Maintaining Consistency between a Design Model and Its Implementation

Hector M. Chavez
Western Michigan University, hmch.chavez@gmail.com

MAINTAINING CONSISTENCY BETWEEN A DESIGN MODEL AND ITS IMPLEMENTATION

by

Hector M. Chavez

A dissertation submitted to the Graduate College
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
Computer Science
Western Michigan University
August 2015

MAINTAINING CONSISTENCY BETWEEN A DESIGN MODEL AND ITS IMPLEMENTATION

Hector M. Chavez, Ph.D.

Western Michigan University, 2015

Software design models are increasingly being used as part of the software development process as analysis and design artifacts and to automatically generate code that developers can further modify or extend, greatly expediting the software development process. This, however, has introduced the challenge of maintaining consistency between the design models and their implementation as they evolve during the development process. Traditional software testing and verification techniques have been well studied in the past, and they are an integral part of many software development projects, however, they are not well suited for consistency checking between a design model and its implementation. In this dissertation, we present a testing-based validation technique that improves over traditional testing and verification techniques for consistency checking that combines coverage criteria and dynamic symbolic execution with path condition analysis. Our experiments have shown the effectiveness and efficiency of our approach when applied to industry strength software systems.

ACKNOWLEDGEMENTS

I first would like to thank my advisor Wuwei Shen for all his help during my years as a graduate student. His guidance and support has been an integral part on the development of my research work, and being where I'm would not have been possible without his constant encouragement. I'm greatly in debt for all that he taught me and for introducing me to the ideas that ultimately lead to this thesis.

I want to thank the members of my doctoral committee for their input when working on my thesis and research. To all my professors at WMU from whom I have learned a great deal during my graduate studies. And to colleges and members of our research group who collaborated with me, specially Benjamin Mechling who worked with me in the implementation of CCUJ's dynamic symbolic execution.

Outside from our research group, I would like to thank the developers in the modeling community who where kind enough to answers many of my questions when working in the project. To Kenn Hussey from the Eclipse Community Forums, for his suggestions and help confirming some of the bugs and findings in my research.

Finally, I would like to thank my family and friends for all their support and encouragement. My parents Miguel and Socorro for their unconditional support and love. My brothers Abel and Jose, and sisters, Marivel and Alicia, for always being there when I need them. And lastly, to my uncle and aunt Andy and Alicia Blodgett, for all their support and for allowing me to be part of their family during my stay in Michigan.

Hector M. Chavez

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In the past three decades, we have become increasingly dependent on computer applications. From simple embedded applications in domestic appliances to medical hardware and information systems, computer applications have penetrated into every aspect of life, making possible most of the activities that are an integral part of modern societies. This wide adoption of computer applications has made software correctness one of the primary goals in the software development process.

Unfortunately, ensuring software correctness has become increasingly challenging as the size and complexity of software applications increases. As an example, the first release of the Linux operating system kernel in 1991 consisted of about 10,000 source lines of code (SLOC), and twelve years later, the kernel version 2.6.0 included over 5 million SLOC [1]. Also, the introduction of hardware improvements has made software development harder. For example, the use of multi-core systems has introduced non-deterministic behavior that makes it harder to implement correct reliable software.

As our reliance on software applications increases, the impact of software errors, or software problems that affect a computer application, has serious consequences in our daily life. A study conducted by the National Institute of Standards and Technology in 2002 estimates that software errors cost about $59 billion every year. This includes resources used by the user to mitigate the undesired effects and the cost for developers to fix the problems [2]. Other errors could have more devastating effects, such as life losses caused by failures in medical equipment [3] or transportation systems.

Numerous techniques have been used to improve the quality and reduce the likelihood or number of errors in computer applications, as well as speed up the development process. These techniques are normally seen as part of the software engineering discipline, an engineering discipline which applies systematic approaches to various phases during the software development process such as design, implementation, testing and maintenance phases [4].

With the increasing size and complexity of a problem domain, it becomes harder to understand, design, and maintain a software system. This creates a gap between the problem domain, which has become more complex as available hardware improves, and the solution domain. One way of tackling the complexity problem is by using a stepwise refinement approach, where at each step a new level of abstraction is introduced, going from the more general, e.g., a high level design, to the more detailed, e.g., a program implementation. It is in this area where Model Driven Architecture (MDA) [6] plays an important role by trying to link the artifacts produced at different levels of abstraction. One way MDA tries to achieve this goal is with the use of forward engineering tools, where an artifact, such as a *class diagram*, can be used to automatically generate a more detailed artifact, e.g., a Java program, greatly expediting the development process.

However, most forward engineering tools only generate skeletal programs and it is the developer's responsibility to manually implement the features not supported by the forward engineering tool. This can lead to inconsistencies introduced by the developers when extending the generated skeletal program that may cause undesired behavior on a system. As a result, it is a challenging issue to determine if the developer's implementation satisfies the constraints specified in the model. Specifically, at the core of the Model Driven Architecture, the Unified Modeling Language (UML) [7], a general purpose modeling language, is used to describe a system's structure, behavior, and architecture. And the Object Constraint Language (OCL) [8] has been proposed as a complementary specification language that can be used to define constraints over the elements in a design model.

Existing approaches have studied the problem of consistency checking between different types of models but non of them have considered consistency checking between a design model and its implementation [34, 35, 36, 37]. Also, reverse engineering techniques have been applied to check specific UML class diagram properties, such as *association* properties, however, their approaches are limited at recovering properties that can be statically inferred, they don't recover features that require a program execution, such as the UML *composition* property. Also related to consistency checking, some existing approaches consider the generation of test cases from behavioral models [22, 23, 24, 25, 26, 27, 28, 29] such as activity or sequence diagrams, but do not consider class diagrams or method constraints.

To tackle the issue of consistency checking between a design model and its implementation, we introduce a bounded exhaustive testing technique with a pruning approach that analyses the fields accessed during the execution of the method under test. This technique is based on the *small scope hypothesis* and it can efficiently generate a small number of test cases. However, it cannot guaranty the correctness of the program. To overcome this limitation, we also introduce an approach to consistency checking between a model, i.e., class diagram, and its Java implementation, in terms of the OCL constraints specified in the model as method post conditions. In this approach, we automatically generate a Java Boolean post method from a method post condition and determine whether the method always returns true right after the execution of the method under test. We follow a testing-based validation technique that combines coverage criteria and dynamic symbolic execution.

Unlike traditional testing, our approach considers the relationship of the path conditions of the method under test and its post method when generating test cases, and thus that allows us to guarantee the correctness of the method under test in terms of its OCL post condition. On the other hand, traditional program testing techniques only rely on different coverage criteria, e.g., branch coverage, to generate test cases and assume that the execution of a faulty statement can expose a software fault, which is not always the case. Also, traditional testing techniques modify the values of the fields that appear in branch conditions in order to explore different execution paths in the program. This can be problematic for consistency checking in the context of Model Driven Architecture, since the automatically generated code often includes auxiliary fields with values that correspond to specific features in the model. Modifying auxiliary fields to cover a different execution path could result in a false positive due to the underlying program structure no longer matching the original model. Moreover, traditional testing techniques cannot handle cases with *black box* methods, i.e., methods that cannot be symbolically executed. To handle this scenarios, we introduce an approximate symbolic execution technique, an extension over traditional program testing.

The main contributions of this dissertation are the following:

- We introduce a bounded exhaustive testing technique to check the consistency between UML association properties in a class diagram and their implementation.

- We introduce a model based testing approach that determines whether OCL constraints in a UML class diagram are violated in a Java implementation.

- We introduce a new pruning technique, that analyses the path conditions of the method under test and its post condition method.

- We introduce an approximate symbolic execution technique, an improvement over traditional testing, that can handle the execution of black box methods.

- We present an approach to automatically generate a set of high-quality test cases to efficiently explore different execution paths of the implementation.
- We applied our approach to two EMF UML2 industry-strength projects.

1.1 Organization

The remainder of this dissertation is organized as follows. Chapter II introduces background topics related to consistency checking and related work, such as model driven architecture, and dynamic symbolic execution. Chapter III presents our approach to consistency checking with bounded exhaustive testing. Chapter IV introduces our approach to consistency checking with path condition analysis, which improves over bounded exhaustive testing. Chapter V discusses the evaluation of both approaches and shows experimental results. Chapter VI concludes.

CHAPTER II

BACKGROUND AND RELATED WORK

2.1 Unified Modeling Language

The Unified Modeling Language is a general purpose modeling language which can be used to describe a system from different aspects, such as structural, behavioral, and architecture. The Unified Modeling Language is composed by different types of diagrams, such as class diagrams, used to model the static structure of a system, and use case diagrams, to model its dynamic behavior. Together, the full set of UML diagrams can be used to fully specify, construct, and document the artifacts of a system [7]. Some other activities that can benefit from UML are system visualization and analysis, model execution for simulation, and testing. Among the different types of diagrams in UML, class diagrams, used to specify the static structure of a system, are the most widely adopted.

2.1.1 Class Diagrams

Class Diagrams (CD) in UML specify the static structure of a system. A class diagram shows the elements of a system, their relationships, and their internal structure. Many of the elements described in a class diagram can be mapped to a counterpart concept found in the Object-Oriented programming languages, such as classes, inheritance relationships, data types, class attributes, packages, interfaces, etc. But some elements may describe a high level of abstraction that have no direct counterpart in object oriented programming languages, such as association's in UML.

In a class diagram, a *Class* is represented by a rectangle with three compartments. The name of the class is shown in the top compartment, a list of attributes is placed on the middle compartment, and a list of operations is placed in the bottom compartment. Figure 2.1 shows an example of a class named *LoyaltyAccount* with these three sections. In the middle section we have a list of two attributes, *points*, and *number*, and the last section shows one operation, *earn()*. The middle and bottom compartments are optional. Some designs with a high level of abstraction may only require the name of the classes in the system.

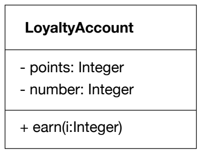


Figure 2.1: UML class with two attributes and one operation.

In the middle compartment of the class, each attribute is shown in a separate line. An attribute definition includes an optional access modifier, "+" or "-" for public or private respectively, followed by the name of the attribute and its type, separated by a colon. An attribute declaration can optionally define a default value by appending "= *value*" at the end of the attribute definition. Also, attributes can be marked as derived, meaning that their value is calculated at runtime using information from other attributes in the model. Similarly, the third compartment lists on each line an operation. The operation definition includes an access modifier, the name of the operation, the list of parameters between parenthesis with the format "*name : type*", and the operations return type. The list of parameters is optional, as well as the return type. In the case of class names and operation names, the identifier can be italicized to signify that the class or operation is abstract.

Another major element in a class diagram is the *inheritance* relationship that models the same inheritance concept found in object oriented programming. The inheritance relationship is modeled with a solid line and a closed arrow head next to the parent class. Figure 2.2 shows an example of the inheritance relationship. This example shows a parent class *Vehicle*, and two sub-classes, *SportsCar*, and *SUV*. As it can be observed in Figure 2.2, UML class diagrams support multiple inheritance, a concept not supported by some programming languages, such as Java.

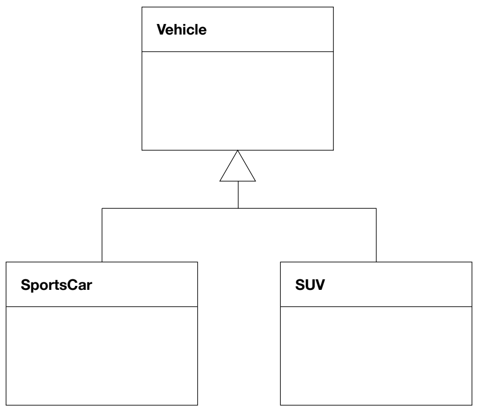


Figure 2.2: Inheritance relationship.

The next modeling element in a class diagram is the *association*. An association represents a kind of relationship between two elements in the diagram, e.g., classes. An association can be further defined with some properties, such as navigability, and multiplicity. Navigability on an association relationship is used to specify the accessibility of class instances. Navigability is represented by an arrow on the navigable end, if both ends are navigable, the arrows are normally omitted. Navigability is defined by the UML specification as follows: "*Navigability means instances participating in links at runtime (instances of an association) can be accessed efficiently from instances participating in links at the other ends of the association. The precise mechanism by which such access is achieved is implementation specific. If an end is not navigable, access from the other ends may or may not be possible, and if it is, it might not be efficient. Note that tools operating on UML models are not prevented from navigating associations from non-navigable ends.*" [7]. Thus, an association can be either *bi-directional* or *uni-directional*. In a uni-directional association, only instances that participate in one of the ends of the association can be efficiently accessed from instances participating in links at the other end. As an example, Figure 2.3 shows a class diagram composed by two classes, *Company*, and *Client*. The classes are related by a bi-directional association, this means instances of both classes can efficiently access instances of the class in the opposite end. Figure 2.4 shows an example of a uni-directional association, modeled with a solid line and an arrow head in one of its ends. In this example, the arrow head next to class *Client* indicates that instances of class *Company* can efficiently access instances of class *Client*, while instances of class *Client* can't efficiently access instances of class *Company*.

An association can also be annotated with a role name and multiplicity. The role name indicates how the relationship will be known to an instance of the class. For example, for class *Company*, the relationship with class *Client* will be known as *clients*. The multiplicity element indicates how many instances of class *Client* can be related to an instance of class *Company* and how man instances of class *Company* can be related to an instance of class *Client*. The multiplicity can be defined by an exact value or a range composed by a lower bound and an upper bound. An upper bound can also be defined with an asterisk "**"* that indicates an unbounded upper multiplicity. In Figure 2.3, the multiplicity "*0..2*" next to *Client* indicates that an instance of *Company* can be related to "*0*" or at most "*2*" instances of *Client*. The "***" next to *Company* indicates that an instance of *Client* can be related to "*0*" or more instances of *Company*.

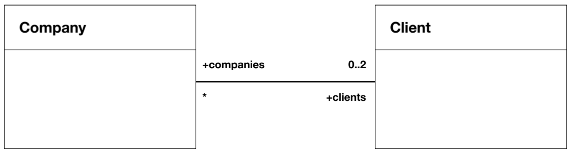


Figure 2.3: Class diagram with bi-directional association.

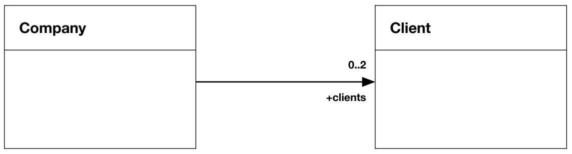


Figure 2.4: Class diagram with uni-directional association.

