Analyzing Behavioral Refactoring of Class Models

Wuliang Sun, Robert B. France, Indrakshi Ray

Colorado State University, Fort Collins, USA

Abstract. Software modelers refactor their design models to improve design quality while preserving essential functional properties. Tools that allow modelers to check whether their refactorings preserve specified essential behaviors are needed to support rigorous model evolution. In this paper we describe a rigorous approach to analyzing design model refactorings that involve changes to operation specifications expressed in the Object Constraint Language (OCL). The analysis checks whether the refactored model preserves the essential behavior of changed operations in a source design model. A refactoring example involving the *Abstract Factory* design pattern is used in the paper to illustrate the approach.
Keywords: Behavioral refactoring, UML/OCL, Alloy

1 Introduction

In Model-Driven Development (MDD) projects, one can expect design models to evolve as developers explore design spaces for high quality solutions. Class models are among the most popular models used in practice and given their pivotal roles, there is a need to manage their evolution. Software refactoring [4][14] is an important class of changes that is applicable to class models. The goal of a refactoring is to improve software qualities such as maintainability and extensibility, while preserving essential structural and behavioral properties. A number of model refactoring mechanisms have been proposed (e.g., see [2][5][12][19][20][21]), and many (e.g., see [19][21]) provide support for checking whether structural properties are preserved in refactored models. However, we are not aware of any approach that supports rigorous analysis of behavioral properties when operation specifications in class models are added, removed, or modified. In this paper we describe a rigorous approach to analyzing the refactoring of design class models that involve changes to operation specifications expressed in the Object Constraint Language (OCL) [15].

The model on which a refactoring is performed is called the *source* model, and the model produced by the refactoring is called the *refactored* model. A refactoring that involves making changes to operation specifications is called a *behavioral refactoring*. In this paper, we present an approach to analyzing behavioral refactorings to check that changes to operation specifications preserve the net effect of the operation (i.e., its essential behavior) as specified in the source model. The net effect of an operation can be expressed using the OCL pre-/post-conditions.

As an example, consider a case in which the operation $FlightManager$ :: $bookFlight()$ in a flight reservation system class model is refactored into the following four operations in the refactored model: $Airline$ :: $getAvailableFlights()$ returns all flights that are available on a given day and airport, $Flight$ :: $getAvailableSeats()$ returns all seats that are available on the flight on a given day and airport, $Flight$ :: $reserveSeat()$ reserves a seat on the flight, and $FlightManager$ :: $bookFlight()$ books a flight by calling the previous three operations. The net effect of the $FlightManager$ :: $bookFlight()$ operation in the source model is specified using an OCL pre-/post-condition stating that if there exists available flight seats, at the end of the operation execution a seat will be reserved by a flight manager. The behavioral refactoring performed on the source model redistributes the functionality of $FlightManager$ :: $bookFlight()$ across different classes (i.e., $Airline$, $Flight$, and $FlightManager$). It is tedious to manually determine if the above behavioral refactoring preserves the net effect of the original operation because it involves manually building a description of the global net effect of a behavior by composing operation specifications that define subbehaviors in local contexts (i.e., classes in which the operations are located).

The above motivates the need for an automated analysis technique that supports rigorous analysis of behavioral refactorings. In the approach described in this paper, an analysis of a behavioral refactoring involves determining whether a sequence of operations in the refactored model preserves the net effect of an operation in the source model. The net effect of a source model operation is preserved by a sequence of refactored operations if the sequence starts in all the states that satisfy the pre-condition of the source model operation, and leaves the system in a state that satisfies the post-condition of the source model operation. The analysis approach requires the software modeler who performed the behavioral refactoring to provide a sequence diagram that describes the sequence of refactored operations. The approach takes the sequence of refactored operations, applies all the states that satisfy the pre-condition of the source model operation, and checks if the sequence of refactored operations produces any state that does not satisfy the post-condition of the source model operation. The net effect of the source model operation is not preserved by a sequence of refactored model operations if the sequence of refactored model operations starts in a state that satisfies the pre-condition of the source operation and produces a state that does not satisfy the post-condition of the source model operation.

