for
end for
for all R → U ∈ Mapping do
  if CheckGlobalConform (R, U) == false then
    Mapping.Remove(M)
  end if
end for
for all rbmlBlock R ∈ RBML do
  if CheckMulti (rb, Mapping) == false then
    Mapping.Possible = false
  end if
end for
return Mapping
```

4 Scoring

We extend the work of Kim and Shen [5] by adding a score which describes how well the UML diagram matches its RBML specification. In developing the scorer, we came up with our own rules and decided on the following. First, an RBML violation (an RBML block with no UML blocks mapping to it) should be considered to be worse than a UML violation (a UML block that does not map to any RBML block). The former indicates that key functionality is missing from the design pattern while the latter indicates additional functionality deemed not essential to the design pattern. Second, both local and global scores are used to find the overall score.

To find the number of RBML violations, we took the number of RBML blocks that had no possible mappings to them (no UML block mapped to the RBML block) and divided that number by the total number of RBML blocks. This gives a percentage of RBML blocks that had no UML blocks mapped to them. Similarly, we found the number of UML violations by taking the total number of UML blocks that were mapped to an RBML block and divided that number by the total number of UML blocks in the UML diagram of the design. We then weighted the RBML violations to be twice as bad as the UML violations and created a total weighted score by multiplying the percentage of RBML blocks that had good mappings by 2/3. This was added to the weighted score of UML violations times 1/3. The following equation gives the score

$$Score = \frac{2}{3} * \frac{NumberOfSatisfiedRBML}{TotalNumberOfRBML} + \frac{1}{3} * \frac{NumberOfSatisfiedUML}{TotalNumberOfUML}$$

This calculation is performed after local and global conformity are established to determine our local and global scores. The total score is then found by adding the local conformity score times 1/4 to the global conformity score times 3/4. The equation $TotalScore = \frac{1}{4} * LocalScore + \frac{3}{4} * GlobalScore$ exemplifies how we calculate the total score using global and local conformity scores. We acknowledge that this score is based upon our own personnel opinions and experience and poses a construction validity threat to the study, but note that changing the score based upon other users preferences is parameterizable and thus possible.

5 Automated Tool

Using the previously described algorithm and scoring function, we created a tool that automatically compares the RBML and UML diagrams for the user and reports whether or not the diagrams match, the score, if there were any errors, and displays the two diagrams [1]. Fig. 2 displays a screen shot of the tool. On the left hand side ("Diagram Selection"), we provide the user drop down menus for choosing which RBML and UML diagrams to compare. In the middle panes of the tool ("RBML/UML Diagram Image"), the two diagrams are displayed (the RBML diagram is displayed on top and the UML diagram is displayed on the bottom). The results of running the algorithm are displayed on the right hand side of the tool. On the top section labeled "Error Log," the tool reports problems that the algorithm found, such as why one UML block did not match with an RBML block. Directly below in the box labeled "Pass or Fail," the tool reports whether there was a UML diagram found that conforms to the RBML diagram. To the right is the score calculated using our scoring equation described above.

[1] The automated tool is free and is available to download at `http://code.google.com/p/rbml-uml-visualizer/`.

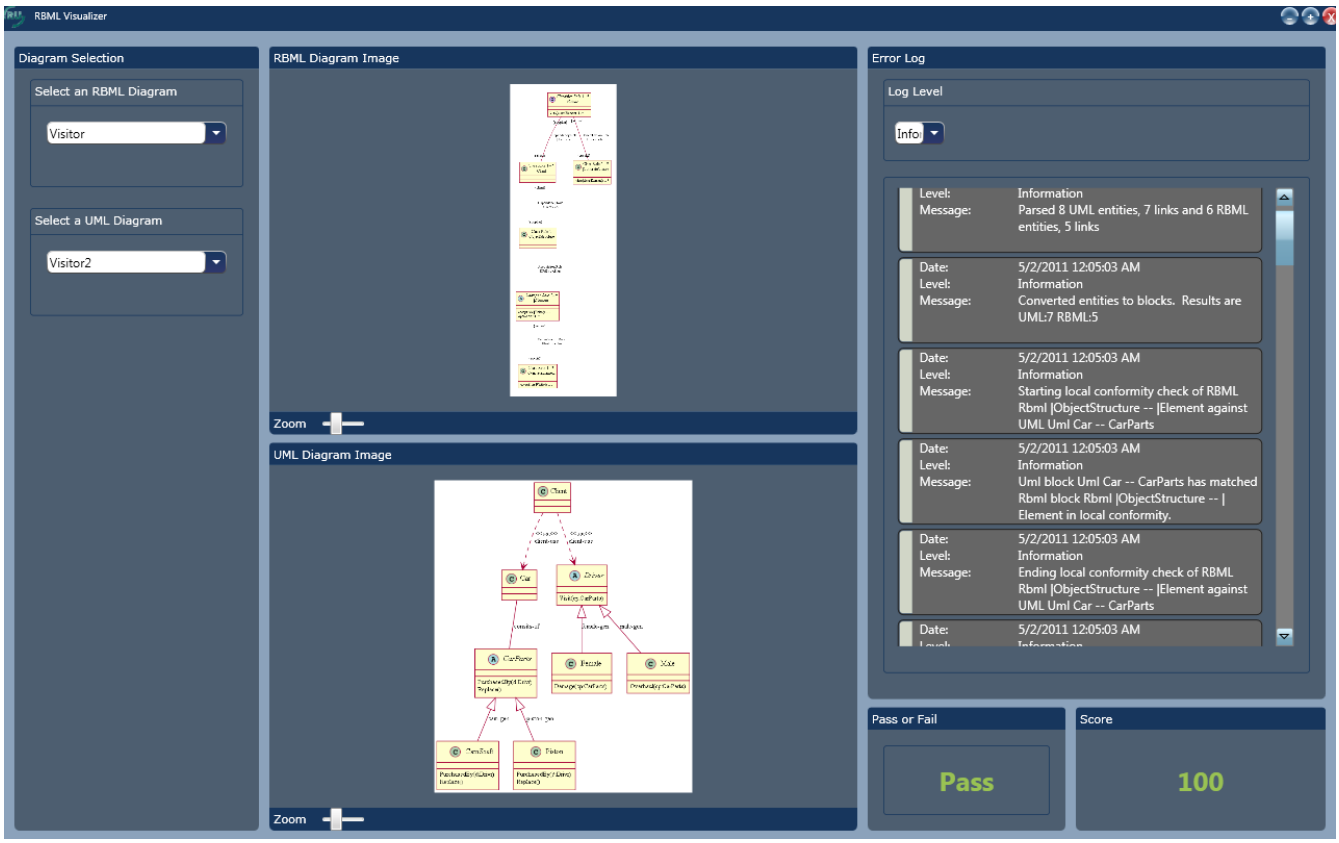

Fig. 2: A screen shot of the automated tool for comparing RBML and UML diagrams.