2.2 Object Constraint Language

As previously mentioned, a class diagram can only be used to specify the static structure of a system. As a result, the Object Constraint Language (OCL) [8] has been proposed to support UML and can be used to express pre or post-conditions and class invariants. The Object Constraint Language is used in the UML metamodel specification to define a set of well-formedness rules and constraints, and its introduction arose from the need to describe additional constraints about the objects in the model [8]. Usually, without a formal language, these constraints would be defined in natural language. OCL is a formal specification language that is guaranteed to have no side effects; the evaluation of an OCL expression only returns a value, and it is a typed language, where each expression has a type.

To illustrate the need of OCL as a specification language, we'll use the example presented by Warmer et al. [12]. Consider the class diagram in Figure 2.5, in this diagram, the association between class *Flight* and *Person* indicates that a flight can have zero or more passengers, indicated by the multiplicity of "*0..**". This indicates that an unlimited number of passengers can be part of the flight, but in reality the number of passengers should be limited by the number of seats in the airplane. To express this constraint, we use the OCL in Listing II.1. OCL can also be used to query the system, specify pre or post-conditions of operations, and specify invariants on classes.

```
context Flight
inv: self.passengers->size() <= self.plane.numberOfSeats
```

Listing II.1: Flight OCL constraint.

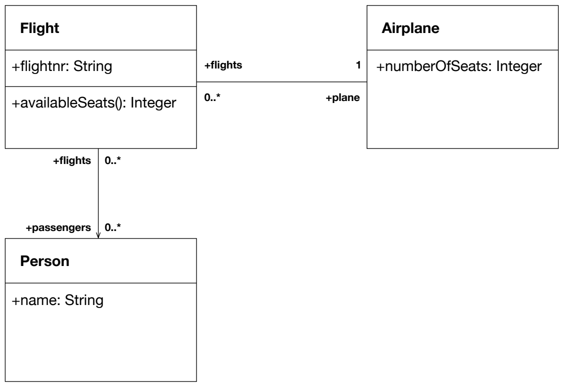


Figure 2.5: Flights class diagram without OCL.

One of the characteristics of an OCL expression is that it must be defined in the context of a class. In the example just introduced, the context is class *Flight* and the *inv* keyword indicates that the constraint is a class invariant, i.e., it must be satisfied by any instance of class *Flight* throughout its lifetime. The Object Constraint Language provides a wide variety of predefined functions and logic expressions designed to work in different data types, such as collection functions, iterators, and casting operations among others [8], which makes it a very powerful language for constraints specification. In the following subsections we introduce some of the most used types of expressions available in the OCL language.

2.2.1 Initial Values

Initial values can be used to define default values for attributes or association ends in the diagram [12]. In an initial value definition, the context is the name of the class followed by the name of the attribute being initialized. Next, the value is given as an expression after the keyword *init*. Note that the expression type must match the type of the attribute being initialized. In a similar way, OCL can be used to define the value of a *derived* attribute or association end by replacing the keyword *init* with *derived*.

```
context Person::isAlive
```

```
init: true

context Car::howOld
derived: currentYear - purchaseYear
```

2.2.2 Query Operations

Query operations are used to define operations that return a value and do not modify the system [12]. To define a query, the context is defined by the class name, operation name, parameters, and return type, and the expression is defined with the keyword *body*. A query operation without parameters can be seen as a derived attribute where only the context declaration varies. The following example shows the definition of a query operation. In this example, the return type is defined as a *Set* with elements of type *Service*. In the body expression, the collection *deliveredServices* is converted to a set with the operation *->asSet()* before being returned by the operation.

```
context LoyaltyProgram::getServices(): Set(Service)
body: partners.deliveredServices->asSet()
```

2.2.3 Attributes and Operations Definition

OCL can also be used to define attributes and operations, beyond the ones already defined in the model. Attributes and operations defined with OCL are derived attributes and query operations respectively. The context is the class where the attribute or operation is being defined. The definition is done with the keyword *def* followed by the name of the attribute or operation and its definition. In the following example, an operation is defined using the operation *select*. This operation selects elements that satisfy a Boolean expression. As illustrated in this example, $shift = s$ indicates that we want to select all the employees where *shift* is equal to the parameter *s*.

```
context Factory
def: numberOfEmployees : Integer = employees->size()

context Factory
def: numberOfEmployeesByShift(s: String): Integer =
     employees.select(shift = s)->size()
```

2.2.4 Invariants

An invariant is a rule attached to a class that must be satisfied by all of its instances during their lifetime [12]. As explained in an earlier example, an invariant includes the context followed by the keyword *inv* and a Boolean expression. The following example shows an invariant for class *Person*. In this example, keyword *self* is used to refer to the contextual instance, i.e., any instance of *Person*. Note that the use of *self* is optional, as seen in previous examples.

```
context Person
inv: self.age < 150
```

2.2.5 Preconditions and Postcondition

As mentioned earlier, another use of OCL is the specification of pre or post-conditions. An OCL post-condition for a method specifies a set of constraints that must be maintained after the execution of the method. This means, the result returned by the method and the resulting program state must satisfy the constraints defined by the OCL expression. An OCL post-condition definition includes a context declaration, which is composed by the name of the class and the name of the method, followed by the constraint. As an example, Listing II.2 shows a post-condition for method *updateIncome()* in class *Person*, with the constraint *self.income = newIncome*, indicating that the value of field income in class *Person* must be equal to the value of the parameter *newIncome* after the execution of the method.

```
context Person::updateIncome(newIncome:Integer):Boolean
```

```
post: self.income = newIncome
```

Listing II.2: OCL post-condition expression.

2.3 Model Driven Architecture

Model Driven Architecture (MDA) describes a software development approach where models are use as the primary source for software development. Some of the advantages or objectives when using MDA include portability, quality, maintenance, and improved testing and simulation. The following subsections introduce MDA's terminology, followed by an overview of the MDA approach.

2.3.1 Terminology

The terms introduced by MDA are in the context of an existing or planned system [6], a system can be a program, a single computer system, or a combination of different types of systems. A model of a system is a "*description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language.*" [6]. In the context of MDA, the specification is considered to be formal, meaning that the specification is "*based on a language that has a well-defined semantic meaning associated with each of its constructs, to distinguish it from a simple diagram showing boxes and lines.*" [13]. Thus, Model Driven Architecture describes an approach where models are used as the main source for software development, which includes documenting, analyzing, designing, constructing, and maintaining a system [6]. An application refers to the functionality being developed. And the architecture of a system is a specification of the parts and connectors of the system and the rules for the interactions of the parts using the connectors. Each of these parts, connectors and rules of a system are represented as a set of interconnected models, in the context of MDA [13].

A *viewpoint* on a system "*is a technique for abstraction using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within that system*" [13]. In the context of modeling, abstraction is a technique in which details are removed in order to simplify the model. The type or level of abstraction will determine the details that are of interest in a particular model. A *view* of a system is a specific representation of a chosen viewpoint, a viewpoint can be represented by one or more views. A platform is a set of technologies and functionality which an application can use without concern for how the functionality provided by the platform is implemented [6], and platform independent is a characteristic of a model indicating that the model is independent of the features of any given platform.

MDA explicitly defines three types of viewpoints with their corresponding models or views. The computational independent viewpoint "*focuses on the environment of the system, and the requirements for the system; the details of the structure and processing of the system are hidden or as yet undetermined*" [6]. And its corresponding model is the computation independent model (CIM). The CIM model serves as a domain model, and it mainly defines the requirements without concerns about specific details of the models or artifacts used to realize them. Next, the platform independent viewpoint focuses on the operation of a system, while hiding the detail necessary for a specific platform [6]. The platform independent model viewpoint must define the area of the system that does not change in different platforms. The platform independent model (PIM) serves as its view. Typically, a PIM is designed against a generic or neutral virtual machine, independent of any particular platform. The last viewpoint is a platform specific viewpoint, in which details about a specific platform are included, ant its corresponding view is the platform specific model (PSM).

A key aspect of MDA is the use of model transformations. A model transformation is the process of converting one model into another, such as a PIM to a PSM. How the transformation is done depends on a set of mapping or transformation rules [13]. Finally, an implementation "*is a specification, which provides all the information needed to construct a system and to put it into operation*" [6].

2.3.2 MDA Process

The MDA process starts with the specification of the system requirements with a computational independent model (CIM). The CIM defines the domain model and business rules and hides most of the details about how the data is processed or how the system is implemented. The CIM provides the vocabulary that will be shared and used by the rest of the models [6]. Next, a platform independent model is constructed (PIM) via UML, and it can finally be transformed into one or more specific platforms. Note that the MDA process is not restricted to vertical transformations across the different layers of abstraction (CIM, PIM, PSM), horizontal transformations may also occur. As an example, within PSM, a JUnit test model can be generated from a Java class model [13]. Also, a PSM model can take the role of PIM and be used for further transformations. In general, the key concepts of MDA are models and transformations, such as the transformation of a UML diagram to Java code. The MDA pattern can be visualized in Figure 2.6, taken from [13].

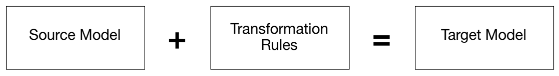


Figure 2.6: MDA pattern.

Central to the model transformation phase is the concept of transformation mappings. MDA transformation mappings specify how an element in a particular model maps to an element in a different model. A mapping can also define marks. A mark is a particular attribute that labels elements and further defines how the transformation is carried out. Mapping can also include templates, to specify particular kinds of transformations [6]. After a model has been marked, the final step is the transformation where the input is a marked PIM and the output is a PSM. Finally, a PSM can be translated to code. In some instances, a direct transformation from a PIM to code can be supported. The transformation process is shown in Figure 2.7 taken from [6]. There are multiple approaches that can be used for model transformation. One such approach is the one previously mentioned and shown in Figure 2.7 that makes use of markings. Other approaches include metamodel and model transformations with transformations specifications, pattern transformations, and model merging.

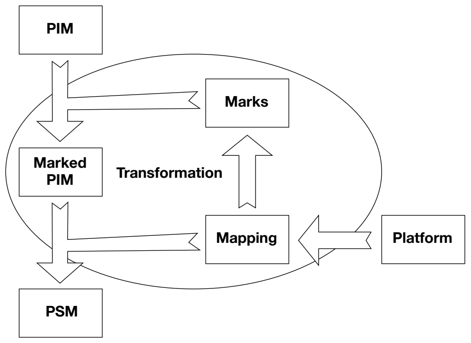


Figure 2.7: Model marking.

A variety of technologies have been adopted to support the MDA approach, this include the Unified Modeling Language (UML), the standard modeling language for specification and visualization, the Meta-Object facility, a model repository that can be used to specify model manipulation standards, and Profiles, used as UML extension mechanisms.

2.3.3 UML to Java

To illustrate the MDA process, we will use a model transformation example from a UML class diagram (PSM) to a Java program (Code). This type of transformation is commonly known as *forward engineering*, an important part of the Model Driven Architecture that greatly expedites the development process by automatically generating a skeletal program the developers can extend. As previously explained, the translation process takes a source model and a set of transformation rules to produce a target model. Figure 2.8 shows the diagram for our specific example. As transformation rules we will use the ones defined by the *forward engineering* tool Rational Software Architect [14]. The rules are shown in Table 2.1. As source model, we will use the class diagram shown in Figure 2.9.

Class Diagram
+
Transformation Rules
=
Java Program

Figure 2.8: Class diagram (PSM) to Java program (Code).

Table 2.1: RSA transformation rules.

Class Diagram Element	Java Element
Class	Java Class
Property	Class Field (with setter and getter methods)
Operation	Class Method
Association	Class Field (with setter and getter methods)
Type	Java Equivalent Type

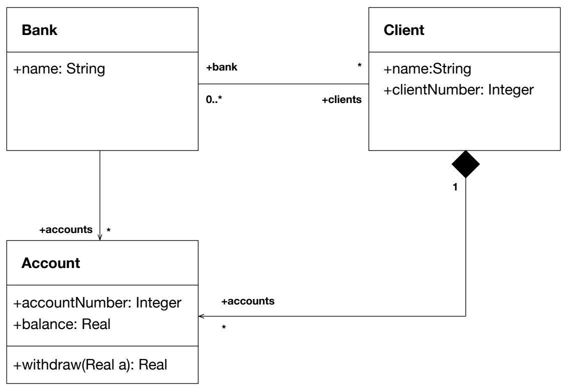


Figure 2.9: Source class diagram (PIM).

Following the transformation rules, the three classes in the diagram will be mapped to three Java classes of the same name. Also, properties are mapped to class fields with their respective get and set methods. Figure 2.10 shows the generated Java code for classes and attributes. Since associations in a class diagram do not have a direct counterpart in most programming languages, they are usually mapped to fields in the

```java
public class Bank {

    private String name;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    ...

}

public class Account {

    private Integer accountNumber;

    public Integer getAccountNumber() {
        return accountNumber;
    }
    public void setAccountNumber(Integer accountNumber) {
        this.accountNumber = accountNumber;
    }

    private Double balance;

    public Double getBalance() {
        return balance;
    }
    public void setBalance(Double balance) {
        this.balance = balance;
    }

    ...

}
```

```java
public class Client {

    private String name;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    private Integer clientNumber;

    public Integer getClientNumber() {
        return clientNumber;
    }
    public void setClientNumber(Integer clientNumber) {
        this.clientNumber = clientNumber;
    }

    ...

}
```

Figure 2.10: Code generated for classes and properties.

classes that are linked by the association. Looking at the association between *Bank* and *Client*, the transformation creates a field named *clients* in class *Bank* and a field named *banks* in class *Client*. For uni-directional associations, such as the one between class *Client* and *Account*, only one field for the navigable end is generated. In this specific example, the transformation does not generate code to enforce the multiplicity constraints of an association; it is the developers responsibility to implement them. Figure 2.11 shows the generated Java code for associations. Finally, operations are mapped to empty method declarations on their respective classes, leaving their final implementation to developers. Figure 2.12 shows the generated Java code for the operations in the diagram.