The *Alloy Analyzer* [9] is used at the back end to statically analyze a behavioral refactoring. The analysis involves using the Alloy trace mechanism to determine whether operations in the refactored model can preserve the net effect of a changed operation specification in the source model. Since the Alloy Analyzer requires users to specify a bounded scope for each class, that is, the maximum number of instances that can be produced for a class, the analysis is performed within a bounded scope of class objects. The approach uses a UML-to-Alloy transformation to shield the software modeler from the "back-end" use of the Alloy Analyzer. Our transformation extends prior work on transforming UML to Alloy models [1][3][7][11][17] by providing support for transforming a

class model and a sequence diagram to an Alloy model that specifies behavioral traces.

The approach described in the paper is lightweight in that (1) it does not expose the modeler to any formal notation other than the OCL, and (2) the net effect preservation analysis is checked within a bounded domain. More heavyweight formal analysis techniques are needed in a setting where the net effect preservation checking requires more exhaustive analysis.

The rest of the paper is organized as follows. Section 2 provides an overview of the approach and Section 3 illustrates its use on a small example. Section 4 presents a research prototype to support the analysis approach. Section 5 describes related work, and Section 6 concludes the paper.

2 Approach Overview

The analysis approach is used to determine whether the net effect associated with a behavior specified in a source model can be preserved by distributed behaviors specified in a refactored class model. The net effect preservation property that is checked is defined as follows:

Definition 1: Net Effect Preservation. A sequence of operation invocations, $OpSeq$, in a refactored model is said to preserve the net effect of an operation, $Op0$, in the source model if the set of net effects (i.e., start and end system states associated with an operation invocation) characterized by the specification of $Op0$ is included in the set of net effects (i.e., start and end system states associated with a sequence of operation invocations) characterized by the sequence $OpSeq$. More precisely, a set of operations specified in a refactored model, $\{Op1, Op2, ..., OpN\}$, is said to preserve the net effect of an operation $Op0$ specified in the source model if there exists an invocation sequence of the refactored model operations, $OpSeq = [Op1; Op2; ...; OpN]$, such that the following holds:

1. $OpSeq$ starts in all the states that satisfy the pre-condition of $Op0$.
2. If $OpSeq$ starts in a state that satisfies the pre-condition of $Op0$ then the sequence of operation invocations leaves the system in a state that satisfies the post-condition of $Op0$.

The analysis approach requires a software modeler to provide the following as inputs:

1. The specification of the source model operation, $Op0$, that is refactored.
2. The result of a refactoring (i.e., a refactored class model), and a sequence diagram that describes how $Op0$'s redistributed behavior is used. The sequence diagram provides the sequence of refactored operations that will be analyzed against the source model specification of $Op0$.

The intermediate output of the approach is an analyzable model that can be used to check the net effect preservation property between $Op0$ and $OpSeq$. In this approach, the analyzable model takes the form of an Alloy model that is

produced from (1) the refactored class model, and (2) a sequence diagram that describes $OpSeq$.

The specifications for $Op0$ and the operations involved in $OpSeq$ are also included in the Alloy model. The inclusion of $Op0$ in an Alloy model produced from the refactored class model can be problematic when $Op0$ refers to elements not included in the refactored model. For this reason the first step of the approach checks that the elements referenced in the $Op0$ operation specification also appear in the refactored model.

The second step of the approach generates the base Alloy model that is extended in following steps to check the preservation property. We use a UML-to-Alloy transformation that builds upon our previous work on rigorous analysis of UML class models [17].

The third step of the approach takes as input the specification of $Op0$ and a sequence diagram, and produces an Alloy assertion (or predicate) that is used to determine whether the sequence described in the sequence diagram ($OpSeq$) preserves the net effect of $Op0$. The assertion (or predicate) is added to the Alloy model generated in the second step of the approach. If a check of the assertion (or predicate) by the Alloy Analyzer produces an Alloy instance then the net effect specified by $Op0$ cannot be preserved by the operation sequence.

More details on the major steps of the approach can be found in [18].

3 An Illustrating Example

A maze game class model from [6] (see Figure 1) is used in this paper to illustrate the analysis approach. The $MazeGame$ class is responsible for creating different types of mazes (e.g., $BombedMaze$ and $EnchantedMaze$) and their parts (e.g., $RoomWithBomb$ and $EnchantedRoom$). A maze room consists of four sides that can be doors, walls, or other rooms.

