Mobile Netw Appl
DOI 10.1007/s11036-017-0807-z

A Formal Approach to Checking Consistency in Software Refactoring

Hong Anh Le[1] · Thi-Huong Dao[2] · Ninh-Thuan Truong[2]

Abstract In software development, refactoring is a process that improves the system internal structure without altering its external behavior. Applying design patterns, which are common reusable solutions of several kinds of problems is widely adopted. This technique, however, raises a challenging issue that after applying design patterns the software system may not preserve some certain behavioral properties. This paper proposes a new approach to checking consistency between original software system and its evolution at both design and implementation phases. First, we formalize elements of software designs and programs. Methods, based on these formalizations, are proposed for verifying the design and implementation of the system. Finally, the paper presents a case study of Adaptive Road Traffic Control system to illustrate the proposed approach in detail.

Keywords Refactoring · Design patterns · Consistency · Formal approach

✉ Hong Anh Le
lehonganh@humg.edu.vn

Thi-Huong Dao
huongdt.di12@vnu.edu.vn

Ninh-Thuan Truong
thuantn@vnu.edu.vn

[1] Hanoi University of Mining and Geology, Hanoi, Vietnam

[2] VNU, University of Engineering and Technology, Hanoi, Vietnam

1 Introduction

In software engineering, software evolution is the process of developing software initially, then repeatedly updating it due to many reasons such as reducing errors, saving efforts in development or improving software quality. Techniques, which are commonly used in this process are re-engineering and refactoring of software code or models.

Refactoring is a powerful technique, which is used to improve the quality of the software (e.g., extensibility, modularity, re-usability, complexity, maintainability...) by changing the internal structure of software without altering its external behavioral properties. Refactoring using patterns is the process of improving the design of existing code, the classic solutions to solve the design problems.

A design pattern [13] is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a completed design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Patterns summarize best practices that programmers can use to solve common problems when designing an application or system. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply object-orientation or more generally mutable state, are not as applicable in functional programming languages. A software system should be optimized by refatoring methods, in which design patterns may be used to improve its source code or design model. Patterns are language independent, they have been broadly used in many programming languages including Java, C++, etc..

Published online: 26 January 2017

These techniques, however, need to be performed carefully. The main danger is that errors can inadvertently be introduced, especially when refactoring is done manually. It means that some certain significant properties are not preserved in the new design and source code. Several approaches have been proposed to checking the consistency between systems using graph transformation techniques [5, 25], description logic frameworks [23], and XML metadata interchange [11]. This paper introduces a new approach to checking the consistency of systems before and after refactoring in two phases: design and implementation. The main contributions of the paper are (1) formalizing of design models and program's constructs; (2) presenting methods for verifying the consistency of the design and the implementation of the software; and (3) utilizing OCL [8] and JML [21] to check the consistency of software design and Java programs of the ARTC system respectively.

The rest of the article is organized as follows. Section 2 presents some works related to our research. Section 3 gives an overview of Strategy pattern, OCL, and JML. A motivating example of ARTC system is introduced in Section 4. Section 5 portrays the framework overview of checking consistency software refactored systems. The basic formalization concepts for checking consistency is introduced in Section 6. Section 7 describes the checking consistency process at both design and implementation phases. Illustrating the proposed approach with the motivating example is depicted in Section 8. Finally, Section 9 concludes and gives some directions for future works.

2 Related work

Mens and Tourwe [19] made a great survey of software refactoring where they mentioned various formal techniques, used for refactoring process such as graph transformation, invariants, dynamic program analysis, and program slicing, etc. In this section, we review some kinds of literature that work on checking the consistency of software refactoring and UML models in more detail.

One research direction in this area is using graph transformation where models or software artifacts can be considered as graphs and refactoring is referred to transforming rules. Bottoni et al. [6] proposed to maintain consistency between the code and design models which are composed of different types of UML models by describing refactoring as a set of distributed graph transformations.

Mens et al. [20] introduced a graph representation and graph formal rewriting rules to specify a program and its refactoring. Their work could prove the preservation properties after refactoring at the source code level while this work focuses on verifying the consistency of properties at the design phase and allows one to check whether the behavior of a scenario is consistent or not.

Zhao et al. [25] presented a graph transformation based approach to design pattern evolution. They proposed a graph grammar based syntax parser to check the structural integrity of the evolved design patterns. The authors defined a rule for each kind of design patterns, then check if evolved models kept consistency properties of applying patterns.

Van Eetvelde and Janssens [12] transformed a program into a graph by using a set of graph production rules. It allowed them to view and manipulate the program and its refactoring. Their paper, however, did not pay attention to the consistency among different phases of a software system.

Researchers also proposed to use description logic or XML metadata to transform between UML models and its evolution. Van Der Straeten et al. [23] make use of the technique to formalize both models, after that, the consistency between them was verified. In the experiment, they translated UML metadata into Loom (an extensive query language) and associated production rule system with description logic tool. They classified the consistency into several categories, then focused on instance definition missing and incompatible behavior caused by referencing. Taking these inconsistencies into account, in our opinion, are insufficient because when applying design patterns, software developers often expect that all external behaviors do not change, i.e., scenarios' properties are preserved. And this is the objective of our research.

Jing Dong et al. [11] recommended a model transformation approach to check the consistencies between design models. They depicted both the origin and evolved UML models by XML metadata interchange (XMI) format which aimed to facilitate the transformations. Afterward, Java Theorem Prover was done to check consistencies between them. In the research, they need one more step to convert XMI into RDF or RDFS before checking system properties, so this step makes the approach more complex to implement. Meanwhile, we use OCL, a common object constraint language embedded in UML models, to check the consistency. Therefore, we do not need to have an intermediate transformation step.