2.4 Symbolic and Dynamic Symbolic Execution

Symbolic execution is a testing technique where instead of executing a program it is symbolically executed for a set of classes of inputs [15]. In the other hand,

```
public class Bank {

    ...

    private List<Client> clients;

    public List<Client> getClients() {
        return clients;
    }
    public void setClients(List<Client> clients) {
        this.clients = clients;
    }

    private List<Account> accounts;

    public List<Account> getAccounts() {
        return accounts;
    }
    public void setAccounts(List<Account> accounts) {
        this.accounts = accounts;
    }

}

public class Account {

    ...

}
```

```
public class Client {

    ...

    private List<Bank> banks;

    public List<Bank> getBanks() {
        return banks;
    }
    public void setBanks(List<Bank> banks) {
        this.banks = banks;
    }

    private List<Account> accounts;

    public List<Account> getAccounts() {
        return accounts;
    }
    public void setAccounts(List<Account> accounts) {
        this.accounts = accounts;
    }

}
```

Figure 2.11: Code generated for associations.

```
public class Bank {

    ...

}


class Account {

    ...

    public Double withdraw(Double a){
        // TODO
        return null;
    }
}
```

```
public class Client {

    ...

}
```

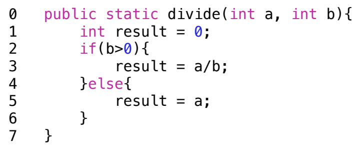
Figure 2.12: Code generated for operations.

dynamic symbolic execution is a combination of concrete and symbolic execution. The following subsections introduce terminology related to dynamic and symbolic execution and an overview of each approach.

2.4.1 Symbolic Execution

During symbolic execution, the execution semantics is changed for symbolic execution by introducing symbolic names that are used to represent the program [15] inputs. This is, each input value is replaced by a symbolic variable, e.g., α_1, and program inputs are eventually assigned to program variables. With the introduction of symbols, the arithmetic expressions used in assignment and *IF* expressions are extended to support them. As a result, the right hand side of an assignment is expressed as a polynomial expression over the symbols introduced, i.e., $\{\alpha_1, \alpha_2, ...\}$. The state of a program execution includes the variables with their values as a symbolic expression, a statement counter, and the path condition (pc) [15]. The path condition is a Boolean expression over the symbolic inputs, e.g., $\alpha_1 + \alpha_2 > 0$. A path condition is "*the accumulator of properties which the inputs must satisfy in order for an execution to follow the particular associated path. Each symbolic execution begins with pc initialized to true. As assumptions about the inputs are made, in order to choose between alternative paths through the program as presented by IF statements, those assumptions are added (conjoined) to pc*" [15].

Unlike a concrete execution, the result of a symbolic execution is equivalent to a set of test cases, where each symbolic value represents a class of concrete values. In symbolic execution, "*the class of inputs characterized by each symbolic execution is determined by the dependence of the program's control flow on its inputs*" [15]. During symbolic execution, all possible execution paths are explored, except on the presence of unbounded loops, where the number of iterations is bounded. Table 2.2 shows an example of symbolic execution of method *divide()* in Figure 2.13. At each statement, the table shows the current symbolic value for each of the variables involved in the method's execution. Input parameters are immediately assigned a symbolic value, e.g., a_1, b_1 respectively, while for other variables their value is unknown until an assignment occurs. The table shows two possible execution paths as *cases*, each case shows a path condition that describes a particular symbolic execution. In this example, one where the evaluation of the *if* statement is *true*, and one where the evaluation is *false*. For example, the path condition $b_1 > 0$ represents all the test case values where parameter b has a value greater than 0.



```
0   public static divide(int a, int b){
1       int result = 0;
2       if(b>0){
3           result = a/b;
4       }else{
5           result = a;
6       }
7   }
```

Figure 2.13: Example method for symbolic execution.

Table 2.2: Symbolic execution example.

Statement	a	b	result	PC
1	a_1	b_1	?	true
2	-	-	0	-
case ¬ (b > 0)				
3	-	-	-	$\neg (b_1 > 0) \wedge$ true
6	-	-	a_1	-
8	-	-	-	-
case (b > 0)				
3	-	-	a_1 / b_1	$(b_1 > 0) \wedge$ true
4	-	-	-	-
8	-	-	-	-

One of the key aspects of a symbolic execution technique is the ability to determine the decidability of the generated path conditions, i.e., whether a particular execution path is feasible or not. In some instances, when working with complex path conditions, this may introduce a decidability problem, when current SAT solvers are not able to successfully evaluate the satisfiability of an expression. Another problem of symbolic execution arises with the introduction of loops that could lead to infinite unfolding. Dynamic symbolic execution tries to alleviate these issues while taking advantage of the symbolic execution.

2.4.2 Dynamic Symbolic Execution

Dynamic symbolic execution is a combination of a concrete execution and symbolic execution. Dynamic symbolic execution starts by executing a program with a specific input while also performing symbolic execution. The constraints or path conditions generated are then used by a constraint solver to decide the next execution, usually forcing the program to follow a different execution path by flipping a sub-expression in the path condition and generating the appropriate input. One of the advantages over symbolic execution is that in the presence of complex symbolic expressions where a constraint solver is unable to produce a result, the concrete values and observed execution can be used to simplify the expression, keeping the constraints decidable [16]. Table 2.3 shows an example of dynamic symbolic execution, with concrete values *5*, *3*, for *a* and *b* respectively. Unlike symbolic execution, there is only one *case* that corresponds to the actual concrete execution.

Table 2.3: Dynamic symbolic execution example.

Statement	a	b	result	PC
1	a_1	b_1	?	true
2	-	-	0	-
3	-	-	a_1 / b_1	$b_1 > 0 \wedge$ true
4	-	-	-	-
8	-	-	-	-

2.5 Software Verification

Consistency checking between a UML class diagram and its Java implementation can be done either with formal verification or testing-based validation techniques. Software verification can be defined as "*the process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase*" [9]. In a more general sense, verification tries to find out if a program satisfies one or more properties, such as pre or post-conditions or assertions [17]. One approach for software verification is model checking, an approach that employs state-space exploration to exhaustively and automatically check whether a software system satisfies a given specification. This is usually done by exploring all possible program states within a bound using static analysis, while checking that each state satisfies a predefined set of properties [18].

While effective at determining if a software system satisfies a given specification, due to the increasing complexity of software systems, formal verification often does not scale effectively to real world applications due to state-space explosion, i.e., exhaustively generating all possible states in large systems becomes impractical. One way to alleviate the state-space explosion problem is to perform model based verification with bounded space exploration, such as bounding the depth of the search [19, 20]. A bounded approach scales more effectively to real world application, since it effectively limits the state-space that needs to be explored. However, bounded model based verification is not as effective since not all the program states are checked, i.e., bounded model based verification can only show the presence of bugs but not their absence.

2.6 Program Testing

Unlike software verification, program testing is a form of dynamic analysis in which the program is executed to check if the resulting state satisfies a set of requirements.

Central to program testing is the use of test cases. A test case is composed by four elements. The first element is the *test case values*, or some input that allows the program to execute successfully. The second element is a set of expected results, also called *oracles*, which are compared to the resulting state after the execution to determine if it was successful. The third element is a set of *prefix values*, any values necessary to initialize the system in to a state where it can receive the test values. And fourth, *postfix values*, needed to complete the execution and evaluation of the software under test [9].

Program testing offers some advantages over verification. One of them is the ability to instrument a program, allowing for a more flexible test case generation process aimed at achieving different types of coverage criteria. Program instrumentation is a code modification activity in which extra code is added to the original program to monitor its execution or modify its behavior as needed. In the context of software testing, instrumentation is typically used to monitor the program execution to determine the execution path (which statements are executed) of a particular test case. Execution paths can be used by a testing procedure to determine how to generate further test cases using coverage criteria. In the scope of program testing, coverage criteria is used to define program execution coverage goals, such as branch coverage, where different control statement branches must be covered by the execution of multiple test cases [9].

Another advantage of program testing is that requirements can be more easily designed. In some instances requirements can be automatically generated from the program or program's model, e.g., class diagram, simplifying the testing activities considerably. However, unlike model based verification techniques, program testing can only show the presence of bugs, but not their absence, since in most scenarios, program testing can only cover a subset of all possible test cases. This is, when testing non-trivial programs it is unfeasible to exhaustively generate all possible test cases.

A trade-off when the complexity of some programs turn model based verification an unfeasible approach.

In the context of consistency checking, traditional program testing has also some limitations. One of the limitations is that traditional testing techniques assume that the execution of a faulty statement can expose a software fault, and rely on different coverage criteria to cover as much statements as possible. Unfortunately, some errors cannot be exposed based on this assumption. If there is not an asserting statement, the error may not be exposed, even after executing the faulty statement. Consider the code in Figure 2.14, introduced by Ammann et al. [9].

```
0   public static int numZero (int[] x) {
1       // Effects: if x = null throw NullPointerException
2       //    else return the number of occurrences of 0 on x
3       int count = 0;
4       for(int i=1; i < x.length; i++){
5           if(x[i]==0){
6               count++;
7           }
8       }
9       return count;
10  }
```

Figure 2.14: Sample code for traditional testing.

The method calculates the number of zeros in array *x*. In this example, the programmer forgot to check the first element of the array in the *for* loop. The statement at line four, "*for(int i=1; ...)*", should be, "*for(int i=0; ...)*". If the method is executed with the input "*[2, 7, 0]*", no error is exposed when the faulty statement is executed. A coverage criteria approach is not sufficient to uncover the error.

Another limitation of traditional program testing is that most techniques rely on flipping conditional branches during the execution of the program to reach different statements and achieve the desired coverage criteria. However, in MDE, some forward engineering tools translate a class diagram to a program that has auxiliary fields that if flipped can cause undesired behavior. For instance, in the Eclipse Modeling Framework (EMF) [21], the attribute *eContainerFeatureID* is introduced as an auxiliary field. This attribute is an integer used to identify a container and specify whether it is a navigable feature or not by assigning a positive (navigable) or negative value.

If the value of *eContainerFeatureID* is altered to cover a different execution path, a false positive may be reported.

2.7 Model Based Testing

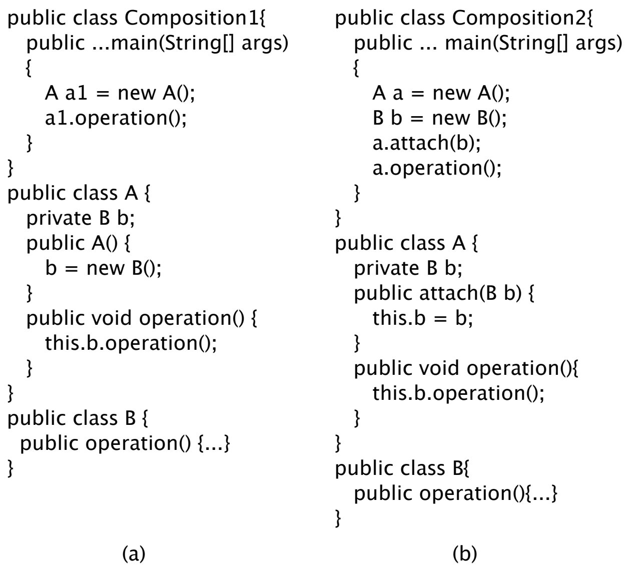
Most model-based testing techniques consider the generation of test cases from behavioral models [22, 23, 24, 25, 26, 27, 28, 29] such as activity or sequence diagrams. As an example, Khandai et al. proposed a technique for generating test cases for concurrent systems using sequence diagrams [22]. In their approach, a UML sequence diagram, showing all possible interactions or sequence of message exchanges between objects, is used to automatically generate a set of test cases that represent the interactions defined in the model. In a similar way, the work by Cartaxo et al. uses UML sequence diagrams with labeled transition systems for test case generation [29]. An approach by Anbunathan et al. [30] makes use of some of the static information contained in the system's class diagrams, such as class names, and associations, but it ultimately relies in the use of state diagrams for the test case generation.

In general, existing model-based testing techniques can be classified into two categories. In state-based approaches the system behavior is described by state machines. Abdurazik et al. proposed an approach to generate test cases based on UML collaboration diagrams to perform dynamic and static testing [31]. And the second category uses scenario-based descriptions of interactions between different system entities. Roychoudhury et al. proposed a new notation, called symbolic message sequence charts, that generates test cases for process classes [32]. A verification tool for Java [33] considers OCL and JML as assertion languages to specify pre or post-conditions on a Java program, and it applies a verification approach to determine if the implementation is consistent with the specification. As far as we know, no prior work has been proposed to support consistency based testing of programs against UML class diagrams.

Another area of interest is consistency checking between different types of models or specification attributes [34, 35, 36, 37], but they do not support consistency between a model and its final implementation. As an example, the work by Sabetzadeh et al. introduces a technique in which two sets of models are merged and then checked against a set of requirements to determine their consistency [35]. In other instances, these approaches utilize a set of transformations or model merging techniques to match elements between different types of models [35, 37]. As an example, the work by Liu et al. [38] introduces an MDA test case generation approach that employs model to model transformations. Their approach relies in transformation rules between structural features, and does not consider behavioral specification such as OCL post-condition.

Reverse engineering techniques have also been applied to check some specific UML properties, such as class diagram association properties. Work by Gueheneuc et al. [39] and Milanova et al. [40] recovers UML composition from a program based on the non-accessibility property. However, this property is not required by the UML specification. Their approach focuses in four properties: exclusivity, invocation site, lifetime, and multiplicity. In terms of the lifetime property, their approach applies the object ownership model which doesn't fully comply with the UML composition.

The ownership model enforces the non-accessibility property, i.e., a part object is not allowed to be used outside of its owner object. According to the non-accessibility property, the implementation in Figure 2.15 (b) is not a composition, since instances of class *B* (owned class) can be accessed from the outside. However, the UML specification allows a third-party object to access an owned object directly. Therefore, the definition of composition based on the exclusivity property is not consistent with the UML specification. In Figure 2.16 we can see a fragment of the UML metamodel where the metaclass *Association* has an association to the owned metaclass *Property*. This means that an instance of metaclass *Association* can access instances of owned



```
public class Composition1{
  public ...main(String[] args)
  {
    A a1 = new A();
    a1.operation();
  }
}
public class A {
  private B b;
  public A() {
    b = new B();
  }
  public void operation() {
    this.b.operation();
  }
}
public class B {
 public operation() {...}
}
```

(a)

```
public class Composition2{
  public ... main(String[] args)
  {
    A a = new A();
    B b = new B();
    a.attach(b);
    a.operation();
  }
}
public class A {
  private B b;
  public attach(B b) {
    this.b = b;
  }
  public void operation(){
    this.b.operation();
  }
}
public class B{
  public operation(){...}
}
```

(b)

Figure 2.15: Two composition implementations.

metaclass *Property* via *memberEnd*. As a result, the application of a composition property that follows the ownership model would require considerable adaptations to existing modeling tools.