The operation *createBombedMaze()* in class $MazeGame$ is used to create a bombed maze that consists of four walls. Its net effect in the form of OCL specification is given below:

Context *MazeGame::createBombedMaze() : BombedMaze*
// Pre-condition: no maze has been created
Pre: *self.maze→isEmpty()*
// Post-condition: a bombed maze has been created, and it includes a room
// with four walls
Post: *result.oclIsNew() and self.maze.bRooms→size() = 1 and*
self.maze.bRooms→forAll(r : RoomWithBomb | r.bwalls→size() = 4)

If a new type of maze, maze room, door or wall were added, the structure of the class model would need to be changed significantly. Incorporating the *Abstract Factory* pattern [6] into the class model results in a more flexible design in which the maze creation responsibilities are localized in factories that the $MazeGame$ class can access.

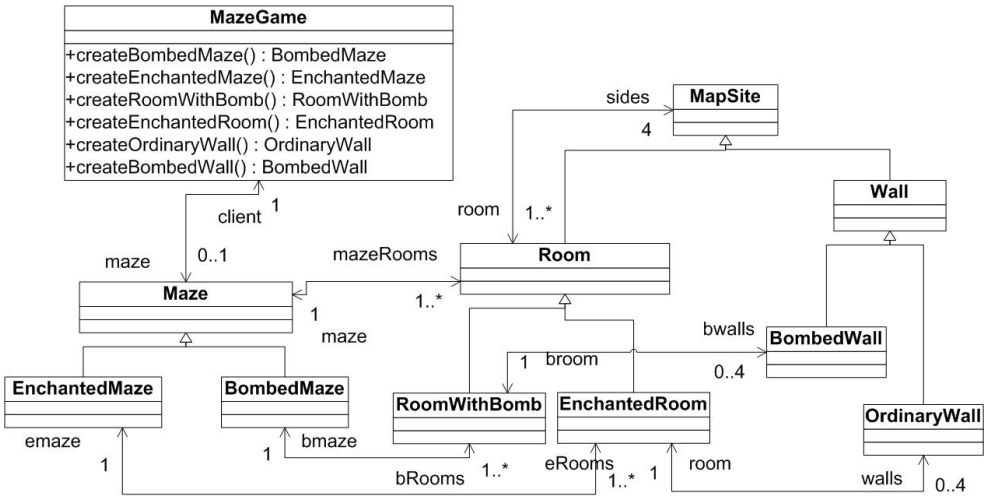

Fig. 1: Maze Game Class Model

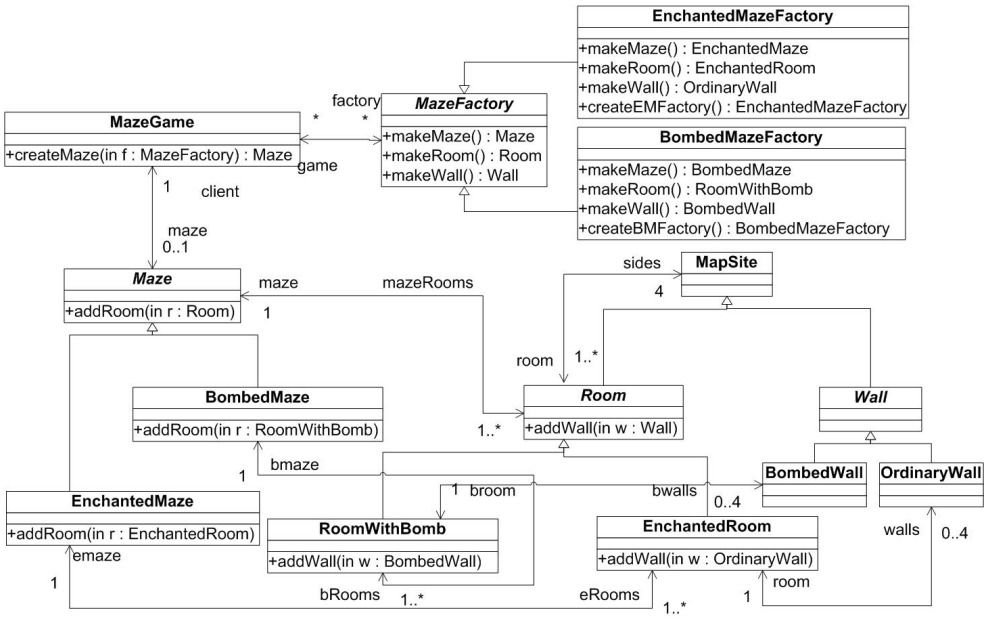

Fig. 2: Refactored Maze Game Class Model

Figure 2 shows a refactored maze game class model that incorporates an instantiation of the *Abstract Factory* pattern. The original $createBombedMaze$ and $createEnchantedMaze$ operations in $MazeGame$ have been replaced by the

