Computer Science
Technical Report

Colorado State University

Analyzing Behavioral Refactoring of Class Models

Wuliang Sun
Colorado State University
sunwl@cs.colostate.edu

Robert B. France
Colorado State University
france@cs.colostate.edu

Indrakshi Ray
Colorado State University
iray@cs.colostate.edu

May 30, 2013

Colorado State University Technical Report CS-13-104

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

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][15] 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][13][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) [16].

The model on which a refactoring is performed is called the *source* model, and the model produced by the refactoring is called the *refactored* or *target* 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 analysis is performed within a bounded scope of class objects.

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 target 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 checking if a sequence of four operations associated with different contexts preserve the net effect of $FlightManager :: bookFlight()$ in the source model.

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 target model preserves the net effect of an operation in the source model in a bounded domain. The net effect of a source model operation is preserved by a sequence of target 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 target operations. The approach takes the sequence of target operations, applies all the states that satisfy the pre-condition of the source model operation in a bounded space, and checks if the sequence of target 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 the sequence of target model operations if a state that does not satisfy the post-condition of the source model operation is produced by the sequence of target model operations when it starts in a state that satisfies the pre-condition of the source 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 target model can preserve the net effect of a changed operation specification in the source model. 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][18] 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 describes the approach and illustrates its use on a small example. Section 4 presents a research prototype to support the analysis

approach. Section 5 discusses limitations of the approach. Section 6 describes related work, and Section 7 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 target 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 target 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 target 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 target class model), and a sequence diagram that describes how $Op0$'s redistributed behavior is used. The sequence diagram provides the sequence of target 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 target 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 target class model can be problematic when $Op0$ refers to elements not included in the target model. For this reason the first step of the approach checks that the elements referenced in $Op0$ operation specification also appear in the target model. In future work we will develop support for checking behavior preservation when elements referenced in $Op0$ are represented differently in the target 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 [18].

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.

3 Approach

In this section we describe (1) the Maze Game class model used to illustrate our approach, (2) the approach used to check that the target class model contains the elements referenced by the source operation that has been refactored, (3) the UML-to-Alloy transformation, and (4) the net effect preservation check.

3.1 Maze Game Class Model

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() : BombedMaze* 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.