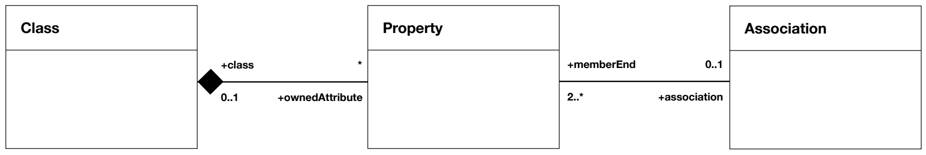


Figure 2.16: UML meta-model fragment.

Barbier et al. [41] introduced a formal specification of the whole-part relationship which includes the composite property. The specification uses the Object Constraint Language (OCL) and it is based in the addition of new metaclasses to the UML specification. Unfortunately, the introduction of new metaclasses renders existing modeling tools incompatible with their approach. As a result, the use of their specification, although effective in precisely defining the whole-part relationship, becomes too restrictive given that the required adaptations on existing tools are in most cases not feasible. The work presented by Milicev [42] in the other hand introduces a formal specification of the UML association properties using the *Z* notation specification language, but their specification doesn't include the composite property.

CHAPTER III

BOUNDED EXHAUSTIVE TESTING

In object oriented programming, the basic building blocks are objects that collaborate with each other to achieve a high level behavior [11]. This collaboration is achieved through interactions, such as method invocations, that establish relationships between objects. These relationships allow them to contribute to the overall behavior of the system; an object in isolation would be of no value. In this scenario, the desired behavior of a system can only be achieved if the objects interact as expected.

In UML, an association is used to identify the semantic relationship between two classes, i.e., it defines the types of interactions that are possible between objects that are instances of those two classes. An incorrect implementation of UML associations would result in objects interacting in unexpected ways, and as mentioned previously, in object oriented programming, the correct functioning of a program is highly dependent on objects interacting as expected. Because of this, maintaining the consistency between an implementation and its design model is an important challenge. To illustrate our approach we will use the UML composition as an example.

UML composition, as a subtype of a whole/part relationship, can be defined in terms of two properties; *lifetime* and *shareability*. The lifetime property indicates that an element is composed by one or more parts, and such element is responsible of the lifetime of its parts, i.e., if the element is deleted, all of its parts are deleted. For the shareability property, a part element cannot be part of more than one element at a time. In a class diagram, UML composition is represented as a solid/filled black diamond. Figure 3.1 shows an example of a UML composition relationship between class *Company*, known as *owner* class, and class *Division*, known as *owned* class. The relationship indicates that a company is composed by zero or more divisions. If a particular company is deleted, the divisions that compose such company are also deleted (lifetime property). Also, a particular division cannot be part of more than one company at a time (shareability property).

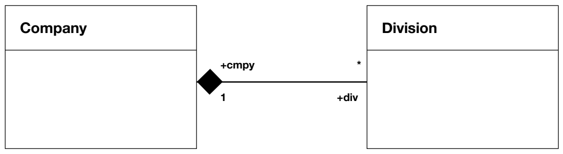


Figure 3.1: UML composition example.

A program that implements the UML composition but fails to satisfy the lifetime property can cause serious consequences at runtime, such as trying to access an object that no longer exists or memory leak problems when memory from unused objects cannot be correctly reclaimed by garbage collectors. Also, divergences between a design model and its implementation can lead to a failure to satisfy or effectively trace requirements during the development process.

The example in Figure 3.2 shows a scenario with an incorrect composition implementation. In this example, the *destroy()* method fails due to the condition in the *for* statement in line 12 "*i > d.getClients.size() - 1*", where the reference to the instance of class *Division* that is being destroyed is not removed from the last *Client* in the collection. Because of this, the code in the *main()* method causes a *NullPointerException* when trying to invoke the method *getName()* on the instance of class *Company* that was previously destroyed.

```
public class Company {
  private List div;
  private String name;
  public void setDiv(Division d) { ... }
  public Division getDiv() { ... }
  public String getName() { ... }
  public void setName() { ... }
  public void destroy() {
  Iterator divisionIt = div.iterator();
  while(divisionIt.hasNext()) {
      Division d = (Division) divisionIt.next();
      for(int i=0; i < d.getClients().size() - 1; i++) {
          Client c = d.getClients.get(i);
          c.setDivisions(null);
      }
      d.setCmpy(null);
   }
   setDiv(null);
  }
}

public class Division {
  private Company cmpy;
  private List clients;
  public void setCmpy(Company c) { ... }
  public Client getCmpy() { ... }
  public void setClients(Client c) { ... }
  public Client getClients() { ... }
}

public class Client {
  private List divisions;
  public void setDivisions(Division d) { ... }
  public List getDivisions() { ... }
}

public class Test {
  public static void main(String args[]) {
    Company c = new Company();
    Division d = new Division();
    c.setDiv(d);
    Client cl = new Client();
    cl.setDivisions(d);
    . . .
    c.destroy();
    . . .
    for(int i=0; i<cl.getDivisions().size; i++) {
      Division d1 = cl.getDivisions().get(i);
      System.out.println(d1.getCmpy().getName());
      // NullPointerException
    }
  }
}
```

Figure 3.2: Exception caused by an incorrect composition implementation.

3.1 Overview

To detect inconsistencies between a design model and its implementation our approach provides a method to test whether the implementation enforces a set of OCL constraints derived from the design model in terms of the UML association properties. Our approach overview is shown in Figure 3.3 and it is explained in the following sections.

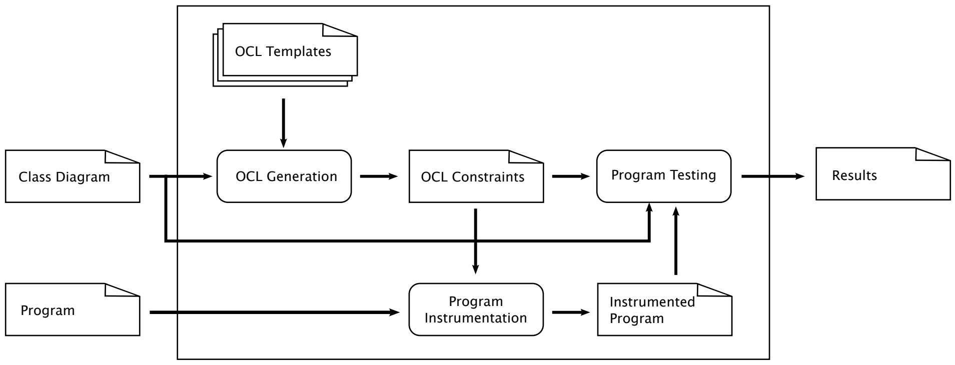


Figure 3.3: Approach Overview.

3.1.1 Input

Our approach takes as input a class diagram and a Java program that must be an implementation of the class diagram. As an example, the class diagram in Figure 3.1 and the implementation in Figure 3.2 serve as input. The implementation can be a combination of code automatically generated by forward engineering tools and code manually added by developers to implement any features not supported by the tool, such as composition associations.

3.1.2 General Steps

As a first step, using information from the class diagram, we generate a set of OCL constraints for each of the UML association properties, the OCL constraints are generated using a set of templates that are introduced in Appendix A. Next, the program is instrumented to gather some information, such as fields being accessed during execution. This information will be used during the testing process, explained in Section 3.1.4. The instrumentation is guided by the generated OCL constraints. The instrumented program, class diagram, and the generated OCL constraints are then used during the program testing phase. During the program testing phase, we generate a set of instances, used as test cases, to which we apply the previously generated OCL constraints to try to detect possible violations of any of the UML association properties. The process is explained in detail in Section 3.1.4 using the composition property as an example.

3.1.3 Formalization of UML Associations

The Unified Modeling Language (UML) [7], a popular modeling language used in software development projects, provides a set of diagrams that can be used to model the static and dynamic aspects of a system. Among these diagrams, class diagrams are used to define the static structure of the system. Some of the concepts introduced in a class diagram, like inheritance, can be easily mapped to most object oriented programming languages. Unfortunately, other concepts, such as associations, have no direct counterpart in the syntax and semantics for programming languages. As a result, while there exist many forward engineering tools which help generate code from a class diagram they often leave some implementation to the users, or in some cases they implement them in a way that does not fully follow the semantics defined by UML.

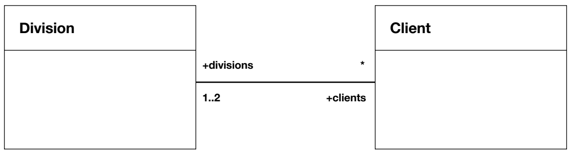


Figure 3.4: A UML *Class* Diagram.

A UML class diagram includes a set of classes and the relationships among them.

Figure 3.4 shows a class diagram with two classes: *Division*, and *Client*. The relationship between classes is denoted by associations, i.e., a line connecting two classes, and each association is marked with a set of properties at each end (member end) that further defines the type of relationship. Thus, an association in a UML class diagram is used to specify a semantic relationship between classes, i.e., the relationship meaning and possible restrictions. An association must have at least two ends, each end connected to a class [7]. For example, the association between class *Division* and *Client* in Figure 3.4 shows that a division can be related to any number of clients, indicated by the "***" next to class *Client*, and also shows that a client can be related to 1 or 2 divisions, indicated by "*1..2*" next to class *Division*.

An instance of a class diagram can be represented by a UML object diagram. An example of an instance of the class diagram in Figure 3.4 is show in Figure 3.5 (b). The figure also includes the class diagram to show the instantiation relationship. The object diagram shows an instance of class *Division* named *motors* related to an instance of class *Client* name *ford*. In an object diagram, an instance of an association is called link and is represented by a single line. A link in an object diagram, unlike an association, does not include any properties.

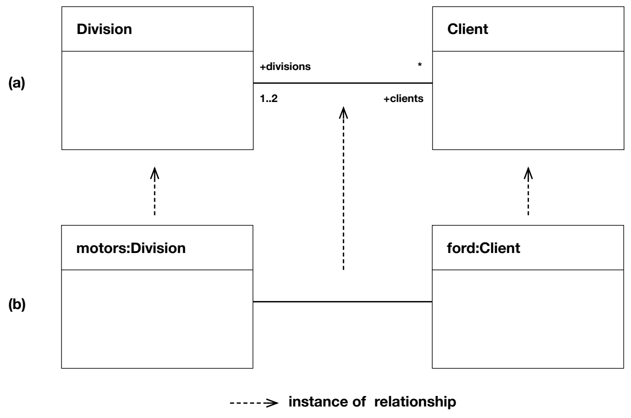


Figure 3.5: Instantiation of a class diagram.

For an object diagram to be considered valid, it must follow the restrictions indicated in the class diagram it instantiates. The object diagram in Figure 3.5 (b) is valid since it does not violate any of the restrictions in the class diagram, e.g., the maximum number of relationships between the instances of classes *Division* and *Client*. The example in Figure 3.6 however, shows an invalid object diagram since the instance of class *Client* has a link to more than two instances of class *Division*.

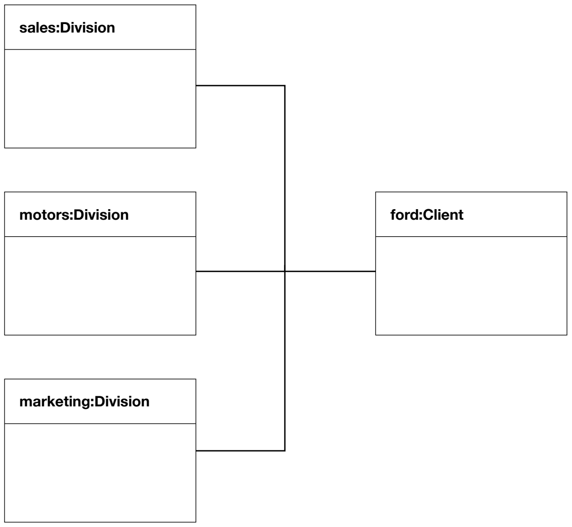


Figure 3.6: Invalid instance of the class diagram in Figure 3.4.

In a similar way, the UML specification provides a set of diagrams that describe the structure and semantics of a valid class diagram, i.e., a class diagram must be a valid instance a UML specification diagrams. Figure 3.7 shows a fragment of the UML specification (a) and an instance (b). A class in a UML specification diagram is called a metaclass. In this example, class *Division* is an instance of metaclass *Class* and the property *isActive* is an instance of the metaclass *Property*. The diagram in Figure 3.7 (a) indicates that an instance of metaclass *Class* can own 0 or more instances of metaclass *Property*.

The diagrams previously introduced are part of the metadata architecture. The metadata architecture includes four layers, three of which we already used in our previous examples; the information layer (M0) represented by an object diagram, the model layer (M1) represented by a class diagram, and the metamodel layer (M2), also represented as a class diagram that describes the structure and semantics of a valid

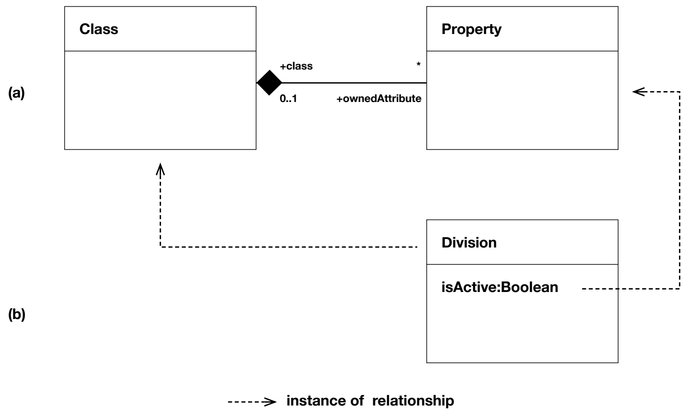


Figure 3.7: UML specification fragment (a) and instance (b).

class diagram at M1 layer. The last layer is the meta-metamodel layer (M3). The information layer includes the data being described, the model layer includes metadata that describes the information layer, the metamodel layer defines the structure and semantics of the metadata and the meta-metamodel layer includes the description and semantics of meta-metadata. The same way an object diagram (M0) represents an instance of a class diagram (M1), a class diagram represents an instance of the metamodel layer (M2). And a diagram at M2 layer represents an instance of the meta-metamodel layer (M3).

The UML language is defined at the metamodel layer, or M2 layer. The specification includes a set of diagrams at the M2 level that describe the structure and semantics of the M1 layer. The fragment of the UML specification shown in Figure 3.8(M2), includes the metaclasses needed to create an association between two classes at the M1 level. The diagram includes three metaclasses: *Class*, *Property*, and *Association*. The diagram shows that an association must have at least two member ends, denoted by the "*2..**" next to metaclass *Property*, where each member end is a property, and it also shows the relationship between a class and a property, i.e., the property is a class attribute. Figure 3.8(M1) shows an instance of the class diagram at M2 level, where classes *Division* and *Client* are instances of metaclass *Class*. The attributes *divisions* and *clients* are instances of metaclass *Property*, which are referenced by their respective class via ownedAttribue. The association (the line linking classes *Division* and *Client*), is an instance of metaclass *Association* and it has a reference to the attributes *b* and *c* via *memberEnd*.