Li et al. [14] analyzed and presented methods for checking five types of consistency properties of UML requirements consisting a use-case model and a conceptual class model. Rasch and Wehrheim [22] proposed to use Object-Z to check the consistency between classes and associated state machines. These approaches, however, have focused on checking consistency between different diagrams of a model but have not considered the consistency in refactoring.

There are various techniques can be used in the perspective of checking programs in refactoring. Program slicing is one of the techniques can be used for refactoring. Takeshi et al. [18] combined program slicer and symbolic simulation to check behavior consistency of C program source code at different refinement levels.

Opdyke [21], who gave the original definition of behavior preservation, were suggested that for the same set of input values, the resulting set of output values should be the same before and after the refactoring. Their research proposed to ensure the behavior preservation by specifying refactoring preconditions. But, in this research, besides the preconditions concept, the post-conditions is also complement.

Ward and Bennett [24] introduced a formal language, named WSL, and its supporting tools provided program structuring that proves behavior preservation of the program. Their approach deals with the consistency at the implementation level while our approach can check the consistency at the design level.

In comparison to the prior work, our approach focuses on consistency properties between each scenario of the model in the refactoring process with design patterns. Based on pre/post-conditions of a scenario, which are calculated from pre/post-conditions of scenario operations before and after, the consistency properties can be verified. Since a scenario represents an external behavior of the system, software engineers then can assure that using design patterns in some scenarios are safe. With frequent usages of design patterns in software engineering, our approach is practical and feasible to implement.

3 Background

In this section, we briefly introduce about the basic knowledge of techniques used in the proposed approach including *Strategy* design pattern, OCL, and JML.

3.1 Strategy pattern

Software design patterns, is originated by Christopher Alexander [3], is a general reusable solution to a commonly occurring problem within a given context in software design. It has became popular since 1994 by GOF [13], in which they have categorized the design patterns into three groups, namely creational, structural, and behavioral patterns.

In this section, we present in detail one behavioural pattern, namely Strategy pattern, because we use this pattern in our case study in Section 4. Strategy pattern encapsulates, defines a family of algorithms and makes them interchangeable independently from clients. It defines objects, which represent various strategies and a context object varying as per its strategy object. The strategy object changes the executing algorithm of the context object.

The **Strategy pattern** comprises three participants as portrayed in Fig. 1:

- **IStrategy**: The interface that is shared among the concrete strategy classes in the family. Class *Context* uses this interface to call the algorithm defined by a concrete strategy.
- **ConcreteStrategy**: Where the real implementation of strategy takes place.
- **Context**: The class maintains a reference of type *IStrategy*. In some cases, *Context* may implement operations so that *ConcreteStrategy* can access its data.

The advantage of using strategy pattern is that encapsulating algorithms in individual classes will render reusing code much more convenient and hence, the behavior of the **Context** can be altered at run-time dynamically.

3.2 OCL

OCL [8] is a formal language adopted as a standard by the OMG [1]. The latest version of OCL v2.4 was released in 2014 which is used to define different kinds of expressions complementing UML models as follows.

- Invariants: stating conditions that must be satisfied in every instantiation of the model.
- Initialization: initializing class properties.
- Derivation rules: computing the value of derived model elements.
- Operation contracts: consisting a set of pre- and post-conditions. A precondition defines a set of conditions on the operation input and the system state that must hold when the operation is performed. A postcondition

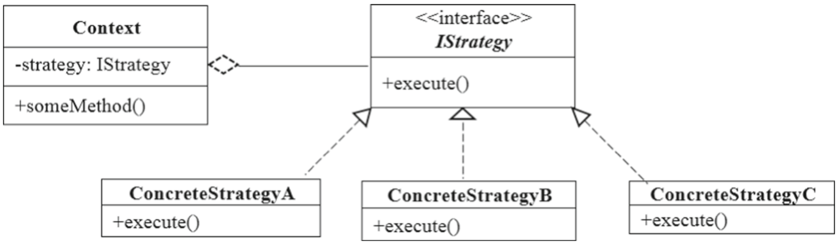


Fig. 1 Strategy pattern

defines the set of conditions that the system state must hold at the end of the operation.

3.3 Java modeling language

Java Modeling Language (JML) [15] is a *behavioral interface specification language* (BISL) that can be used to specify Java classes and interfaces. JML specifications or assertions can be added directly to source code as a special kind of comments called annotation comments, or they can live in separate specification files. These assertions are usually written in a form that can be compiled, so that their violations can be detected at run-time.

One should employ JML due to some following reasons [15]:

- the precise, unambiguous description of the behavior of Java program modules (i.e., classes and interfaces), and documentation of Java code;
- the possibility of tool support [7].

JML's syntax is very close to the Java programming language, so it is to use by programmers who have familiar with Java. Th details of JML can be found in [16].

4 A motivating example: adaptive road traffic control system

4.1 ARTC system's description

In order to illustrate the proposed approach, we extract scenarios from the ARTC system [2]. Traffic congestion is an ever increasing problem in towns and cities all over the world. Local authorities must continually work to maximize the efficiency of their road networks and to minimize any disruptions caused by accidents and events.

From the object-oriented perspective, the initial ARTC system is described by a simplistic model with four classes, namely *Detector, TrafficController, Road* and *Optimizer*. The *Traffic controller* is an essential role which coordinates the other classes. The *Detector* describes the physical location of the detectors as well as counting of count vehicles number which passing on the road during a particular time. The next class is *Optimizer*, which includes the *optimizerTraffic* method, is used to optimize the signal, time and direction at the moment time of traffic flow. The UML class diagram of initial ARTC system is shown in Fig. 2.