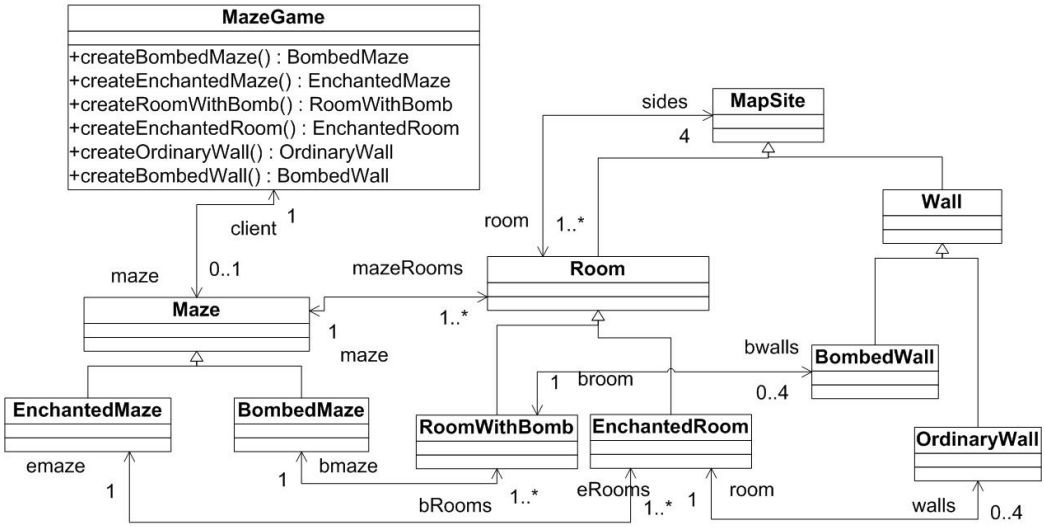

Fig. 1: Maze Game Class Model

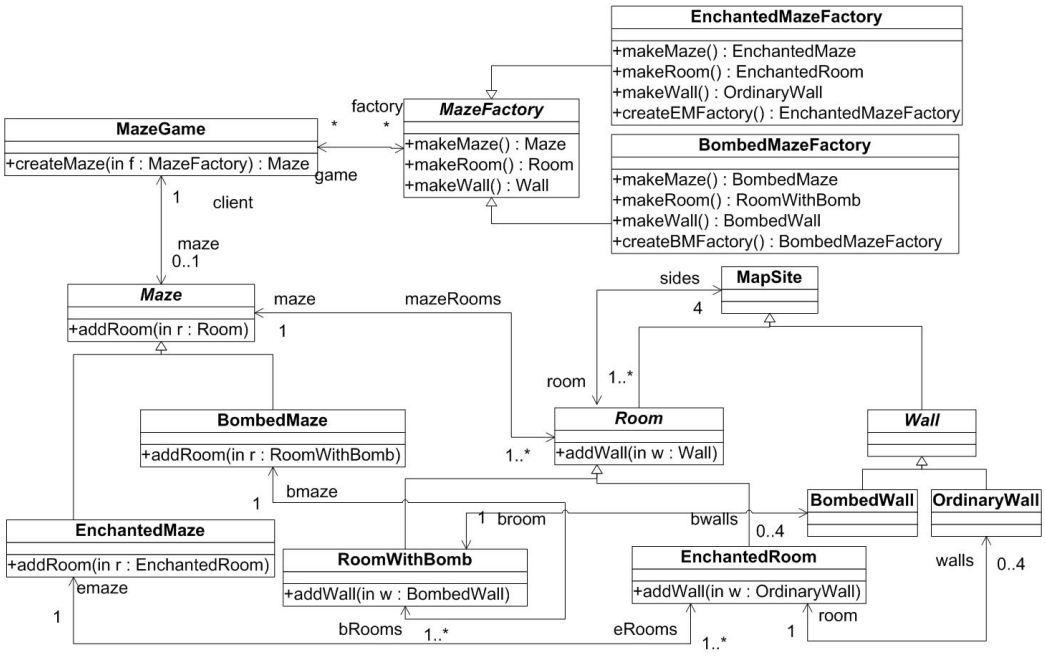

Fig. 2: Refactored Maze Game Class Model

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 target model.

The OCL specifications for $createMaze$, $makeRoom$, $addRoom$ and $addWall$ 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 *MazeFactory::makeRoom() : Room*
Pre: *true*
// Post-condition: a room has been created
Post: *result.oclIsNew()*

Context *EnchantedMazeFactory::makeRoom() : EnchantedRoom*
Pre: *true*
// Post-condition: an enchanted room has been created
Post: *result.oclIsNew()*

Context *BombedMazeFactory::makeRoom() : RoomWithBomb*
Pre: *true*
// Post-condition: a room with bomb has been created
Post: *result.oclIsNew()*

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)*

Context *BombedMaze::addRoom(r:RoomWithBomb)*
// Pre-condition: a room has not been associated with a bombed maze
Pre: *self.bRooms→excludes(r)*

// Post-condition: a room has been associated with a bombed maze
Post: *self.bRooms→includes(r)*

Context *EnchantedMaze::addRoom(r:EnchantedRoom)*
// Pre-condition: a room has not been associated with an enchanted maze
Pre: *self.eRooms→excludes(r)*
// Post-condition: a room has been associated with an enchanted maze
Post: *self.eRooms→includes(r)*

Context *RoomWithBomb::addWall(w:BombedWall)*
// Pre-condition: a wall has not been associated with a room with bomb
Pre: *self.bwalls→excludes(w)*
// Post-condition: a wall has been associated with a room with bomb
Post: *self.bwalls→includes(w)*

Context *EnchantedRoom::addWall(w:OrdinaryWall)*
// Pre-condition: a wall has not been associated with an enchanted room
Pre: *self.walls→excludes(w)*
// Post-condition: a wall has been associated with an enchanted room
Post: *self.walls→includes(w)*

Unlike the $createBombedMaze$ operation, the $createMaze$ operation delegates its responsibility to other operations (i.e., $makeMaze$, $makeRoom$, $addRoom$, $makeWall$, and $addWall$) in the target class model. A sequence diagram (see Figure 3) is used to describe the result of the behavioral refactoring. It describes an invocation sequence of the target model operations that is intended to preserve the net effect of the $createBombedMaze$ operation in the source model.