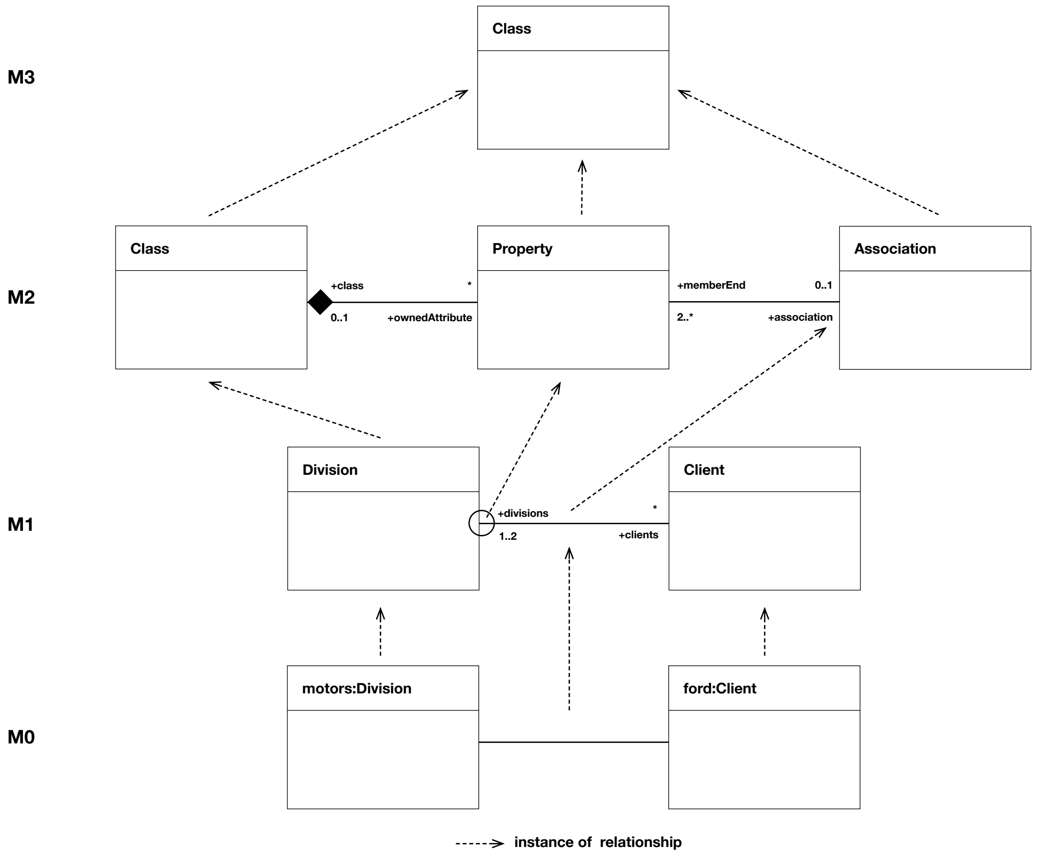


Figure 3.8: Four layer metadata architecture

For an M1 instance, or class diagram, to be valid, it must follow the restrictions of the class diagram at the M2 layer. An example of an invalid instance is shown in Figure 3.9, where the instance of metaclass *Association* has a reference to one member end *b*, an instance of metaclass *Property*. This is a violation since the multiplicity for the *memberEnd* is "*2..**", meaning that an association must have a reference to at least two instances of metaclass *Property*. The relationship between the different layers in the context of the Unified Modeling Language is shown in Figure 3.8.

The UML specification also includes a set of constraints that are applied to the diagrams at M2 level. Some constraints are defined using the Object Constraint

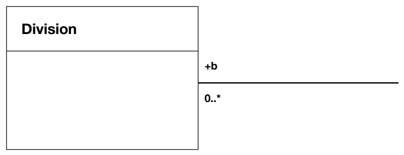


Figure 3.9: Invalid class diagram.

Language (OCL). Having a constraint defined at M2 level means that an instance of the model in the next layer, i.e. M1 level, must satisfy the restriction. As an example, the following OCL constraint is given for the class diagram in Figure 3.8(M2):

```
context Association
inv: self.memberEnd->exists(aggregation <> Aggregation::none)
     implies self.memberEnd->size() = 2
```

The constraint states that if a member end exists in the association whose aggregation type is different than none, e.g., aggregation type is composite; the association must have exactly two member ends. The operation *size()* returns the number of elements in the collection *memberEnd*. For example, the class diagram in Figure 3.10(a) shows a valid instance of an association with three member ends, i.e., *i*, *c*, and *it*. While the class diagram in Figure 3.10(b) shows an invalid instance, since one of the member ends is marked as *Aggregation::Composite* (black diamond), and *self.memberEnd->size()=3*.

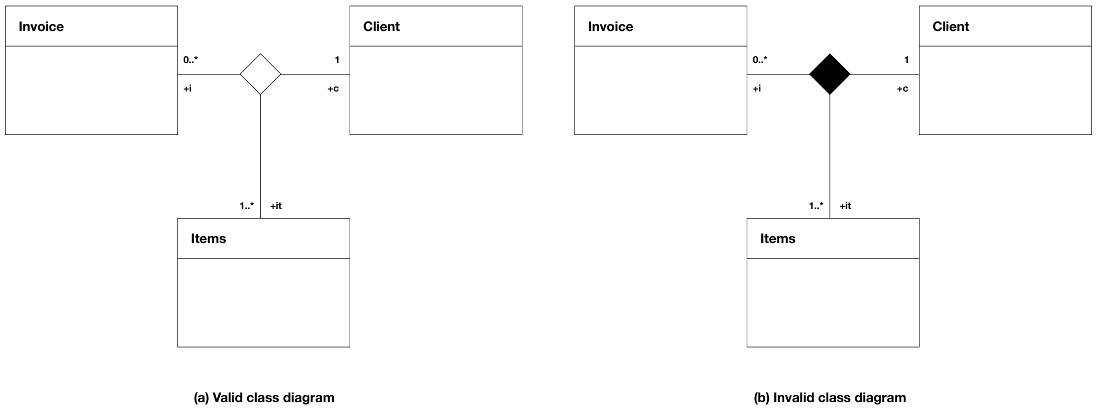


Figure 3.10: Invalid class diagram.

Constraints defined with OCL or other formal languages can be used to automatically validate M1 level instances (M0). Unfortunately, not all constraints are formally defined in the UML specification; some are only described using natural language. For example, the following constraint is found in the specification: "*When an association specializes a second association, every end of the specific association corresponds to an end of the general association, and the specific end reaches the same type or a subtype of the more general end.*" [7] p.40. As a result, validating that an implementation fully adheres to the model specification becomes a challenging task, since it is hard to automatically process and understand natural language.

3.1.4 Our Approach

To detect inconsistencies between a design model and its implementation our approach provides a method to test whether the implementation enforces a set of OCL constraints derived from the design model in terms of the UML association properties. Our approach uses as input a class diagram and a Java implementation and it can be divided in two main modules: OCL generation, and program testing. The class diagram is used by the OCL generation module to generate a set of OCL constraints at M1 level for each of the association properties. The OCL constraints generation is done according to the templates presented in Appendix A. Next the program is instrumented and together with the class diagram and the generated OCL constraints is given as input to the program testing module. The program testing module generates a set of instances (test cases) to which it applies the OCL constraints to try to detect possible violations. To illustrate our approach, in the following section we will concentrate on the UML composition property.

3.1.4.1 Detecting UML Composition Violations

To illustrate our approach to detect violations of the UML composition property, we will consider the program shown in Figure 3.11 that implements the class diagram in Figure 3.12. In this class diagram we have a composition association, where the owner class is class *Company* and the part class is class *Division.* We also have two associations, one between class *Division* and class *Client*, and one between class *Client* and class *Division.* In relation to the composition association, we call class *Client* a third-party class since it has a relationship to the part class. We call instances of an owner class, owner objects, and instances of a part class, part objects. In this example, we omitted the implementation of class *University.* Also, the code has a reference to class *Display*; an auxiliary class that displays information about a company and its relationships.

```
public class Company {
  private List div;
  private String name;
  private Display disp;
  public void setDisp(Display d) { ... }
  public Display getDisp() { ... }
  public void setDiv(Division d) { ... }
  public Division getDiv() { ... }
  public String getName() { ... }
  public void setName() { ... }

  public void destroy() {
  Iterator divisionIt = div.iterator();
  while(divisionIt.hasNext()) {
      Division d = (Division) divisionIt.next();
      for(int i=0; i < d.getClients().size(); i++) {
          Client c = d.getClients.get(i);
          c.setDivisions(null);
      }
      d.setCmpy(null);
   }
   setDiv(null);
   disp.removeObject(this);
   disp = null;
 }
}

public class Division {
  private Company cmpy;
  private List clients;
  public void setCmpy(Company c) { ... }
  public Client getCmpy() { ... }
  public void setClients(Client c) { ... }
  public Client getClients() { ... }
}

public class Client {
  private List divisions;
  private University univ;
  public void setDivisions(Division d) { ... }
  public List getDivisions() { ... }
  public void setUniversity(University u) { ... }
  public University getUniversity() { ... };
}

public class Display {
  private List<Company> cmpy;
  public Display() { ... }
  public void addCmpy(Company c) { ... }
  public void removeCmpy(Company c) { ... }
  public List<Company> getCmpys() { ... }
  public void display(){ ... }
}
```

Figure 3.11: A Java implementation of the class diagram in Figure 3.12.

The execution of a specific sequence of instructions using the code in Figure 3.11 can be used to generate a program state that is an instance (M0) of the class diagram in Figure 3.12 (M1). As an example, the execution of the instructions in Figure 3.13 result in a program state where an instance of class *Company* has a reference to an instance of class *Division.* The resulting program state can be represented as an object diagram as it is shown in Figure 3.14.

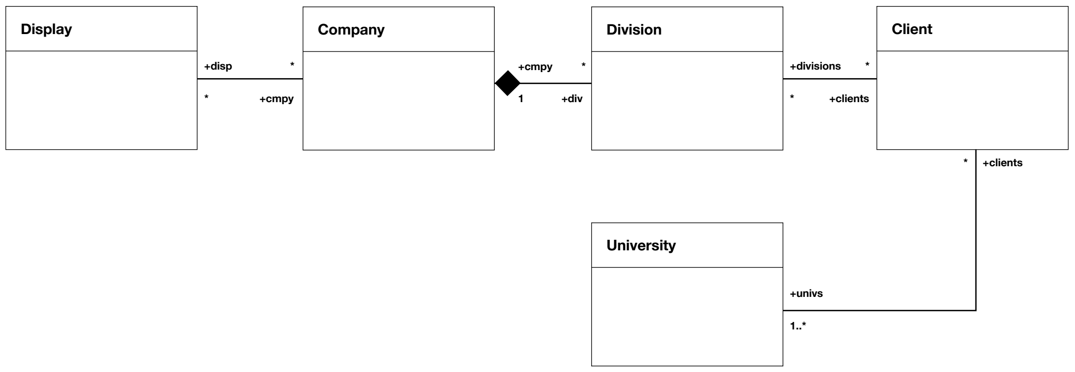


Figure 3.12: Class diagram for a *Company* example.

```
public class Application {
    public static void main(String[] args){
        Company c = new Company();
        Division d = new Division();
        c.setDivision(d);
        Display disp = new Display();
        disp.addObject(c);
    }
}
```

Figure 3.13: Code execution from Figure 3.11.

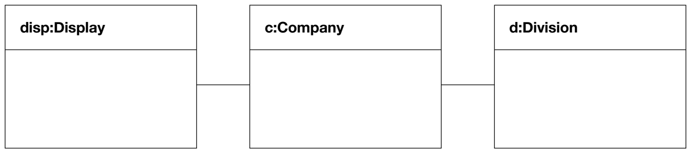


Figure 3.14: Object diagram after the execution of the program in Figure 3.13

On the first stage of our approach we need to generate a set of OCL constraints at M1 level for the composition property using as input the class diagram in Figure 3.12. These constraints will be used by the program testing module. The constraints are defined as a post-condition of the method that implements the lifetime property in the owner class. For this example, we generate the following constraint:

```
context Company::destroy()
post:  self.div->forAll(x | Client.allInstances(y |
                                  y.divisions->excludes(x))
```

In the code in Figure 3.11, the method we are interested on is *destroy()*. As we mentioned in previous sections, the lifetime property requires that if a composite is removed its parts should also be removed, i.e. when an owner object does not exist; then neither does its part objects. An owner object and its part objects are said to "*not exist*" when all external links to the objects are removed, making them unreachable by any other object. Figure 3.15 shows an example of the destruction of an owner object and its part object.

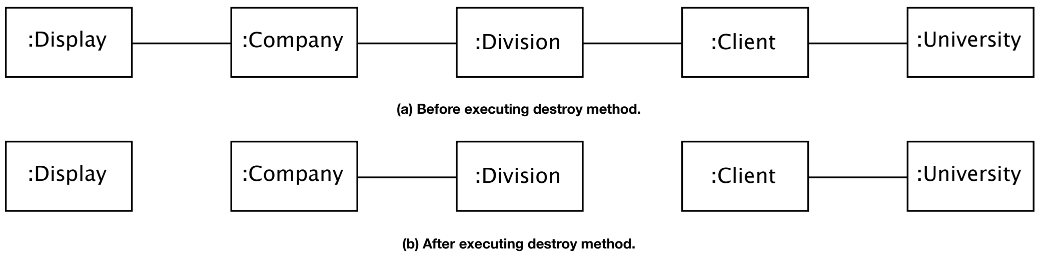


Figure 3.15: The destruction of an owner and part objects.

To check the correctness of the method *destroy()*, we need to execute it on all possible test inputs up to a bounded size. Each test input is an instance (object diagram) of the class diagram in Figure 3.12 and can be generated by executing a set of instructions similar to the ones in Figure 3.13. In Figure 3.16 we have three object diagrams where *cmpy1* is an instance of class *Company*, *div1* is an instance of class *Division*, *cli1* is an instance of class *Client*, and *univ1* and *univ2* are instances of class *University*. Also, the diagrams above and below the arrows denote the pre-states and post-states before and after executing the method *destroy()*.