6 Demo

To test the functionality of the tool, we chose four different design patterns and created an RBML specification for each one. We chose the Bridge, Builder, Observer and Visitor design patterns [7]. For each of these design patterns, we created 4 different UML diagrams adhering to various levels of conformity. For each RBML specification (Bridge, Builder, Observer and Visitor), the first UML diagram (UML1) was built such that it was missing classes or classifiers that were required by the RBML. The second UML diagram (UML2) was built so that it conformed perfectly to the RBML diagram. The third (UML3) and fourth (UML4) UML diagrams also satisfied all of the RBML requirements, but had extra UML classes that did not conform to anything in the RBML specification. Additionally, we created the third and fourth UML diagrams such that the fourth UML diagram was larger than the third one and had just as many extra UML classes. This would have the affect of the fourth UML score always being greater than the third UML score because the percentage of satisfied UML diagrams in the fourth UML model will be greater than that of the third. Table 1 gives the results of running all the RBML and UML patterns through our tool and the corresponding scores.

Table 1: Scores reported from running our tool on selected RBML and UML diagrams.

RBML	UML1	UML2	UML3	UML4
Bridge	58.33	100.00	94.44	95.83
Builder	41.67	100.00	92.59	95.83
Observer	37.50	100.00	96.33	96.70
Visitor	32.50	100.00	96.30	96.67

As we can see from Table 1, we have the desired score for each UML diagram. For UML diagrams that are missing components required by the RBML, the scores are very low (less that 60%). In the perfectly conforming UML diagrams, the score was 100%. UML3 and UML4 both have high scores, but are not quite 100% because both have extra UML classes that do not map to anything in the RBML diagram. Additionally, UML4's score is always better than UML3's because it has a higher percentage of classes that conform to the RBML design pattern. The scoring algorithm was callibrated so that it matched our expec-

tions of a real world to mathematical representation of a mapping. This satisfied our empirical relation system as defined by [8] and reduced our threat to the validity of the study.

7 Conclusions

In this paper, we presented an algorithm and a tool that validate design patterns by performing RBML to UML model comparisons. The tool informs the user if the UML diagram satisfies the RBML diagram's requirements. In addition, the tool reports a score that tells the user the level of conformity of the UML diagram as compared to its RBML specification. This automated RBML-UML diagram comparison tool allows the user to see in the logs which parts of the RBML and UML models are not being satisfied. This can then be used to help a person develop better UML models when using a certain design pattern because the person can quickly find errors in their UML model. Computer and engineering applications can greatly benefit from continuous monitoring of deviations from intended designs.

In future work, we will explore approximate matching in the local conformity stage of the comparison algorithm. Currently, the algorithm only allows a UML object to be mapped to an RBML model if the UML satisfies all of the RBML blocks requirements. However, this could be relaxed so that the UML model that best satisfies the RBML blocks requirements is mapped to that RBML. In this way, if a user forgot to add a function or attribute in any one of the participating UML classes, the comparer would still be able to find the most likely location of which RBML class or classifier that said UML class would be mapped to. Additionally, this would result in a better score since the only thing the user may be missing is an attribute or behavior.

Acknowledgments

The authors would like to thank Shaun Ross of RightNow Technologies for his help in making this project possible.

References

[1] G. Arango and R. Prieto-diaz, "Introduction and overview: Domain analysis concepts and research directions," 1991.

[2] D.-K. Kim, R. France, and S. Ghosh, "A uml-based language for specifying domain-specific patterns," *Journal of Visual Languages & Computing*, vol. 15, no. 3-4, pp. 265–289, 2004.

[3] *OMG Unified Modeling Language 2.3*, OMG, 2010.

[4] D.-K. Kim, "Evaluating conformance of uml models to design patterns," in *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*. Washington, DC: IEEE Computer Society, 2005, pp. 30–31.

[5] D.-K. Kim and W. Shen, "Evaluating pattern conformance of uml models: a divide-and-conquer approach and case studies," *Software Quality Control*, vol. 16, pp. 329–359, sep. 2008.

[6] E. S. Robert France, Dae-Kyoo Kim and S. Ghosh, *Metarole-Based Modeling Lanugage (RBML) Specification V1.0*, 2002.

[7] R. E. J. Erich Gamma, Richard Helm and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[8] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed. Boston, MA, USA: PWS Publishing Co., 1998.