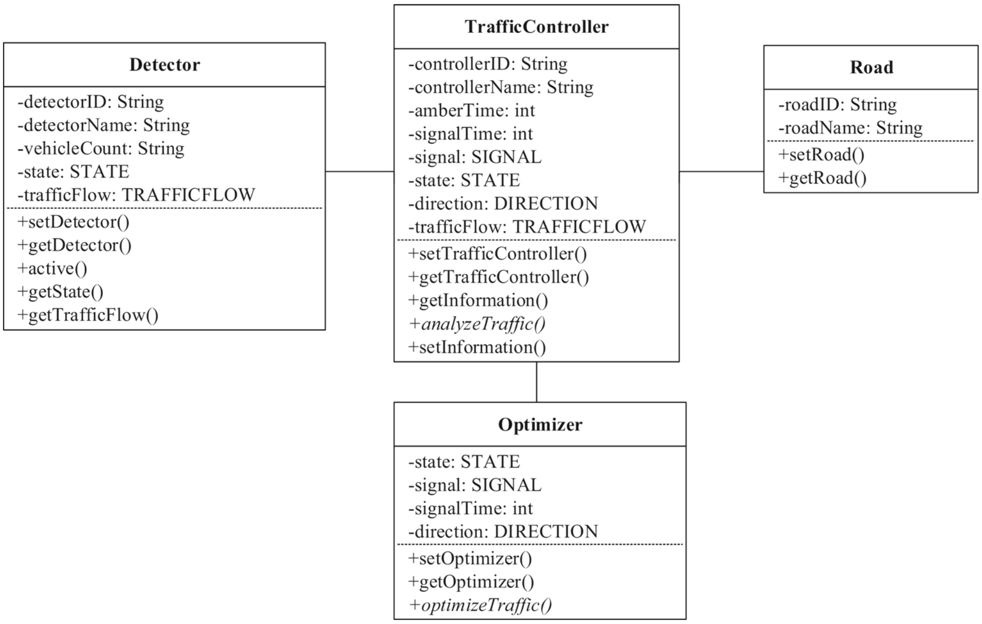
The UML sequence diagram have accomplished the task of showing how the objects interact with each other in a scenario. We will demonstrate our approach with the main scenario: *gettrafficFlow()* and *optimizeTraffic()*. This scenario of the system is depicted in Fig. 3.

4.2 Applying patterns

In Fig. 2, the method *optimizeTraffic()* belong to class *Optimizer*, which is employed to optimize light signals of the ARTC system. This solution is adequate if traffic environment is stable or witnesses no considerable changes. However, the system design may have following problems with the *optimizeTraffic()* of the class *Optimizer*:

- Algorithms are so complex to implement in one, therefore make the source code as large and arduous to maintain.

Fig. 2 The initial class diagram of ARTC system



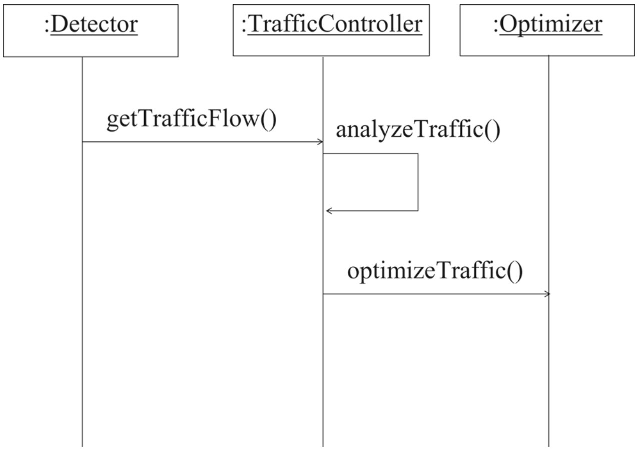


Fig. 3 Sequence diagram for calculating optimal control

- It takes time as well as effort to add new algorithms to the existing ones.
- The code of the existing algorithms are difficult to reuse, especially when one wants to create a hierarchy from *Optimizer* class.

In order to overcome these limitations, we are going to optimize the ARTC system by using Strategy pattern. As illustrated in Fig. 4, we detach three optimization strategies (*SignalOptimizeStrategy, TimeLimitOptimizeStrategy, AdjacentOptimizeStrategy*) from the class *Optimizer* then formed a hierarchy of algorithm classes that share the interface *OptimizerStrategy*. After applying Strategy pattern, the sequence diagram of the scenario calculating optimal control is re-drawed in Fig. 5.