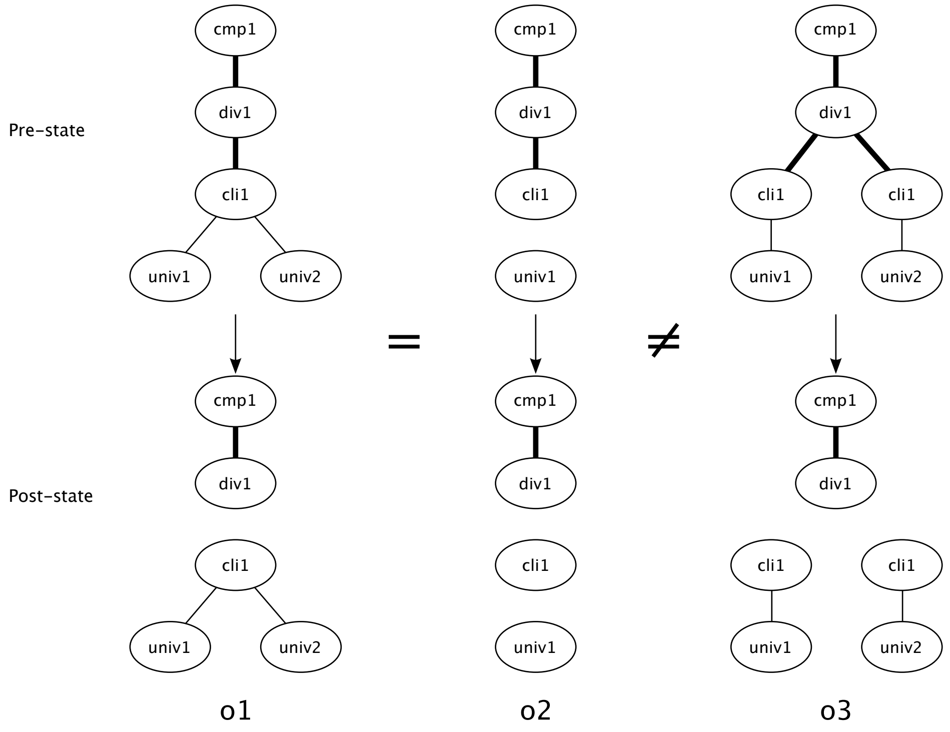


Figure 3.16: Object diagrams before and after calling *destroy()*.

After the execution of the method *destroy()*, we validate the lifetime property by applying the previously generated OCL constraint to the resulting object diagram. The constraint is satisfied if all the links to the owned objects are removed. In this example, in the post-state of the object diagram *o1* in Figure 3.16, the link between *div1* and *cli1* was removed. Therefore, the lifetime property is satisfied.

During the execution of the method *destroy()*, we monitor its execution to detect the fields read and use this information to prune the test input space. Figure 3.16 shows a set of three object diagrams before the execution of method *destroy()* (pre-state) and after the execution of method *destroy()* (post-state); the highlighted links correspond to the fields we detect being read during its execution. After validating the lifetime property, we can prune similar object diagrams where the fields read have the same values. The logic behind this decision is based on the fact that fields that are not being read by the method *destroy()* do not affect its execution. Therefore, as long as the fields read have the same values, the result for the lifetime property will be the same. In this example, the object diagrams *o1* and *o2* are considered similar, unlike the object diagram *o3* where the value of one of the fields read is different.

Based on the implementation of the method *destroy()* in Figure 3.11, we can detect that only the links between instances of class *Company* and class *Division* are relevant to the lifetime property. Therefore, it is sufficient to check every possible object diagram with different number of links between instances of class *Company* and class *Division*. This approach greatly reduces the size of the test input space. Without pruning similar object diagrams, the number of object diagrams checked is exponential in the bounds of the number of objects instantiated for each of the classes in Figure 3.12.

3.1.4.2 Program Testing Implementation

As mentioned in the previous section, to effectively determine the absence of UML association properties violations in a program, we use the bounded exhaustive testing technique. Following this technique, we generate all possible test inputs whose size is within a certain bound, and for this, it must be provided with a domain value for each of the variables.

A domain value indicates the maximum number of objects instantiated for each of the classes in the test case. A single combination of these objects represents a test input or object diagram, and all possible combinations consist of a large number of test inputs that we call test input space. Among all possible test inputs, some are not valid. To generate only valid test inputs, we consider the multiplicity and navigability information from the class diagram (when testing properties other than multiplicity and navigability).

When testing composition, our tool calls the method implementing the lifetime property on one of the valid test inputs. After its execution, we check the lifetime

```
for each composition property in the class diagram
od = use a solver to generate a valid test input
while(od does exist)
    call method under test on od (i.e., destroy)
    apply OCL constraint on post-state
    use the solver to prune similar test inputs
    od = use solver to generate the next valid test input
```

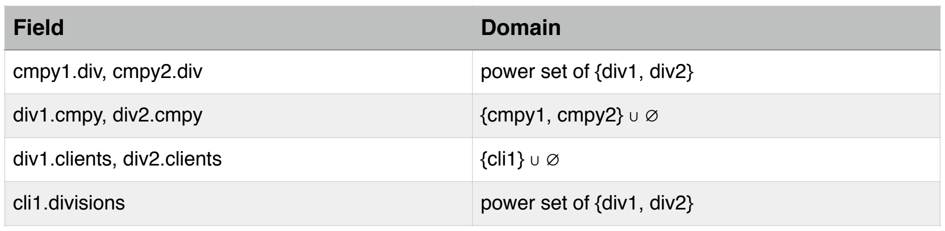
Listing III.1: Algorithm to check UML composition property.

property using the generated OCL constraints and prune away similar test inputs from the test input space. We repeat the process until there are no more valid test inputs.

3.1.4.3 Test Input Space Representation

To generate the test input space we use a *finitization* to specify the maximum number of objects for each for the classes in the test case. For the composition property, the finitization includes values for the owner, owned, and third-party classes. For example, for the finitization (2,2,1) for the class diagram in Figure 3.12, the test input space includes all object diagrams with up to two instances of classes *Company*, up to two instances of class *Division*, and one instance of class *Client*. Using this finitization, our tool generates the domain values for each of the fields needed to create an object diagram. Table 3.1 shows the fields and domain values for finitization (2,2,1).

Table 3.1: Test input space (2,2,1) for the class diagram in Figure 3.12.



Field	Domain
cmpy1.div, cmpy2.div	power set of {div1, div2}
div1.cmpy, div2.cmpy	{cmpy1, cmpy2} ∪ ∅
div1.clients, div2.clients	{cli1} ∪ ∅
cli1.divisions	power set of {div1, div2}

To generate a test input, our tool uses the SAT solver SAT4J [44]. SAT4J uses a vector structure where each element is a Boolean value. Therefore, we need to translate the finitization information to a vector form. For finitization (2,2,1) shown in Table 3.1, our tool generates a twelve bit vector. The first two bits, (N0 and N1) represent whether objects *div1* and *div2* are referenced by *cmpy1*. If N0 is true, it means that *div1* is referenced by *cmpy1*, i.e., object *cmpy1* has a link to object *div1*, and a false value means that there is no reference. The bits N2 and N3 represent whether objects *div1* and *div2* are referenced by *cmpy2*. Bits N4 and N5 represent whether objects *cmpy1* and *cmpy2* are referenced by *div1*. The bits N6 and N7 denote whether objects *cmpy1* and *cmpy2* are referenced by *div2*. Bits N8 and N9 represents whether object *cli1* is referenced by *div1* and *div2* respectively. And finally, bits N10 and N11 specify whether objects *div1* and *div2* are referenced by *cli1*. Also, since the multiplicity is greater than 0, our tool includes an instance of class *University*, *univ1*, adding bits N12 that specifies whether object *cli1* has a reference to *univ1*, and N13 to specify whether *univ1* has a reference to *cli1*.

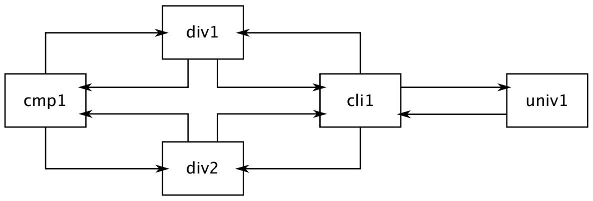


Figure 3.17: An object diagram.

The following is the vector for the object diagram in Figure 3.17. This vector is then converted to an object diagram that is used as a test input.

T	T	F	F	T	F	T	F	T	T	T	T	T	T

3.1.4.4 Pruning Invalid Test Inputs

To generate a valid test input, SAT4J needs a set of formulas that specify valid assignments to the vector structure. These formulas reflect the different domain values in our test case, as well as the constraints obtained from the class diagram. Also, we dynamically apply new formulas to the solver after the execution of the method under test to prune away similar test inputs.

We consider two constraints from a class diagram: multiplicity and navigability. The multiplicity constraint specifies a lower and upper bound on the number of instances of a class on an association end. In the class diagram in Figure 3.12, the multiplicity for the class *Company's* end indicates that an instance of class *Division* can have a reference to one and only one instance of class *Company*. If an instance of class *Division* has a reference to more than one instance of class *Company*, the object diagram is considered invalid. The second constraint, navigability, specifies whether an object has a reference to another object. The navigability is related to the type of association between two classes in a class diagram. If the association is bidirectional, instances of the classes related to that association must reference each other. If it is unidirectional, only one object will reference the second object. In the class diagram in Figure 3.12, the association between classes *Division* and *Client* is bidirectional. This means that an instance of class *Division* must reference an instance of class *Client* and that same instance of class *Client* must reference the same instance of class *Division*.

To prune invalid test inputs with respect to the multiplicity constraint, we apply a set of formulas. For example, considering the multiplicity constraint for the class *Company's* end in Figure 3.12, the following is an invalid vector:

-	-	-	-	T	T	-	-	-	-	-	-	-	-

In this vector, the assignment to bits N4 and N5 is true. This means that object *div1* has references to objects *cmpy1* and *cmpy2*, which violates the multiplicity constraint. We use the formula $\neg(N4 \wedge N5)$ to eliminate all the vectors where the multiplicity constraint is not satisfied by the object *div1*. With this single formula, we eliminate 212 invalid object diagrams. A similar formula is used to eliminate invalid vectors where the multiplicity constraint is not satisfied by the object *div2*.

Also, to eliminate invalid vectors with respect to the navigability constraint, we use a similar approach. For example, considering the bidirectional association between classes *Division* and *Client* in Figure 3.12, the following is an invalid vector:

-	-	-	-	-	-	-	-	-	T	-	T	-	-	-

In this vector, the assignment to bits N8 and N10 is true. This assignment means that object *div2* has a reference to object *cli1* while object *cli1* has a reference to object *div1*. The bidirectional constraint is not satisfied. We can use the formula $N8 \iff N11$ to enforce the navigability for the association between classes *Client* and *Division*.

3.1.4.5 Monitoring the Method under Test

After pruning invalid test inputs, we further prune the test input space by using runtime information. Specifically, we use two techniques: the detection of *do not care fields* and the pruning of isomorphic test inputs (object diagrams).

3.1.4.6 Do Not Care Fields

During the execution of the method under test, we monitor the fields derived from the class diagram that are being read in order to identify do not care fields. A field is considered a "*do not care*" field if it is not read during the method's execution. The fields are called do not care fields because they are not relevant to the execution of the method, i.e. no matter what values the do not care fields have, the result of the lifetime property after the execution of the method is the same.

To illustrate how we use the do not care fields to prune the test input space, let's consider the following example. After calling the method *destroy()* in Figure 3.11 on the object diagram in Figure 3.17, we detect that only three fields are read: field div in class *Company*, field clients in class *Division*, and field divisions in class *Client*.

In our vector representation, these fields correspond to the bits N0, N1, N4, N6, N8, N9, N10, N11, N12, and N13. Among those bits, N12 and N13 are identified as do not care fields. The following vector shows the bits corresponding to the do not care fields marked with a dash.

T	T	F	F	T	F	T	F	T	T	T	T	-	-

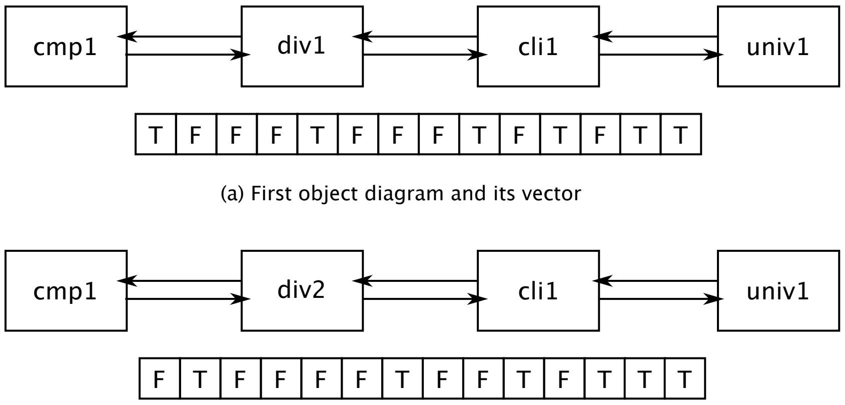
After identifying the do not care fields, we prune the test input space by applying a new formula to SAT4J. This formula eliminates the set of vectors that only differ on the values for the do not care fields. For the previous example, the formula $\neg(N0 \wedge N1 \wedge N2 \wedge N3 \wedge N4 \wedge N5 \wedge N6 \wedge N7 \wedge N8 \wedge N9 \wedge N10 \wedge N11)$ is added to the solver.

3.1.4.7 Pruning Isomorphic Object Diagrams

After eliminating similar states by identifying do not care fields, we further reduce the test input space by pruning isomorphic object diagrams (test inputs). Two object diagrams are said to be isomorphic when they are equivalent in terms of the number of objects and number of links in the diagram. In other words, the number of objects of each class is the same, as well as the number of links between them. As an example, the object diagrams in Figure 3.18 are equivalent, differing only by the name of the object of type *Division*, *div1* and *div2*.

Executing the method under test and checking the lifetime property in two isomorphic test inputs generates the same result. For example, considering the method destroy() in Figure 3.11, its execution is only affected by the number of objects in the fields corresponding to the associations in the class diagram, namely div, clients, and divisions. As long as the number of objects is the same, the execution of the method will be the same.

To prune the isomorphic test inputs, we generate the formula $\neg(x.div = y \wedge y.clients = cli1)$ where $x \in \{cmpy1, cmpy2\}$ and $y \in \{div1, div2\}$. This formula



(a) First object diagram and its vector

(b) Second object diagram and its vector

Figure 3.18: Two isomorphic object diagrams.

prunes away three isomorphic object diagrams. Pruning isomorphic inputs states can greatly reduce the size of the test input space without reducing the fault detection accuracy of our approach.