$createMaze(f : MazeFactory) : Maze$ operation, that uses a factory to create a specific type of maze. The net effects of the original operations in $MazeGame$ need to be preserved by the behavioral refactoring. The analysis approach described in this paper can be used to check if the net effect of $createBombedMaze$ is preserved by relevant operations in the refactored model.

The OCL specifications for $createMaze$ and $addRoom$ are given below:

Context *MazeGame::createMaze(f:MazeFactory) : Maze*
// Pre-condition: a maze factory has been associated with a maze game
Pre: *self.factory→includes(f)*
Post: *true*

Context *Maze::addRoom(r:Room)*
// Pre-condition: a room has not been associated with a maze
Pre: *self.mazeRooms→excludes(r)*
// Post-condition: a room has been associated with a maze
Post: *self.mazeRooms→includes(r)*

Unlike the $createBombedMaze$ operation, the $createMaze$ operation delegates its responsibility to other operations (i.e., $makeMaze$, $makeRoom$, $addRoom$, $makeWall$, and $addWall$) in the refactored class model. Due to space limitations, only the specifications of $createMaze$ and $addRoom$ are given in the paper (see above). More operation specifications can be found in [18]. A sequence diagram (see Figure 3) is used to describe the result of the behavioral refactoring. It describes an invocation sequence of the refactored model operations that is intended to preserve the net effect of the $createBombedMaze$ operation in the source model.

The analysis showed that if we removed an operation (e.g., $addRoom$) from the operation sequence in Fig. 3, the net effect of $createBombedMaze$ cannot be preserved by the rest of operations in Fig. 3. We also used the same analysis approach to check if the net effects of other source model operations are preserved by refactored model operations. Our analysis results showed that all the operations in the source model (e.g., $createEnchantedMaze$, $createRoomWithBomb$, $createEnchantedRoom$, $createOrdinaryWall$ and $createBombedWall$) can be preserved by relevant operations in the refactored model.

4 Tool Support

We developed a research prototype to investigate the feasibility of developing tool support for the approach. The prototype consists of an Eclipse OCL parser, an Ecore/OCL transformer and an Alloy Analyzer. The Ecore/OCL transformer is developed using Kermeta [13], an aspect-oriented metamodeling tool. The inputs of the prototype are (1) an EMF Ecore [16] file that specifies a refactored class model, (2) a textual OCL file that specifies the pre-/post-conditions of $Op0$ and