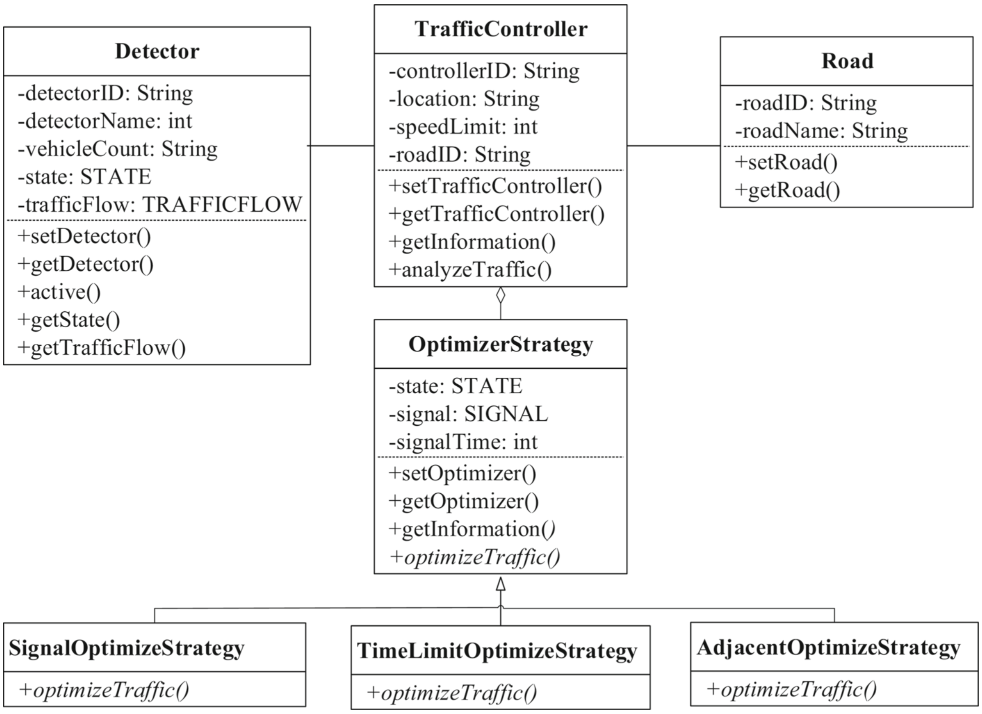
4.3 Behaviour preservation

ARTC system has a real time characteristic because of immediate responses to variant of traffic flow conditions. It should be executed in efficient way and returns to correct result. In ARTC system, without loss of generality, we assume that detector works in specific direction, some identified constraints need to be preserved are:

- If the **state** is *heavyTraffic* and the **signal** is *red*, it will be ensured that the **signal** is turned to *green*.
- If the **state** is *lowTraffic* and the **signal** is *green*, it will be ensured that the **signal** is turned to *red*.
- If the **state** is *heavyTraffic* and **signal** is *green*, it will be ensured that the **signalTime** is increased.
- If the **state** is *lowTraffic* and **signal** is *red*, it will be ensured that the **signalTime** is increased.
- If the **state** is *highTraffic* and **direction** is *noChoose*, it will be ensured that the **direction** is turned to *choose*.
- If the **state** is *lowTraffic* and **direction** is *choose*, it will be ensured that the **direction** is turned to *noChoose*.

In this article, we are going to check the system's behaviors preservation at a variety of different phases, especially Design and Implementation. For the former, we use the Object Constraint Language (OCL) to represent these

Fig. 4 Class diagram of ARTC system after applying Strategy pattern



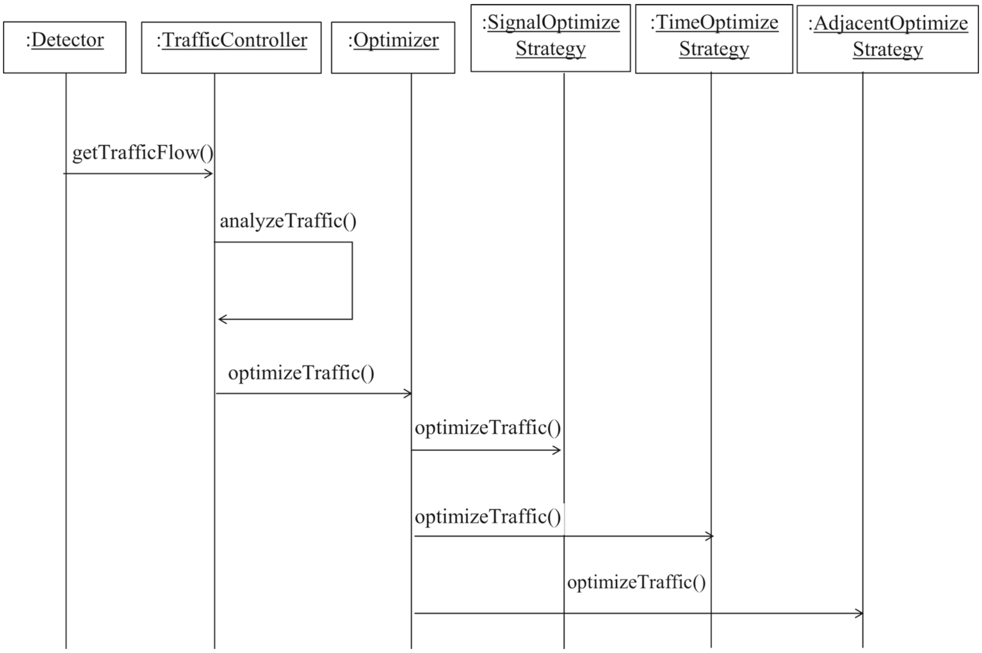


Fig. 5 Sequence diagram for calculating optimal control after applying Strategy pattern

constraints, for the latter, Java Modeling Language (JML) is employed to annotate into purely Java code to guarantee the correct behavioral execution.

5 Approach overview

In this section, we outline the proposed approach depicted in Fig. 6. There are three constituent which have been identified: (1) refactoring, (2) computing pre/post-conditions of scenarios, and (3) checking consistency. Note that, in this article we spend the most interested in *checking consistency process which is executed in both design and implementation phases* of the system.

To check consistency at design phase, we use class and sequence UML diagrams to model the software system. After applying design patterns in refactoring process, we are going to check whether the system's behaviors are preserved or not? In order to achieve this intention, we employ Object Constraint Language (OCL) to describe the constraints of the system which include invariant and pre/post-conditions. At design level, the essential of refactoring process is redistribute classes and methods [21], therefore, based on the defined consistent rules, we re-compute all constraints on the refactored models, then compare with their original and give out the result of checking consistency at the design phase.

Checking the consistency at implementation has been done analogously. The implementation phase of the software life cycle is the process of realization the specified design document into executable programming language code. In this article, we use Java to implement the motivating example. Thefore, we employ the JML to complement to Java code to portray the constraints of the system. With the support of existing tools [9, 17], the checking consistency at implementation phase is done automatically.

Generally, we have been checked the consistency of the software system at variety phases after applying design pattern in refactoring process. The result of each stage is the foundation to improve the reliability of the software system.