3.1.5 Small Scope Hypothesis

The small scope hypothesis is a key aspect to confirm the effectiveness of our approach, and as such, it must be verified. To do this, we used the mutation testing technique. In order to perform the mutation testing we generated a set of mutants. A mutant can be generated by applying a mutation operator to the program under test. In our case, we concentrated in mutations to the *destroy()* method and accessor methods. The operations we used to generate the mutants are:

- Java operator replacement, e.g., replacing "==" by "*!*=".
- Change of variable/method calls, e.g., calling a different method.
- Change of variables value by adding additional assignments.

In mutation testing, it is important to generate valid mutants, i.e., to generate mutants that can be compiled. Therefore, when generating mutants by changing variable/method calls, the variables used for the mutation were a subset of the original variable.

We found that in most cases, a finitization as small as (1,1,1) that represents up to one owner object, up to one part object, and up to one third party object, was enough to expose the faults introduced by the mutations. The reason for this is that the UML2 implementations heavily use collection types where the behavior of a program with a collection of one element is no different from the behavior when the collection contains many elements. There were two exceptions, however, when a higher number of objects was needed. The first case occurred when checking the composition relationship between the classes Connector and *ConnectorEnd* in the UML metamodel implementation. In this composition relationship, the multiplicity on the *ConnectorEnd* enforces the creation of two objects of *ConnectorEnd* with a link to Connector. In this case, the finitization needed to expose this fault was (1,2,2). The second case occurred when we mutated the method *getOwnedEnd()* in class *Association.* When an *Association* object has two *Property* objects, the fault is not exposed, but in an *Association* object with three *Property* objects the fault is exposed. The reason is that an *Association* object owns the *Property* objects only when it has three or more. If it only has two, the objects are not owned by the *Association* object. For this case, the finitization (1,3,3) was needed.

3.2 Bounded Exhaustive Testing Limitations

3.2.1 Effectiveness

One of the drawbacks of exhaustive test case generation is that it does not guaranty a full exploration of the possible execution paths in the method under test, which may result in missing test cases that could expose a problem in the implementation. As previously explained, the small scope hypothesis can give us a high degree of confidence that a problem will be uncovered, but it cannot be guarantee the method is correct in terms of a particular OCL constraint. This uncertainty may not be acceptable in some scenarios. When working with bounded exhaustive testing, there is always a risk of not selecting a bound large enough. As an example, Figure 3.19 shows a scenario where a particular behavior can only be exposed by an input large enough, making the bound selection an important factor to correctly check all possible behaviors in the method.

```java
public int intPowerOfTwo(int a, int b){
    if(a>30){
        throw new RuntimeException("Result out of range.");
    }
    int result = 2;
    for(int i=1; i<a; i++){
        result *= 2;
    }
    return result;
}
```

Figure 3.19: Bounded exhaustive testing limitations.

3.2.2 Efficiency

In terms of efficiency, compared to an approach that leverages information about previous executions when generating test cases, e.g., test case generation based on branch coverage, bounded exhaustive test case generation is more inefficient. The analysis of do not care fields in our approach can greatly improve on simple exhaustive test case generation, but it still does not completely rule out the generation of extraneous test cases that produce the result. Consider again the method in Figure 3.19. Using bounded exhaustive test case generation with do not care fields, with a large enough bound, e.g., a,$b \leq 35$ we can correctly explore all possible execution paths in the method while reducing the number of test cases generated by detecting b as a do not care field. However, it would still generate a number of extraneous test cases, namely, all test cases where $a \leq 30$ will result in the exact same execution, resulting in a loss of efficiency. Without leveraging previous execution information, such as path condition for branch coverage, it is hard to detect such scenarios to improve the efficiency.

In the following chapter we introduce an improvement that uses a testing validation technique combining coverage criteria and dynamic symbolic execution. Our approach reduces the number of test cases making it more efficient and it guaranties the correctness of a method in terms of a specific OCL constraint.

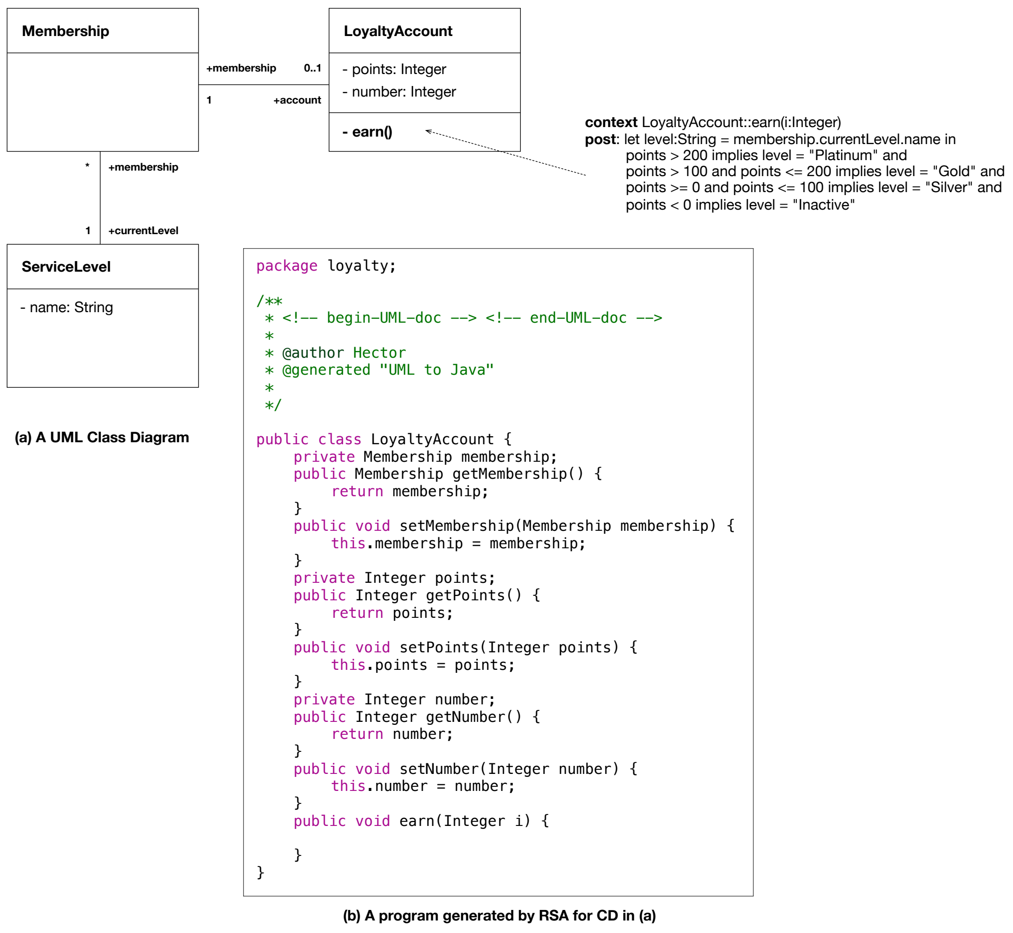
CHAPTER IV

PATH CONDITION ANALYSIS

4.1 Overview

To illustrate our consistency checking approach with path condition analysis, we will use as an example the class diagram shown in Figure 4.1 (a), an excerpt from the Royal and Loyal system [12]. The example consists of the classes *ServiceLevel*, *Membership*, and *LoyaltyAccount*, with a few attributes and associations each. The class diagram also includes an OCL post-condition attached to the method *earn*() in class *LoyaltyAccount*. We will refer to *earn*() as the method under test. Based on this example, the objective of our approach is to determine if the OCL specification for the method *earn*() is satisfied by the method's Java implementation. Also, Figure 4.1 (b) shows a partial skeletal Java Program that was automatically generated by the forward engineering tool Rational Software Architect (RSA) [14].

The automatic code generation by a forward engineering tool can be illustrated by a translation schema that includes a set of rules that map elements in the model to elements in the target language. In this example, the model is the class diagram shown in Figure 4.1 (a) and the target language is Java. For instance, based on RSA's translation schema, shown in Table 2.1, a property in the UML class diagram is mapped to a private class field with a setter and getter method in the Java program. An association in the UML class diagram is treated as a property and it is also translated to a field in the corresponding class. Most translation schemas follow a similar approach when translating UML associations, since most target languages do not directly support the extra semantics associated with them, such as multiplicity.



```
context LoyaltyAccount::earn(i:Integer)
post: let level:String = membership.currentLevel.name in
      points > 200 implies level = "Platinum" and
      points > 100 and points <= 200 implies level = "Gold" and
      points >= 0 and points <= 100 implies level = "Silver" and
      points < 0 implies level = "Inactive"
```

(a) A UML Class Diagram

```java
package loyalty;

/**
 * <!-- begin-UML-doc --> <!-- end-UML-doc -->
 *
 * @author Hector
 * @generated "UML to Java"
 *
 */

public class LoyaltyAccount {
    private Membership membership;
    public Membership getMembership() {
        return membership;
    }
    public void setMembership(Membership membership) {
        this.membership = membership;
    }
    private Integer points;
    public Integer getPoints() {
        return points;
    }
    public void setPoints(Integer points) {
        this.points = points;
    }
    private Integer number;
    public Integer getNumber() {
        return number;
    }
    public void setNumber(Integer number) {
        this.number = number;
    }
    public void earn(Integer i) {

    }
}
```

(b) A program generated by RSA for CD in (a)

Figure 4.1: Forward engineering feature supported by RSA.

The class diagram, Java skeletal program, shown in Figure 4.1, and the developer's implementation for the method $earn()$, shown in Figure 4.2, serve as the input to our approach. To determine if the OCL specification for the method *earn()* is satisfied by the method's Java implementation, as a first step we automatically generate a Java Boolean post-method from the OCL post-condition. The OCL post-condition and the resulting Java Boolean method are shown in Figure 4.3 (a) and Figure 4.3 (b) respectively. Based on the relationship between the OCL post-condition and

```java
public void earn(Integer i){
    points += i;
    if(points > 100){
        membership.getCurrentLevel().setName("Gold");
    }else{
        if(points >= 0){
            membership.getCurrentLevel().setName("Silver");
        }else{
            membership.getCurrentLevel().setName("Inactive");
        }
    }
}
```

Figure 4.2: Developer's *earn()* implementation.

```
context LoyaltyAccount::earch(i:Integer)
post: let level:String = membership.currentLevel.name in
      points > 200 implies level = "Platinum" and
      points > 100 and points <= 200 implies level = "Gold" and
      points >= 0 and points <= 100 implies level = "Silver" and
      points < 0 implies level = "Inactive"
```

(a)

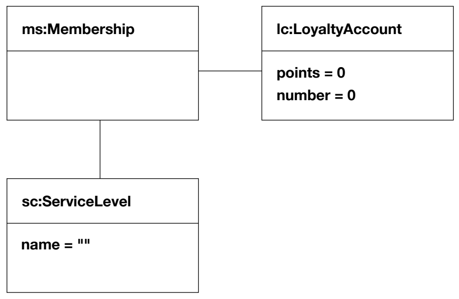
```java
public boolean post_earn(int i){
   String level = this.getMembership().getCurrentLevel().getName();
   boolean r0 = false;
   if ((!(this.getPoints() > 200) || level == "Platinum")
        && (!(this.getPoints() > 100 && this.getPoints() <= 200)
            || level == "Gold")
        && (!(this.getPoints() >= 0 && this.getPoints() <= 100)
            || level == "Silver")
        && (!(this.getPoints() < 0) || level == "Inactive")) {
            r0 = true;
   }
   return r0; }
```

(b)

Figure 4.3: OCL post-condition to Java translation.

the generated Java Boolean method, the problem of finding if the method *earn()* satisfies its OCL post-condition can be reduced to testing if the equivalent Boolean post method always returns true after the execution of the method under test.

After the translation, we generate an initial minimal test case value for the method *earn*() using the translation schema to match elements between the class diagram and its implementation. Each test case value must correspond to a valid object diagram, i.e., the object diagram must be a valid instance of the class diagram shown in Figure 4.1 (a). To generate a minimal test case, i.e., a test case with the smallest number of objects needed to execute the method under test, we start by instantiating the class where the method resides, in this example *LoyaltyAccount*. Next, we parse the class diagram to find required associations in terms of multiplicity and navigability. As an example, looking at the relationship between *LoyaltyAccount* with *Membership*, the multiplicity of 1 tells us that we need exactly one instance of *Membership*. The code in Figure 4.4 (b) show the statements needed to generate the initial minimal test value. These statements are specific to the translation schema introduced earlier.



```java
public void test_case(){
    LoyaltyAccount lc = new LoyaltyAccount();
    Membership ms = new Membership();
    ServiceLevel sc = new ServiceLevel();
    lc.setMembership(ms);
    ms.setLoyaltyAccount(lc);
    ms.setServiceLevel(sc);
    sc.setMembership(ms);
    int i = 0;
    lc.earn(i);
}
```

(a) Object diagram generated by code in (b) **(b) Code to generate object diagram in (a)**

Figure 4.4: Initial test case value.

The next step consists on executing the method under test, $earn()$, at the same time we perform symbolic execution to collect the path condition pc. Next, we execute the Java Boolean post method, $post_{earn()}$, while also performing symbolic execution to collect the path condition pc_{post}. For the purposes of consistency checking, during symbolic execution, we only track all object references, class fields, and method parameters derived from the class diagram. The relationship of the resulting path conditions is then analyzed to guide the test case value generation. Figure 4.5 and 4.7 shows the execution trace and resulting path conditions after the execution of the initial test case and Java Boolean post method. Note that the path condition of the method under test and the post method are both defined in terms of the initial test input values.

If the method $post_{earn()}$ returns false, a software fault has been found and it can be reported. Otherwise, we determine with the use of a SAT solver whether $pc \implies pc_{post}$ is a tautology. If the implication relationship is a tautology, then all test case values satisfying pc also satisfy pc_{post} and follow the same execution path in $earn()$ and $post_{earn()}$ as in the previous execution. In this scenario, the set of test cases satisfying pc is a subset of test cases satisfying pc_{post} as shown is Figure 4.6.

Based on the previous observation, we need to explore a different execution path on $earn()$. To do so, we use a SAT solver to find a new test case value satisfying

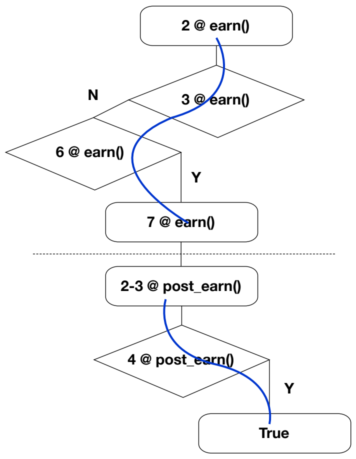


Figure 4.5: Trace I execution, $earn(0)$.