The major steps of the approach are described in the following subsections.

3.2 Step 1: Analyzing a Target Class Model

The specification of an operation is said to be invalid in the context of a class model if there exists an element referenced by the operation specification that is not included by the class model. Since the analysis approach described in the paper uses the target class model to produce the Alloy model, it must be checked to determine if all elements referenced in the source operation exist in the target model.

If the specification of $Op0$ refers to elements not included in the the target class model, one may choose to adapt the specification of $Op0$ to the target class model, and use the adapted specification to check the net effect preservation property between $Op0$ and $OpSeq$. If this is done, then one is obligated to show that the adaptation is equivalent to the original operation specification of

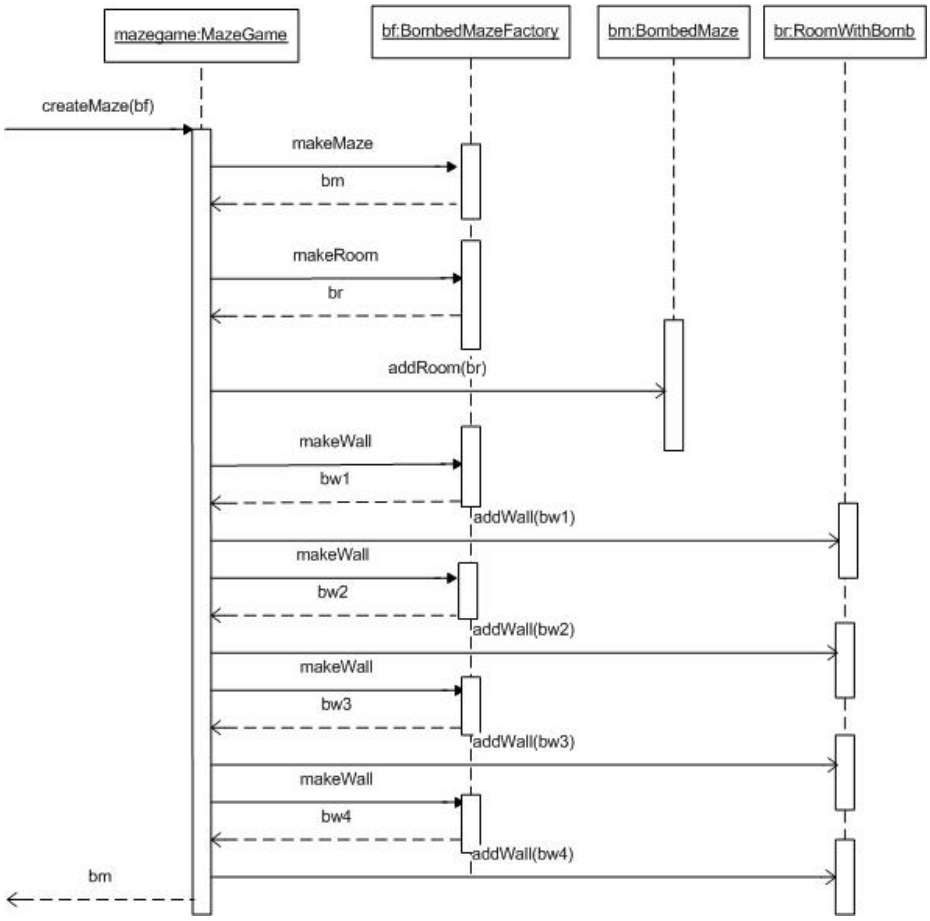

Fig. 3: A Sequence Diagram that Describes an Invocation Sequence of the Target Model Operations

the source operation. In this paper we do not adapt the specification of $Op0$ to the target class model. More details on OCL specification refactorings and adaptations can be found in [12].