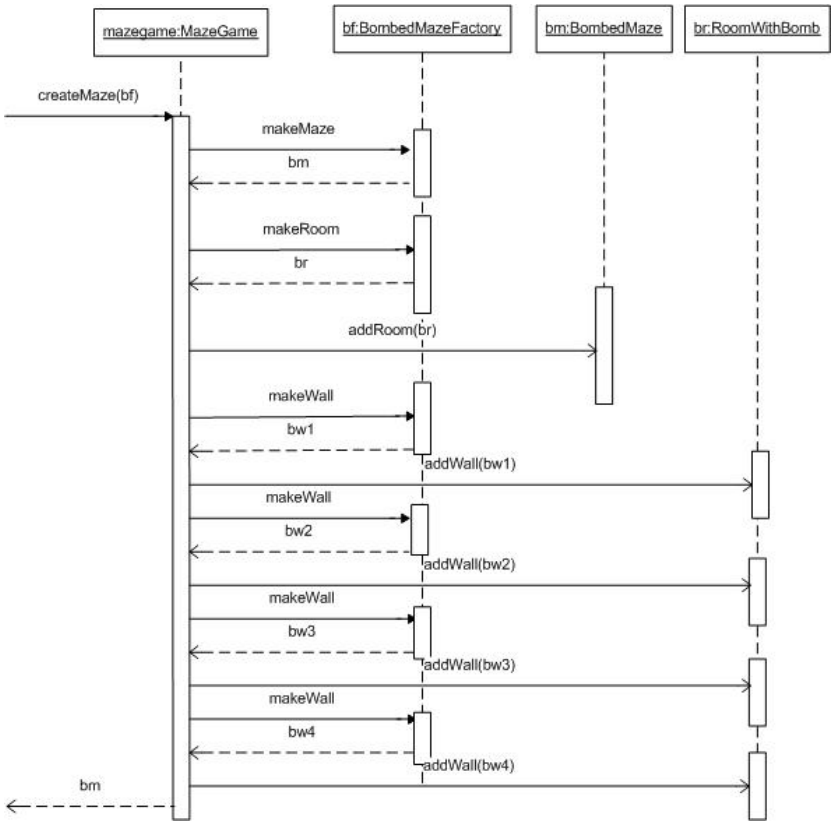

Fig. 3: A Sequence Diagram that Describes an Invocation Sequence of the Refactored Model Operations

operations involved in $OpSeq$, and (3) a textual file that describes a sequence diagram.

The inputs are automatically transformed to an Alloy model consisting of signatures and predicates. The prototype then uses the APIs provided by the Alloy Analyzer to pass the Alloy model to the Alloy SAT solver. The result returned by the Alloy SAT solver is interpreted by the prototype. The interpreted result provides the net effect preservation property between $Op0$ and $OpSeq$.

The prototype implementation uses a visitor pattern to transform a class model with operation specifications into an Alloy model. The traditional visitor design pattern keeps the separation of the structure (i.e., the metamodel elements) and the behavior (i.e., the visitor) by using a specific class for the visitor, and thus results in ping-pong calls between the objects of the structure and the objects of the visitor. The Kermeta [10] language provides an aspect weaving mechanism to simplify the visitor pattern by allowing a user to define a *visit* method for each model element being visited in an aspect class that is

woven into an existing base class at runtime. There is thus no need to keep a visitor class that is used to traverse each model element of a metamodel.

5 Related Work

Two broad categories of related work are discussed in the section: work on model refactoring and work on UML-to-Alloy transformation.

5.1 Model Refactoring

Refactoring has attracted much attention from the MDE community since it was first introduced by Opdyke in his PhD dissertation [14]. Boger et al. [2] applied the idea of refactoring to UML class diagrams, statechart diagrams, and activity diagrams. Their approach, however, does not provide support for rigorously reasoning about a behavioral refactoring.

Both Sunye et al. [19] and Van Gorp et al. [21] used OCL to formally specify the refactoring for UML models. An operation is defined for each type of the refactoring and its OCL pre-/post-condition specifies the model structure that must be satisfied before and after the refactoring associated with the operation. Their approach, however, can only be used to verify the refactoring involving the changes to model structures.

France et al. [5] described a metamodeling approach to pattern-based model refactoring in which refactorings are used to introduce a new design pattern instance to the model. Mens and Tourwe [12] used logic reasoning to detect if a design pattern instance that is introduced to a class model, limits the applicability of certain refactorings.

Straeten et al. [20] proposed a behavior preserving refactoring approach for UML class models. Unlike our approach, the behavior of a class model in their approach is expressed using state machines and sequence diagrams. Gheyi et al. [8] described a rigorous approach to verifying the refactoring for Alloy models.

However, based on our knowledge, none of the above approaches can be used to analyze operation-based model refactoring that involves changes to operation specifications.

5.2 UML to Alloy Transformation

Georg et al. [7] used both Alloy and UML/OCL to specify the runtime configuration management of a distributed system. An ad-hoc comparison between Alloy and UML/OCL is discussed in their paper.

Dennis et al. [3] used the Alloy Analyzer to uncover the errors in a UML model of a radiation therapy machine. The operations in the design model are specified using OCL. An informal description of OCL-to-Alloy transformation is described in their approach. Their approach, however, does not provide support for automated transformation between UML/OCL and Alloy.

Anastasakis et al. [1] described a tool, namely UML2Alloy, that automatically transforms a UML class model with OCL invariants into an Alloy model. Their tool builds upon a formal mapping between UML/OCL metamodel and Alloy metamodel. Unlike their approach, our approach leverages Alloy's trace mechanism to generate an Alloy model with trace features from a UML/OCL model.