6 Formalization of software designs

In this section, we introduce the formalizations which are used as a foundation to check the consistency in software refactoring. When designing a software system with UML, it can be presented multiple scenarios consisting of several diagrams. Among them, scenario sequence diagrams are used for depicting the behavior of main functionalities. UML constraints are employed to deal with the request that conditions and restrictions on an UML element which must be satisfied in order to fulfill the functionality requirement. There are three standard types of constraints, which includes invariant, precondition, and postcondition, are defined on classes as follows.

Invariant: a class invariant defined on class attributes is a condition that every instance of that class must satisfy.

Precondition: a precondition of an operation is a condition that operation has to satisfy before executing.

Postcondition: a postcondition of an operation is a condition that operation has to satisfy after executing.

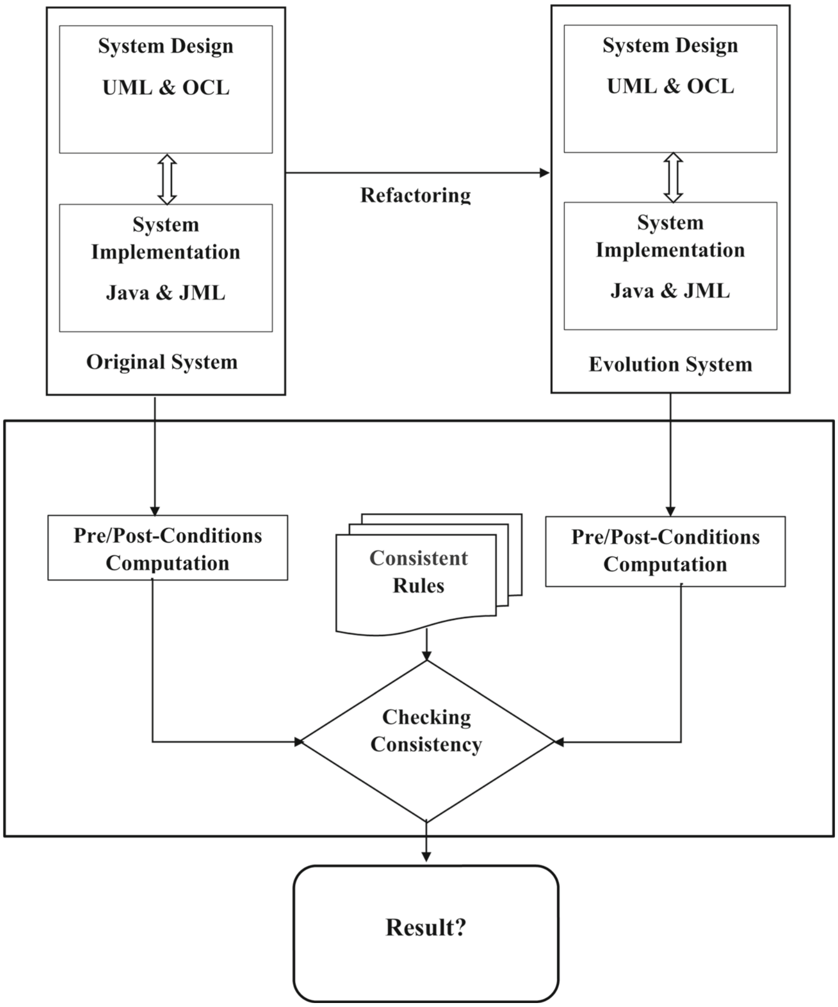


Fig. 6 The approach to checking consistency

In this paper, we introduce two more constraints, which are defined on scenarios of the design.

Scenario precondition: a scenario precondition is a constraint such that the state of the all involved classes' attributes must satisfy before the scenario starts.

Scenario postcondition: a scenario postcondition is a constraint such that the state of the all involved classes' attributes must satisfy after the scenario finishes.

The details of formal representation of the design are given as follows:

Definition 1 (Model) A model M is a tuple $\langle C_M, S_M \rangle$, where C_M is a set of classes and S_M is a set of behavioral scenarios.

Definition 2 (Class) A class $Ci_M \in C_M$ is formalized by a 3-tuple $Ci_M = \langle OP_{CiM}, A_{CiM}, I_{CiM} \rangle$, where OP_{CiM} is a set of public operations, A_{CiM} is a set of public attributes, and I_{CiM} states a set of class invariants.

Definition 3 (Abstract operation precondition) The precondition $PRE_{op_{iM}}$ of the op_{ei} in the abstract class Ci_M which is overridden by N operations op_{ei} in concrete classes $Cksi_M$ is defined by the disjunction of the precondition of all the op_{ei} in the concrete classes $Cksi_M$.

Assume $Pi(op_{ei})$ is the precondition of the operation op_{ei} in every concrete class, we can compute the precondition for op_{ei} in the abstract class according to this formula $PRE_{op_{iM}} = \bigvee Pi(op_{ei})$, where $op_{ei} \in Cksi_M$ is a realization operation, and P is predicate which dedicates the precondition of the op_{ei}.

Definition 4 (Abstract operation postcondition) The postcondition $POST_{op_{iM}}$ of the op_{ei} in the abstract class Ci_M which is overridden by N operations op_{ei} in concrete classes $Cksi_M$ is defined by the disjunction of the postcondition of all the op_{ei} in the concrete classes $Cksi_M$.

In a similar way with the abstract operation precondition, it is easy to realized that $POST_{op_{iM}} = \bigvee Pi(op_{ei})$, where $op_{ei} \in Cksi_M$ is a realization operation, and P is predicate which dedicates the postcondition of the op_{ei}.

Behavioral scenarios of the model represent the system external behavior. In this paper, we consider scenarios as sequence diagrams.