The first step of the approach uses a simple slicing technique to extract the model elements referenced by the specification of $Op0$ from the source class model. The slicing technique takes as input the OCL specification of $Op0$, traverses an OCL syntax tree generated from the input specification, and collects the model elements (i.e., classes and attributes) accessed by the specification through navigation. If the target class model does not include all the model elements referenced by $Op0$ then the net effect preservation check is not performed.

In the maze game example, the model elements referenced by $createBombedMaze$ are $MazeGame$, $Maze$, $Room$, $Wall$, $BombedMaze$, $RoomWithBomb$,

and $BombedWall$, and the target class model (see Fig. 2) includes all the model elements needed by $createBombedMaze$. Therefore the target class model in Fig. 2 can be used to check the net effect preservation property between $Op0$ and $OpSeq$.

3.3 Step 2: Generating an Alloy Model from a Refactored Class Model

The second step of the approach described in the paper involves transforming a refactored UML design class model to an Alloy model that specifies behavioral traces. The UML-to-Alloy model transformation uses an intermediate model that provides a static description of behavior in terms of sequences of state transitions, where a transition represents an invocation of an operation described in the class diagram. The *snapshot transition model* (STM) developed by Yu et al. [24][23][22] is the intermediate model used in the UML-to-Alloy transformation.

In the remainder of this section we describe (1) the *snapshot transition model* (STM), (2) the class-model-to-STM transformation, and (3) the STM-to-Alloy model transformation.

Snapshot Transition Model Software behavior can be represented as a sequence of state transitions, where each transition is triggered by an operation invocation. Yu et al. [24][23][22] describe how a design class model with operation specifications can be transformed to a static model of behavior, called a *snapshot transition model* (STM). A *snapshot* represents a system object configuration at a particular time (i.e., a system state). A *snapshot transition* describes the behavior of an operation in terms of how system state changes after the invoked operation has completed its task. It consists of a $before$ state, an $after$ state, and the operation invocation that triggers the transition. An operation invocation is described by the operation name and the parameter values used in the invocation.

Generating a Snapshot Transition Model from a Refactored Class Model Figure 4 shows a partial snapshot transition model generated from the refactored maze game class model in Fig. 2. The instances of class $Snapshot$, are snapshots, and the instances of class $Transition$ are transitions that each relates a *before* snapshot with an *after* snapshot. A snapshot consists of linked instances of classes in a class model (i.e., an object configuration).

Each operation in the original design class model is transformed into a specialization of class $Transition$. The parameters (including the return type) of each operation in the original design class model are transformed into references (shown as attributes) in the $Transition$ specialization. Moreover, if a parameter has a class type, it is transformed into two references, one of which specifies the parameter's state before the execution of the operation and the other specifies the parameter's state after the execution of the operation. Also, two references pointing to before and after states of the object on which the operation is called

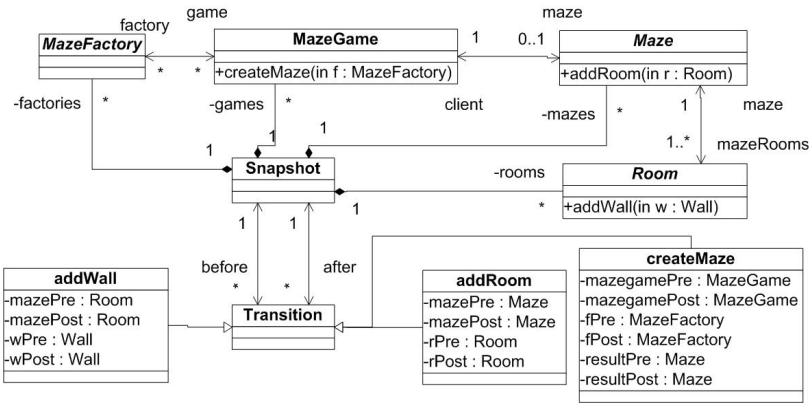

Fig. 4: An Example of a Snapshot Transition Model that is Generated from the Refactored Class Model in Fig. 2 (We omitted most of classes and operations in Fig. 2 for simplicity)

are generated and placed in the specialized $Transition$ class representing the operation. Operation specifications in the design class model are transformed into transition invariants that specify the before and after snapshots that are associated with $Transition$ instances.