Maoz et al. [11] developed a tool that implements the transformation between UML class models and Alloy models. Unlike the approach described in [1], Maoz's tool produces a single Alloy module from two class models. Maoz's approach, however, does not provide support for class models with OCL invariants and operation specifications.

6 Conclusion

We presented an approach to rigorously analyzing a behavioral refactoring that involves making changes to operation specifications expressed in the OCL. The behavioral refactoring analysis involves checking whether relevant operations in the refactored model can preserve the net effects of the operations targeted by the refactoring in the source model. The net effect preservation checking technique described in the paper builds upon the Alloy Analyzer and thus requires a translation from UML class models and OCL operation specifications to Alloy models. We developed a prototype for transforming UML+OCL models to Alloy models with traces to support the net effect preservation check. We applied the approach to a pattern-based model refactoring to demonstrate how software modelers can use the approach to analyze a behavioral refactoring.

We plan to extend the behavioral refactoring analysis approach by providing support for more complex OCL operators. Specifically we are currently investigating how we can use SMT solvers (e.g., Microsoft Z3) at the back-end to analyze the OCL specifications. Our future work will also explore how mappings between equivalent source and refactored forms can be used to support the net effect preservation checking.

ACKNOWLEDGMENT

The work described in this report was supported by the National Science Foundation grant CCF-1018711.

References

1. K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On challenges of model transformation from UML to Alloy. *Software and Systems Modeling*, 9(1):69–86, 2010.
2. M. Boger, T. Sturm, and P. Fragemann. Refactoring browser for UML. *Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 366–377, 2003.

3. G. Dennis, R. Seater, D. Rayside, and D. Jackson. Automating commutativity analysis at the design level. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 165–174. ACM, 2004.
4. M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
5. R. France, S. Chosh, E. Song, and D.K. Kim. A metamodeling approach to pattern-based model refactoring. *Software, IEEE*, 20(5):52–58, 2003.
6. E. Gamma, H. Richard, J. Ralph, and V. John. Design patterns: elements of reusable object-oriented software. *Reading: Addison Wesley Publishing Company*, 1995.
7. G. Georg, J. Bieman, and R. France. Using Alloy and UML/OCL to specify run-time configuration management: a case study. *Practical UML-Based Rigorous Development Methods-Countering or Integrating the eXtremists*, 7:128–141, 2001.
8. R. Gheyi, T. Massoni, and P. Borba. A rigorous approach for proving model refactorings. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 372–375. ACM, 2005.
9. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.
10. J.M. Jézéquel, O. Barais, and F. Fleurey. Model driven language engineering with kermeta. *Generative and Transformational Techniques in Software Engineering III*, pages 201–221, 2011.
11. S. Maoz, J. Ringert, and B. Rumpe. Cddiff: Semantic differencing for class diagrams. *ECOOP 2011–Object-Oriented Programming*, pages 230–254, 2011.
12. T. Mens and T. Tourwe. A declarative evolution framework for object-oriented design patterns. In *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pages 570–579. IEEE, 2001.
13. P.A. Muller, F. Fleurey, and J.M. Jézéquel. Weaving executability into object-oriented meta-languages. *Model Driven Engineering Languages and Systems*, pages 264–278, 2005.
14. W.F. Opdyke. *Refactoring: A program restructuring aid in designing object-oriented application frameworks*. PhD thesis, PhD thesis, University of Illinois at Urbana-Champaign, 1992.
15. O.M.G.A. Specification. Object constraint language, 2007.
16. D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.
17. W. Sun, R. France, and I. Ray. Rigorous analysis of UML access control policy models. In *Policies for Distributed Systems and Networks (POLICY), 2011 IEEE International Symposium on*, pages 9–16. IEEE, 2011.
18. W. Sun, R. France, and I. Ray. *Analyzing Behavioral Refactoring of Class Models*. Technical Report CS-13-104, Colorado State University, http://www.cs.colostate.edu/TechReports/, 2013.
19. G. Sunye, D. Pollet, Y. Le Traon, and J.M. Jezequel. Refactoring UML models. *UML 2001The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 134–148, 2001.
20. R. Van Der Straeten, V. Jonckers, and T. Mens. A formal approach to model refactoring and model refinement. *Software and Systems Modeling*, 6(2):139–162, 2007.
21. P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer. Towards automating source-consistent UML refactorings. *UML 2003-The Unified Modeling Language. Modeling Languages and Applications*, pages 144–158, 2003.