Definition 5 (Scenario) A scenario Si_M is represented by a 4-tuple $Si_M = \langle CI_{SiM}, PRE_{SiM}, E_{SiM}, POST_{SiM} \rangle$ where $CI_{SiM} \subseteq Ci_M$ represents a set of classes involved in the scenario, PRE_{SiM} is the scenario precondition, E_{SiM} is a sequence of operations of involved classes, and $POST_{SiM}$ states the scenario postcondition.

Definition 6 (Scenario operation) An operation in the scenario is a 4-tuple $Ek_{SiM} = \left\langle PRE_{Ek_{SiM}}, OP_{Ek_{SiM}}, POST_{Ek_{SiM}}, k \right\rangle$, where $PRE_{Ek_{SiM}}$ states the operation precondition, $OP_{Ek_{SiM}}$ is the public operation of the involved in the scenario, $POST_{Ek_{SiM}}$ is the operation postcondition, and k is the execution order of operation in the scenario.

In this article, we consider the case that both preconditions and post-conditions of an operation is the conjunction of predicates on the attributes of classes involved in the scenario, i.e., $PRE_{Ek_{SiM}} = \bigwedge P(A_{CijM})$, where $A_{CijM} \in A_{CiM}$ is a attribute, $CiM \subseteq CI_{SiM}$ and P is predicate. A scenario consists of a sequence of operations, hence its pre/post-conditions are formed by their pre/post-conditions. Note that, one public operation of a class in different scenario may have different prep/post-conditions. A scenario pre/post-condition is defined on post-conditions of all operations involved in the scenario as follows.

Definition 7 (Scenario precondition) The scenario precondition PRE_{SiM} is defined by the precondition of the first happened operation in the scenario.

The precondition of the first operation in the scenario specifies constraints of all scenario-related public attributes.

Definition 8 (Scenario postcondition) The scenario postcondition $POST_{SiM}$ is defined by the conjunction of the constraint on public attribute A_{CijM} in the operation postcondition $POST_{Ek_{SiM}}$ of the last happened operation in E_{SiM}.

Let a scenario $S = (e_1, e_2, ..e_n)$, where e_i, $i = \overline{1..n}$, is the i-th operation happened in the scenario. From the Def. 6, we have $e_i = (pre_{ei}, op_{ei}, post_{ei}, i)$ and $post_(ei) = \bigwedge P_k(A_kC)$, where P_k are the predicate on A_kC, which is the attribute of class C involved in the scenario. Assume that the scenario has one public attribute A_C that appears in both postconditions of two scenario operations e_i and e_j such that $1 \leq i < j \leq n$. Then we have $post_{ei} = P_i(A_C)$ and $post_{ej} = P_j(A_C)$. Since e_i happens before e_j, $P_j(A_C)$ must be hold after executing the j-th operation.

Definition 9 (Refactor) A refactor R using design patterns is denoted $R : M \xmapsto{\mathrm{D}(SUB_{MS})} M'$, where M and M' are the original model and its evolution, respectively, D is the applied pattern name, and $SUB_{MS} \subseteq S_M$ is set of affected scenarios.

7 Verifying consistency of software refactoring

7.1 Verification at design phase

In this section, we discuss how to verify the consistency of a model after applying design patterns. For the sake of reducing the difficulty in checking consistency, we classify preservation properties into two categories: static and dynamic preservation. For the former perspective, we take into account the invariant of classes. To check the latter, we concentrate to the pre/postconditions of selected scenarios. We propose to verifying such properties by giving new propositions of preservation.

Verifying static preservation

Proposition 1 (Static preservation) *A refactored model is called the static preservation with the original one if invariants of its classes are logically equivalent with the original ones.*

Formally, let $R : M \xmapsto{D(SUB_{MS})} M'$ *be a refactor,* M' *is called the static preservation with* M *if* $\forall Ci_M \mid Ci_M \in C_M \wedge Ci_M \in C'_M \implies I_{CiM} \equiv I_{CiM'}$.

Verifying dynamic preservation The critical requirement of refactoring with design patterns is that the refactored model does not change the behavior. The behavior can be described via scenarios. Hence, we need to check if a set of selected scenarios of a model is whether or not behavior preservation after refactoring.

Proposition 2 (Total dynamic consistency) *A refactored model is said to be total dynamic consistency with the original one if precondition and postcondition of any scenario execution are logically equivalent with the original ones.*

Formally, let $R : M \xmapsto{D(SUB_{MS})} M'$ *be a refactor,* M' *is called the total dynamic consistency with* M *if* $\forall S_{iM} \in SUB_{MS} \wedge PRE_{SiM} \equiv PRE_{SiM'} \wedge POST_{SiM} \equiv POST_{SiM'}$.

In Section 6, we state that the scenario pre/post conditions can be computed from scenario involved operations, hence if $PRE_{SiM} \equiv PRE_{SiM'} \wedge POST_{SiM} \equiv POST_{SiM'}$, then all constraints of public attributes of the refactored scenario are preserved before/after its execution.

Proposition 3 (Partial dynamic consistency) *A refactored model is said to be partial dynamic consistency with the original one if with any scenario execution, its preconditions are preserved and its postconditions are satisfied the one of original model's scenario.*

Formally, let $R : M \xmapsto{D(SUB_{MS})} M'$ *be a refactor,* M' *is called the partial dynamic consistency with* M *if* $\forall S_{iM} \in SUB_{MS} \wedge PRE_{SiM} \equiv PRE_{SiM'} \wedge POST_{SiM} \implies POST_{SiM'}$.

In this case, if $POST_{SiM} \implies POST_{SiM'}$, then the values of public attribute after executing are still in expected range.

Proposition 4 (Model inconsistency) *A refactored model is called the inconsistency with the original one if it violates consistent rules (static and/or dynamic consistency).*

The proposition of consistent rules between original model and refactored model is the basis that we used to check the consistency in software model evolution.