For example, the specification for the $createMaze$ operation is transformed to the following transition invariant on the $Transition$ specialization, $createMaze$:

Context *createMaze*
inv*:*
before.games→includes(mazegamePre) and ...
after.games→includes(mazegamePost) and ...
// Generated from the pre-condition of the createMaze operation
mazegamePre.factory→includes(fPre) and
// Frame conditions
before.games - mazegamePre = after.games - mazegamePost
...

More details on the class model-to-*snapshot transition model* transformation can be found in [22][23].

Generating an Alloy Model from a Snapshot Transition Model Alloy [9] is a textual modeling language based on first-order relational logic. An Alloy model consists of *signature* declarations, $fields$, $facts$ and *predicates*. Each $field$ belongs to a *signature* and represents a relation between two or more *signatures*. $Facts$ are statements that define constraints on the elements of the model. *Predicates* are parameterized constraints that can be invoked from within $facts$ or other *predicates*.

```
module MazeGame
open util/ordering[Snapshot] as SnapshotSequence
sig MazeGame{} sig Maze{} sig MazeFactory{} …
abstract sig ID{}
one sig ID_createMaze, ID_addRoom, ⋯ ID_Null extends ID{}

sig Snapshot{
OperID: one ID,
// Objects
games: set MazeGame, mazes: set Maze, factories: set MazeFactory…
// Links
gamefactory: MazeGame set->set MazeFactory,
clientmaze: MazeGame one->lone Maze,
...
}

pred createMaze[disj before, after: Snapshot, mazegamePre,
mazegamePost: MazeGame, fPre, fPost: MazeFactory, resultPre,
resultPost: Maze]{
after.OperID = ID_createMaze
mazegamePre in before.games ...
mazegamePost in after.games ⋯
// Pre-condition
fPre in mazegamePre.(before.gamefactory)
// Frame conditions
after.games - mazegamePost = before.games – mazegamePre
…
}
// More predicates
…
```

Fig. 5: Partial Alloy Model Transformed from the Snapshot Transition Model in Fig. 4

Figure 5 shows a partial Alloy model generated from the *snapshot transition model* in Fig. 4. The figure identifies the parts that are generated by the three-step transformation algorithm described in [18]. First, each part of the *Snapshot* class in the snapshot transition model is transformed to a signature in Alloy. If a class has attributes, its attributes are transformed to fields of the signature corresponding to the class. The *Snapshot* signature is declared as an *ordering* type (e.g., open util/ordering[*Snapshot*] as *SnapshotSequence*) representing sequences of *snapshots*.

Second, the *Snapshot* class is transformed to a *Snapshot* signature containing fields that specify the object configuration within a snapshot. Two groups of fields in the *Snapshot* signature are used to specify object configurations: fields defining a set of objects (e.g., $games : setMazeGame$) and fields defining links between objects (e.g., $gamefactory : MazeGame\ set \rightarrow set\ MazeFactory$).

The Snapshot signature also includes a field, $OperID$, that is used to identify the operation that causes a transition to the snapshot when the snapshot is part of a sequence of transitions. There is an identifier type for each operation in the original class model (e.g., $ID\_createMaze$ is the identifier that corresponds to the operation $createMaze$).

Third, each $Transition$ specialization in the snapshot transition model is transformed to a predicate in Alloy that defines a relationship between before and after states. If a $Transition$ specialization has attributes, its attributes are transformed to parameters of the predicate. Two more parameters, $before$ and $after$ with the type $Snapshot$, are added to each predicate to represent the system states before and after the transition. An equality that identifies the operation causing the transition (e.g., $after.OperID = ID\_createMaze$) is also included in each predicate.

OCL invariants associated with each $Transition$ specialization in the snapshot transition model are transformed into the body of the predicate corresponding to the $Transition$ specialization. Objects and links that are not changed during the transition (i.e., frame conditions) are explicitly specified in the predicate.

More details on the *snapshot transition model*-to-Alloy model transformation can be found in [18].

3.4 Step 3: Checking Net Effect Preservation Property

When an operation invocation sequence (i.e., $OpSeq$) only consists of one operation (e.g., $OpSeq = [Op1]$), a software modeler can use the Alloy Analyzer to check if the pre-condition of $Op0$ implies the pre-condition of $Op1$, and if the post-condition of $Op1$ implies the post-condition of $Op0$. If both implications hold, $OpSeq$ can preserve the net effect of $Op0$.

However, when $OpSeq$ consists of more than one operation, $OpSeq = [Op1; ...; OpN]$, the situation is more complex. The pre- and post-conditions of a sequence of operations are not simply the pre-condition of the first operation and the post-condition of the last operation in the sequence. The pre-condition of a sequence of operations specifies only those start states that satisfy the pre-condition of the first operation and that produce intermediate states that satisfy the pre-condition of the next operation to execute in the sequence, for each operation in the sequence. That is, the pre-condition of a sequence of operations implies the pre-condition of the first operation. Similarly the post-condition of a sequence of operations implies the post-condition of the last operation of the sequence.

Thus, a software modeler cannot determine the net effect preservation property between $OpSeq$ and $Op0$ by simply checking if the pre-condition of $Op0$ implies the pre-condition of $Op1$ since the pre-condition of $Op1$ does not ensure that the pre-conditions of all operations in the sequence will be satisfied when the operations are invoked. Similarly, one cannot use the post-condition of OpN to check if $OpSeq$ can preserve the net effect of $Op0$ since the post-condition of

OpN does not ensure that the post-conditions of all operations in the sequence will be satisfied after the operations have been invoked.

In addition, it is not a simple task to infer the pre-/post-condition of *OpSeq* from the pre-/post-conditions of the operations in *OpSeq* since the pre-/post-condition of *OpSeq* may be any combination of the pre-/post-conditions of the operations in *OpSeq*. Thus, rather than using the pre-/post-conditions to determine the net effect preservation property between *Op*0 and *OpSeq*, a software modeler can use the Alloy Analyzer to check if *OpSeq* starts in a state that satisfies the pre-condition of *Op*0, and ends in a state that does not satisfy the post-condition of *Op*0 (i.e., a state satisfies the negation of the post-condition of *Op*0). If the Analyzer does not return an instance, *OpSeq* can preserve the net effect of *Op*0 in the bounded scope. In the following we describe the two cases in more details.

***OpSeq* consists of only one operation** When $OpSeq = Op1$, *OpSeq* can preserve the net effect of *Op*0 only if *Pre_Op*0 (i.e., the pre-condition of *Op*0) implies *Pre_Op*1 (i.e., the pre-condition of *Op*1), and *Post_Op*1 (i.e., the post-condition of *Op*1) implies *Post_Op*0 (i.e., the post-condition of *Op*0). Since the Alloy Analyzer is an instance finder, it can be used to search a counterexample for an assertion that specifies *Pre_Op*0 *implies Pre_Op*1 *and Post_Op*1 *implies Post_Op*0. If the Analyzer finds such counterexamples, *Op*1 cannot preserve the net effect of *Op*0.

Suppose that we want to check that the net effect of *createBombedMaze* is preserved by only one operation, *addRoom*. An example of the Alloy assertion for checking if *addRoom* preserves the net effect of *createBombedMaze* is given below:

assert *Assertion*{
all s: Snapshot | all mg : s.mazegames | all maze : s.mazes | all r : s.rooms|
Pre_createBombedMaze[s, mg]
implies Pre_addRoom[s, maze, r] and
...
},

where *Pre_createBombedMaze*, shown below,

pred *Pre_createBombedMaze[s: Snapshot,*
mg : MazeGame]{
mg.(s.clientmaze) = none
},

is a predicate generated from the pre-condition of *createBombedMaze*, and *Pre_addRoom*, shown below,

pred *Pre_addRoom[s : Snapshot, maze : Maze,*

r : Room]{
r not in maze.(s.mazemazerooms)
},

is a predicate generated from the pre-condition of *addRoom*.

The Alloy Analyzer checks whether there exists an instance violating the assertion, that is, $Pre\_createBombedMaze$ *implies* $Pre\_addRoom$ *and* $Post\_addRoom$ *implies* $Post\_createBombedMaze$. For this example, the Analyzer did find a counterexample for the assertion, indicating that the net effect of $createBombedMaze$ cannot be preserved by *addRoom*.