7.2 Verification at implementation phase

In Section 6, we address that the pre/post-conditions of a scenario can be computed from pre/post conditions of involved operations. From implementation perspective, the execution of a scenario must be preserved its pre/post-conditions.

Proposition 5 *[Execution preservation of refactored program] A refactored program* P' *is said to be execution preservation with the original one* P *if with the same scenario execution, its preconditions are preserved before and its postconditions are hold after execution.*

It is formally defined as clause: $PRE_{S_P} \equiv PRE_{S_{P'}} \wedge POST_{S_P} \equiv POST_{S_{P'}}$.

In this proposition, the scenario pre/post-conditions of the refactored program are figured out through the scenario one of the original program according to Defs. 7 and 8.

8 Illustrating with ARTC system

8.1 Checking consistency of design refactoring

According to Proposition 1, we can see that the process of model evolution has preserved invariants in both models so that static preservation is verified.

We now consider the dynamic preservation of the model evolution. The preconditions and postconditions of the scenario *calculating optimal control()* specified in initial model (Figs. 2 and 3) can be calculated, according to Defs. 7 and 8 as the following:

```
PRE_S'iM = trafficFlow -> isEmpty()
POST_S'iM = if (state = heavyTraffic) then ((signal =
      green) AND (greenTime > 60))
           else if (state = lowTraffic) then ((signal =
       red) AND (greenTime <= 60))
          else if (state = highTraffic) then (direction
       = CHOOSE)
                      else (direction = NO_CHOOSE)
                      endif
          endif
       endif
```

In a similar way, we figure out the preconditions and postconditions of the scenario *calculatingoptimalcontrol()* in the evolution model (Figs. 4 and 5):

```
PRE_SiM = trafficFlow -> isEmpty()
POST_SiM = if (state = heavyTraffic) then ((signal =
      green) AND (greenTime > 60))
          else if (state = lowTraffic) then ((signal =
      red) AND (greenTime <= 60))
         else if (state = highTraffic) then (direction
      = CHOOSE)
                     else (direction = NO_CHOOSE)
         endif
      endif
      endif
```

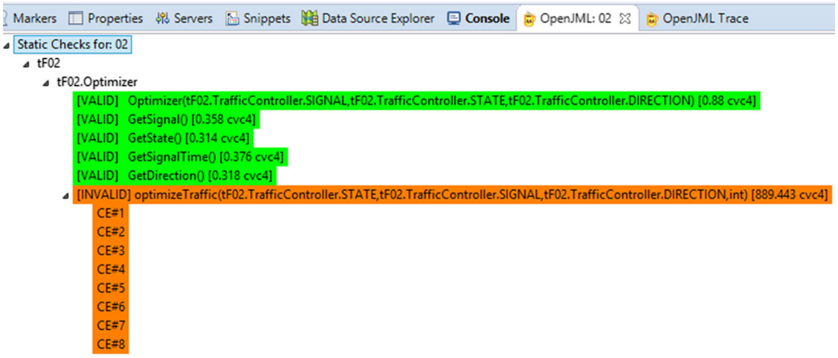
From the values of preconditions and postconditions above, we conclude that the scenario is strongly preserved all the attributes of the model. According to the Proposition 2, the refactoring process with Strategy pattern in this case-study leads to the consistent behavior between the original and evolution models. Note that, OCL description is a form of pseudo first-order logic, it can be transformed to pure first order predicate logic [4] so that we can check automatically the consistency conditions in software evolution model using theorem provers [10].

8.2 Checking consistency of source code refactoring

Back to the example in Section 4, all initial behaviors specifications of the ARTC system have been validated checking on Eclipse software by plug-in OpenJML. Now, after refactoring, we are going to consider whether evolution program satisfy all behaviors specification of the initial program.

In experiments, we have carried out the implementation the source code of the ARTC system after refactoring. Based on the set of rules which was built in this Section, we have figured out the pre/post-conditions of the evolution scenario as well as checked the constraints on it. The experimental results are illustrated in Fig. 7 where we can observe that the execution of *optimizeTraffic* does not preserve the consistency.

In other words, the refactored program does not preserve all behaviors of the initial one, so ones should consider the applied refactoring process.



```
Static Checks for: 02
  tF02
    tF02.Optimizer
      [VALID]  Optimizer(tF02.TrafficController.SIGNAL,tF02.TrafficController.STATE,tF02.TrafficController.DIRECTION) [0.88 cvc4]
      [VALID]  GetSignal() [0.358 cvc4]
      [VALID]  GetState() [0.314 cvc4]
      [VALID]  GetSignalTime() [0.376 cvc4]
      [VALID]  GetDirection() [0.318 cvc4]
      [INVALID] optimizeTraffic(tF02.TrafficController.STATE,tF02.TrafficController.SIGNAL,tF02.TrafficController.DIRECTION,int) [889.443 cvc4]
        CE#1
        CE#2
        CE#3
        CE#4
        CE#5
        CE#6
        CE#7
        CE#8
```

Fig. 7 The result of checking behavior preservation on refactored program

9 Conclusions and future work

In this paper, we have proposed an approach to checking the consistency between original software system and its refactoring after applying design patterns.

We have conducted consistency checks at design and implementation phases by introducing new formalizations. To check the former, we modeled the system by UML and use OCL to describe the constraints. To check the latter, the Java programs were annotated by JML and automatically checked by OpenJML integrated into Eclipse environment. Our approach expressed the theoretical unification within the field of software engineering by means of checking consistency throughout from design to implementation phase as well as feasibility of experimental execution.

Check the consistency of a program or a software model has been received a lot of interest, however, the existing research results focus on the consistency between different phases of life cycle development model (e.g., implementation and design phase) or different diagrams of a model (e.g., state diagrams and sequence diagrams). Our approach pays attention in checking consistency between original software system and its evolution in both the design and implementation phase.

To illustrate the proposed approach, we have conducted experiments on the ARTC system. In the motivating example, we just illustrated only the consistency verification when applying Strategy pattern in the only a pair of scenario, respectively in the both programs, other scenarios may be done in a similar way for the more complex system.

As mention earlier, we can see that the calculation of preconditions and post-conditions of scenarios is time-consumed and error-prone if we do it manually. For the future works, we will adopt tools to calculate automatically constraints and verify the program evolution process.

Acknowledgments This work is partly supported by the project no. 102.03–2014.40 granted by Vietnam National Foundation for Science and Technology Development (Nafosted).

References

1. Object management group. ocl 2.4 specification. (2014)
2. Advice leaflet 1: The “scoot” urban traffic control system: http://www.scoot-utc.com, (2016)
3. Alexander C, Ishikawa S, Silverstein M (1977) Pattern languages. Center for Environmental Structure 2:1977
4. Beckert B, Keller U, Schmitt PH (2002) Translating the object constraint language into first-order predicate logic. In: Proceedings, VERIFY, workshop at federated logic conferences (FLoC), pp 113–123
5. Bottoni P, Parisi-Presicce F, Taentzer G (2003) Coordinated distributed diagram transformation for software evolution. Electron Notes Theor Comput Sci 72(4):59–70
6. Bottoni P, Parisi-Presicce F, Taentzer G (2003) Coordinated distributed diagram transformation for software evolution1 1partially supported by the ec under research and training network segravis. Electron Notes Theor Comput Sci 72(4):59–70
7. Burdy L, Cheon Y, Cok DR, Ernst MD, Kiniry JR, Leavens GT, Rustan K, Leino M, Poll E (2005) An overview of jml tools and applications. Int J Softw Tools Technol Transfer 7(3):212–232
8. Cabot J, Gogolla M (2012) Object constraint language (ocl): a definitive guide. In: Proceedings of the 12th international conference on formal methods for the design of computer, communication, and software systems: formal methods for model-driven engineering, SFM’12. Springer, Berlin, Heidelberg, pp 58–90
9. Cok DR (2011) Openjml: Jml for java 7 by extending openjdk. In: NASA formal methods symposium. Springer, pp 472–479
10. De Moura L, Bjørner N (2008) Z3: an efficient smt solver. In: Proceedings of the theory and practice of software, 14th international conference on tools and algorithms for the construction and analysis of systems, TACAS’08/ETAPS’08. Springer, Berlin, Heidelberg, pp 337–340
11. Dong J, Sheng Y, Zhang K (2006) A model transformation approach for design pattern evolutions. In: 13th annual IEEE international symposium and workshop on engineering of computer based systems, 2006. ECBS 2006, pp 10–92
12. Van Eetvelde N, Janssens D (2003) A hierarchical program representation for refactoring, vol 82. UNIGRA’03, Uniform Approaches to Graphical Process Specification Techniques (Satellite Event for {ETAPS} 2003)
13. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston
14. He J (2005) Consistency checking of uml requirements. In: Proceedings of the 10th IEEE international conference on engineering of complex computer systems, ICECCS ’05. IEEE Computer Society, Washington, DC, USA, pp 411–420

15. Leavens GT, Baker AL, Ruby C (2006) Preliminary design of jml: a behavioral interface specification language for java. ACM SIGSOFT Software Engineering Notes 31(3):1–38
16. Leavens GT, Poll E, Clifton C, Cheon Y, Ruby C, Cok D, Müller P, Kiniry J, Chalin P, Zimmerman DM, Dietl W, et al. (2008) Available from http://www.jmlspecs.org/
17. Marché C, Paulin-Mohring C, Urbain X (2004) The krakatoa tool for certificationof java/javacard programs annotated in jml. J Logic Algebraic Program 58(1):89–106
18. Matsumoto T, Sakunkonchak T, Saito H, Fujita M (2003) Verification of behavioral consistency in c by using symbolic simulation and program slicer. In: Model checking for dependable software-intensive systems workshop, pp 80–84
19. Mens T, Tourwe T (2004) A survey of software refactoring. IEEE Trans Softw Eng 30(2):126–139
20. Mens T, Demeyer S, Janssens D (2002) ForMalising behaviour preserving program transformations. In: Corradini A, Ehrig H, Kreowski H-J, Rozenberg G (eds) Graph transformation, volume 2505 of lecture notes in computer science. Springer, Berlin Heidelberg, pp 286–301
21. Opdyke WF (1992) Refactoring: a program restructuring aid in designing object-oriented application frameworks. PhD thesis, University of Illinois at Urbana-Champaign
22. Rasch H, Wehrheim H (2003) Checking consistency in uml diagrams: classes and state machines. In: Najm E, Nestmann U, Stevens P (eds) Formal methods for open object-based distributed systems, volume 2884 of lecture notes in computer science. Springer, Berlin Heidelberg, pp 229–243
23. Van Der Straeten R, Mens T, Simmonds J, Jonckers V (2003) Using description logic to maintain consistency between uml models. In: Stevens P, Whittle J, Booch G (eds) ÇUMLÈ 2003 - the unified modeling language. Modeling languages and applications, volume 2863 of lecture notes in computer science. Springer, Berlin Heidelberg, pp 326–340
24. Ward MP, Bennett KH (1995) Formal methods to aid the evolution of software. Int J Softw Eng Knowl Eng 5:25–47
25. Zhao C, Kong J, Zhang K (2007) Design pattern evolution and verification using graph transformation. In: 40th annual hawaii international conference on system sciences, 2007. HICSS 2007, pp 290a–290a