***OpSeq* consists of more than one operation** An Alloy predicate is generated from the specification of $Op0$ and a sequence diagram that describes $OpSeq$. The predicate specifies that the operations must adhere to the invocation order described in the sequence, start in states satisfying the pre-condition of $Op0$, and end in states satisfying the negation of the post-condition of $Op0$. The Alloy Analyzer uses the predicate to determine if the net effect of $Op0$ is preserved by $OpSeq$.

An example of the Alloy predicate generated from the operation sequence shown in Fig. 3 is given below:

pred *Predicate{*
let first = SnapshotSequence/first|
let second = SnapshotSequence/next[first]|...
some mg:first.mazegames|
some bf:first.bombedmazefactories|
some bm:BombedMaze|...
// Start states specified using the pre-condition of createBombedMaze
mg.(first.clientmaze) = none
// An operation invocation sequence
createMaze[first, second, mg, mg, bf, bf] and
makeMaze[second, third, bf, bf, bm, bm] and
...
// End states specified using the post-condition negation of createBombedMaze
not (
bm in last.mazes
#mg.(last.clientmaze).(last.bmazebRooms) = 1
all r : mg.(last.clientmaze).(last.bmazebRooms)|
#r.(last.broombwalls) = 4)
}

Note that *SnapshotSequence* represents a sequence of snapshots, and $SnapshotSequence/first$ returns the first snapshot of the snapshot sequence. The $next[s : Snapshot]$ function returns the input snapshot's next snapshot. Thus,

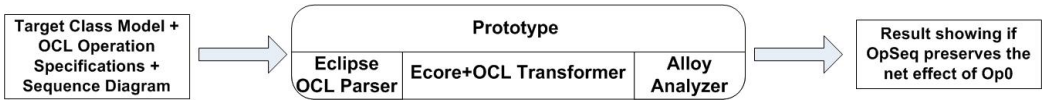

Fig. 6: Prototype Overview

$SnapshotSequence/next[first]$ returns the second snapshot of the snapshot sequence. The clause *result.oclIsNew()* in the post-condition of $createMaze$ is semantically equivalent to the Alloy specifcation *bm in last.mazes*. The Alloy Analyzer uses the above predicate to check whether there exists an instance representing the given sequence that starts in a state satisfying the precondition of $Op0$ and ends in a state satisfying the post-condition negation of $Op0$. For this example, the Analyzer did not find such instances, indicating that the operation sequence described in Fig. 3 can preserve the net effect of the $createBombedMaze$ operation in the bounded scope. The analysis also 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 target 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 target model.

4 Tool Support

We developed a research prototype to investigate the feasibility of developing tool support for the approach. Figure 6 shows an overview of the prototype. It consists of an Eclipse OCL parser, an Ecore/OCL transformer and an Alloy Analyzer. The Ecore/OCL transformer is developed using Kermeta [14], an aspect-oriented metamodeling tool. The inputs of the prototype are (1) an EMF Ecore [17] file that specifies a target class model, (2) a textual OCL file that specifies the pre-/post-conditions of $Op0$ and 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

```
aspect class IteratorExpCS{
        ...
        // Translate an OCL expression that includes iterators into an Alloy specification
        // OCL expression format:  <source> <iterator> ( [<variable1>[<variable2>]] | <body> )
        // E.g., self.nodes->forAll(n|n.edges.size() > 2)
        // Variable temp: a string buffer used to save the translation result
        // Variable iterators: a hash table that maps an OCL iterator to an Alloy keyword
        // E.g.,  forAll === >  all, exists === > some, ...
        method translate(temp : String, iterators : Hashtable<String, String>) : Void is do
                var iteratorName : String init self.simpleNameCS.~value.toString
                temp.append(iterators.getValue(iteratorName))

                if self.variable1 != void then
                        temp.append(self.variable1.name)
                end
                if self.variable2 != void then
                        temp.append(" , ")
                        temp.append(self.variable2.name)
                end

                temp.append(" : ")
                self.source.translate(temp, iterators)
                temp.append(" | ")
                self.body.translate(temp, iterators)
        end
        ...
}
```

Fig. 7: Excerpt of the OCL2Alloy Translation implemented using Kermeta

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.

Figure 7 shows an excerpt of the OCL2Alloy translation implemented using Kermeta. Class $IteratorExpCS$ is an element of OCL metamodel. An instance of $IteratorExpCS$ is an OCL expression that includes iterators such as $forAll$, $exists$, and etc. The *translate* method is used to transform an OCL iterator expression into an Alloy specification. The method takes as input a string buffer that is used to save the translation result and a hash table that maps an OCL iterator to an Alloy keyword. The method first uses the name of an OCL operator (i.e., $iteratorName$) to find a corresponding Alloy keyword, and then visits the variables, source and body of the iterator.

5 Discussion

In this section we discuss limitations of our work, and its scope of practice. Specifically we discuss the scope of the OCL specifications supported in the behavioral refactoring analysis approach.

We have developed a prototype to support the behavioral refactoring analysis approach described in the paper. The prototype uses an Alloy Analyzer at the back end to analyze the net effect preservation property between $Op0$ and $OpSeq$, and thus requires a translation from OCL to Alloy. Therefore the scope of the OCL specifications supported by the approach is determined by the OCL2Alloy translation implemented in the prototype.

Most of the OCL operators have corresponding Alloy constructs. For example, OCL operator $forAll$ corresponds to Alloy construct all, $exists$ corresponds to $some$, $includes$ corresponds to in, $excludes$ corresponds to $!in$, sum corresponds to sum, and $closure$ corresponds to $*$. OCL expressions that involves such operators can be directly transformed into Alloy specifications.

However, as pointed out by Anastasakis et al. [1], the translation from OCL to Alloy is not seamless. There are some OCL operators that do not have corresponding Alloy constructs, and thus OCL expressions including such operators cannot be easily transformed into Alloy specifications. Some of them can be partially supported by the prototype according to the Alloy libraries used for that. For instance, OCL operators like $select$ and $collect$ are translated by the prototype using Alloy functions that implement their semantics. Imperative flavor operator $iterate$ is partially supported by the transformation tool. The prototype provides support for OCL specifications including $iterate$ expressions that can be rewritten as $forall$ with $select/collect$ operators. However, the prototype cannot deal with $iterate$ expressions that involve arithmetic accumulation since Alloy is a purely declarative language that does not provide support for imperative accumulators. Finally, the translation cannot deal with OCL casting operators like $oclAsType$ since Alloy has a very simple type system that does not support type casting.

The approach is also limited in that it currently does not support refactoring in which the elements referenced by the source operation do not exist in the same form in the target model. There are refactorings in which elements referenced by a source operation have different, but equivalent forms in the target model. Our future work will explore how mappings between equivalent source and target forms can be used to support preservation checking in these situations.

6 Related Work

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

6.1 Model Refactoring

Refactoring has attracted much attention from the MDE community since it was first introduced by Opdyke in his PhD dissertation [15]. 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 [13] 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 verify operation-based model refactoring that involves changes to operation specifications.

6.2 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.

7 Conclusion

We presented an approach to formally 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 target 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. Slaviša Marković and Thomas Baar. Refactoring ocl annotated uml class diagrams. *Model Driven Engineering Languages and Systems*, pages 280–294, 2005.
13. 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.
14. 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.
15. 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.
16. O.M.G.A. Specification. Object constraint language, 2007.
17. D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.
18. 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.
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.
22. L. Yu, R. France, and I. Ray. Scenario-Based Static Analysis of UML Class Models. *Model Driven Engineering Languages and Systems*, pages 234–248.
23. L. Yu, R. France, I. Ray, and S. Ghosh. A Rigorous Approach to Uncovering Security Policy Violations in UML Designs. In *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*, pages 126–135. IEEE, 2009.
24. L. Yu, RB France, I. Ray, and K. Lano. A light-weight static approach to analyzing UML behavioral properties. In *12th IEEE International Conference on Engineering Complex Computer Systems, 2007*, pages 56–63, 2007.