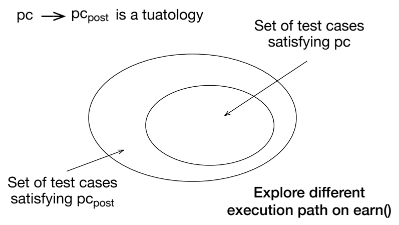

pc → pc_post is a tuatology
Set of test cases satisfying pc
Set of test cases satisfying pc_post
Explore different execution path on earn()

Figure 4.6: pc and pc_{post} relationship after first execution.

$$Path\ Condition$$

$$pc : points_0 + i_0 \leq 100 \wedge points_0 + i_0 \geq 0$$

$$Post\ Method\ Post\ Condition$$

$$pc_{post} : points_0 + i_0 \leq 200 \wedge poits_0 + i_0 \leq 100 \wedge points_0 + i_0 < 0 \wedge name_0 = `Silver'$$

Figure 4.7: pc and pc_{post} path conditions after first execution.

$(points_0 + i_0 \leq 100) \wedge \neg(points_0 + i_0 \geq 0)$, to enforce a different execution path. Our approach uses a last-input-first-output stack to store the path conditions collected during execution and follows a back-tracking approach to explore different execution paths by flipping branches. In this example, the SAT solver returns an assignment of 0 for $points_0$, and -1 for i_0. With this information, we generate a new object diagram as a test case with the values returned by the SAT solver, and execute the method under test, i.e., *earn(-1)*, and post method. Figure 4.8 shows the resulting path conditions.

$$\textit{Path Condition}$$
$$pc : points_0 + i_0 \leq 100 \wedge points_0 + i_0 < 0$$

$$\textit{Post Method Post Condition}$$
$$pc_{post} : points_0 + i_0 \leq 200 \wedge poits_0 + i_0 \leq 100 \wedge points_0 + i_0 < 0 \wedge name_0 = \text{`}Inactive'$$

Figure 4.8: pc and pc_{post} path conditions after second execution.

Once again $post_{earn()}$ returns true and $pc \implies pc_{post}$ is found to be a tautology by the SAT solver. Using a backtracking approach, the SAT solver is used to find an assignment that satisfies $\neg(points_0 + i_0 \leq 100)$. The resulting assignment is $points_0 = 0$, and $i_0 = 150$. Using these assignments, a third test case value is generated, and the methods are once again executed. Figure 4.9 shows the resulting path conditions.

$$\textit{Path Condition}$$
$$pc : points_0 + i_0 > 100$$

$$\textit{Post Method Post Condition}$$
$$pc_{post} : points_0 + i_0 \leq 200 \wedge poits_0 + i_0 > 100\ name_0 = \text{`}Gold'$$

Figure 4.9: pc and pc_{post} path conditions after third execution.

Again, $post_{earn()}$ returns true, but the SAT solver determines that $pc \implies pc_{post}$ is not a tautology. This means that some test values that satisfy pc, following the same execution path on $earn()$, does not follow the same execution path on $post_{earn()}$. In this scenario, the set of test cases satisfying pc is not a subset of test cases satisfying pc_{post} as shown is Figure 4.10. Since a different execution path on $post_{earn()}$ could return false, we attempt to find a test case value which alters the execution path on $post_{earn()}$ with the formula $pc \wedge \neg pc_{post}$, using a backtracking approach on pc_{post}. In this example, the backtracking technique on pc_{post} will generate three test cases. In the first test case, the sub-expression ($name_0$ = "$Gold''$) is negated. As a result, the following formula is sent to the SAT solver: $points_0 + i_0 > 100 \wedge points_0 + i_0 \leq 200 \wedge points_0 + i_0 > 100 \wedge \neg(name_0 = \text{“}Gold'')$. In this case, the value of $points_0 + i_0$ will remain > 100 *and* $<= 200$, which will result in the same execution as the one on the third execution. Regardless of the initial value of $name_0$, its value is overridden by method $earn()$. As a result, we once again find that $pc \implies pc_{post}$ is not a tautology and continue with the backtracking of pc_{post}. In the second case, the sub-expression $points_0 + i_0 <= 200$ is negated, resulting in the following assignment: $points_0 = 0$, $i_0 = 220$. Note that when sending a formula, it is up to the SAT solver how the values are assigned. In this example, an equivalent assignment could be: $points_0 = 220$, $i_0 = 0$.

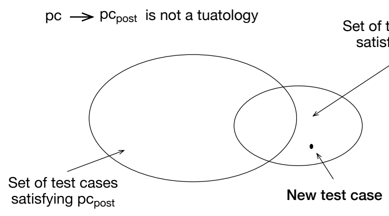

pc → pc_post is not a tuatology
Set of test cases satisfying pc
Set of test cases satisfying pc_post
New test case

Figure 4.10: pc and pc_{post} relationship after third execution.

This time, after the execution of the method under test, we find that $post_{earn()}$ returns false, which means the method $earn()$ does not satisfy the OCL post-condition defined in the class diagram. This means a fault has been found.

4.2 Algorithm

Our approach uses as input a UML class diagram, with at least one method (method under test) with an OCL post-condition, and a Java implementation of the diagram. From the diagram, the OCL constraint is parsed and automatically translated to a Boolean Java method (post method). Next, the method under test and post method are instrumented to support dynamic symbolic execution. If the class diagram contains multiple methods with OCL post-conditions, each method is tested individually using the same procedure. Following the instrumentation, a minimal initial test case value is generated. Finally, a minimal test case that includes the smallest number of objects needed to execute the method under test is generated. The previous initialization steps are shown in lines 3-10 in Listing IV.1.

Having the initial test case value, the method under test and post methods are executed while concurrently performing symbolic execution to collect each method's path condition (lines 11-14 in Listing IV.1). After the execution of the methods, if the post method returns false, the error is reported, otherwise, the path conditions are evaluated and a different test case value is generated to follow a different execution path from the previously observed (lines 15-30 in Listing IV.1). The process is repeated until an error is found or branch coverage has been achieved on the method under test. More details about the initialization, methods execution, and results evaluation are given in the following sub-sections.

4.2.1 Initialization

The initial step consists on parsing the class diagram to extract available OCL post-conditions. To parse the diagram, we use the MDT EMF framework [6], and we support the EMF Ecore model format. Other model formats can be translated to the Ecore model format using a modeling tool such as Eclipse. After parsing the diagram, the OCL post-conditions are translated to an equivalent Java Boolean

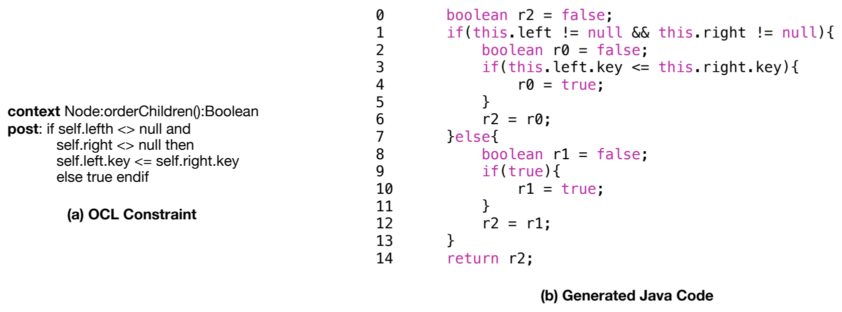
```
input: class diagram CD with OCL, and Java program P

methods = select all methods with attached OCL post-conditions
for each m in methods
   m_post = generate_post_method(m, CD)
   add m_post to class of m
instrument program P to collect the path condition during execution
for each m in methods
   init_values = null
   test_values = generate_testvalues(m, cd, init_values)
   while there exists a new test_values
      pc=[]; heap=[]; pc_post =[]
      exec_symbolic(m, test_values, heap, pc)
      is_valid = exec_symbolic(m_post, test_values, heap, pc_post)
      if is_valid is false
         report error and return
      else
         result = use SAT solver to check if $pc \Rightarrow pc_{post}$ is tautology
      if result is true
         do
            init_values = sat_find_next_init $\neg(pc)$
            if init_values != null
               test_values = generate_test_values(m, cd, init_values)
         while test_values ==null && init_values != null
      else
         do
            init_values = sat_find_next_init $(pc \wedge \neg pc_{post})$
            if init_values != null
               test_values = generate_test_values(m, cd, init_values)
         while test_values == null && init_values != null
```

Listing IV.1: Main algorithm.

method. The translation is based on the OMG OCL 2.31 specification [8] and uses the OCL translation schema introduces my Warmer et al. [12]. The translation process begins by parsing the OCL post-condition using the MDT OCL project [45]. The parsing generates an abstract syntax tree (AST) of the expression. During the parsing process, we construct an equivalent abstract syntax tree that trims some information from the previous AST for easy access and translation. In the generated tree, each node implements a translate method that is recursively called from the root node to generate the corresponding Java code.

To illustrate the translation process, we will use the OCL expression shown in Figure 4.1 (a) as an example. The generated AST is shown in Figure 4.12 where the root element corresponds to the whole IF expression and its child nodes represent the enclosing sub-expressions. For instance, the root node in Figure 4.12 shows an IF expression that is composed of three different sub-expressions, namely, a *condition* expression, a *then* expression, and an *else* expression, which are the child nodes of the IF Exp root node. After the tree is generated, using the Visitor pattern we visit the nodes and recursively call the translate method on each. The translate method generates a Java code fragment for each node. In non-leaf nodes, the method further traverses the child node(s) and generates its corresponding code fragment using the result of its child nodes, and leaf nodes only generate a code fragment that is used by the corresponding parent. Finally, a root node collects all the generated code fragments and produces the final complete Java code for the post method. The code in Figure 4.11 (b) shows the result of the translation and Figure 4.13 shows the translation templates used in this example.



```
context Node:orderChildren():Boolean
post: if self.lefth <> null and
          self.right <> null then
          self.left.key <= self.right.key
          else true endif
```

(a) OCL Constraint

```java
boolean r2 = false;
if(this.left != null && this.right != null){
    boolean r0 = false;
    if(this.left.key <= this.right.key){
        r0 = true;
    }
    r2 = r0;
}else{
    boolean r1 = false;
    if(true){
        r1 = true;
    }
    r2 = r1;
}
return r2;
```

(b) Generated Java Code

Figure 4.11: OCL to Java translation example.

4.2.2 Test Case Generation

Using the class diagram in Figure 4.14 as an example, where *earn*() is the method under test, the minimum test case generation starts by instantiating the class where the method under test resides. In this example, the class *LoyaltyAccount* is instantiated. Note that the class is only instantiated if the method under test is an instance method, otherwise the method under test would not require an instance to be executed, as it is the case with static methods Java. After the instantiation of class *LoyaltyAccount*, we need to recursively check for required associations in terms of multiplicity and navigability. An association is said to be required if the multiplicity's

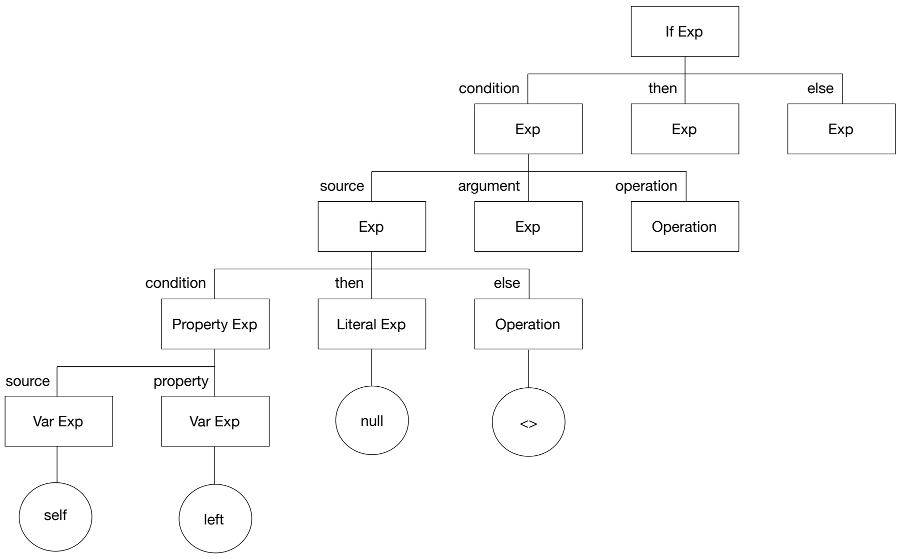
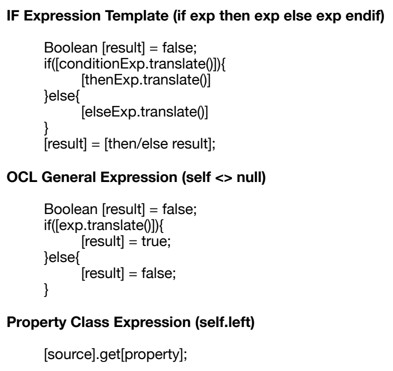


Figure 4.12: Partial generated AST from OCL in 4.11 (a)



IF Expression Template (if exp then exp else exp endif)

```
Boolean [result] = false;
if([conditionExp.translate()]){
        [thenExp.translate()]
}else{
        [elseExp.translate()]
}
[result] = [then/else result];
```

OCL General Expression (self <> null)

```
Boolean [result] = false;
if([exp.translate()]){
        [result] = true;
}else{
        [result] = false;
}
```

Property Class Expression (self.left)

```
[source].get[property];
```

Figure 4.13: OCL Translation Templates.

lower bound is greater than 0. In this example, the association between *LoyaltyAccount* and *Membership* is required, since the lower bound is equal to 1. As a result, we need to instantiate class *Membership* and set the appropriate field on *LoyaltyAccount* to establish the link to *Membership*'s instance. Also, since the association is bi-directional, we need to set the appropriate field on class *Membership* to establish the link to *LoyaltyAccount*'s. This process is repeated recursively for each new instance added to the minimum test case value. In this example, the process is repeated on *Membership*'s instance, where we find a required association to class *ServiceLevel*. In this example, the association between *LoyaltyAccount* and Transaction is not found to be required since the multiplicity's lower bound is 0, making the instantiation of Transaction class optional. For other non-association fields or parameters, we use the primitive types default values and null for reference types. The algorithm for the minimal test case generation is shown in Listing IV.2.

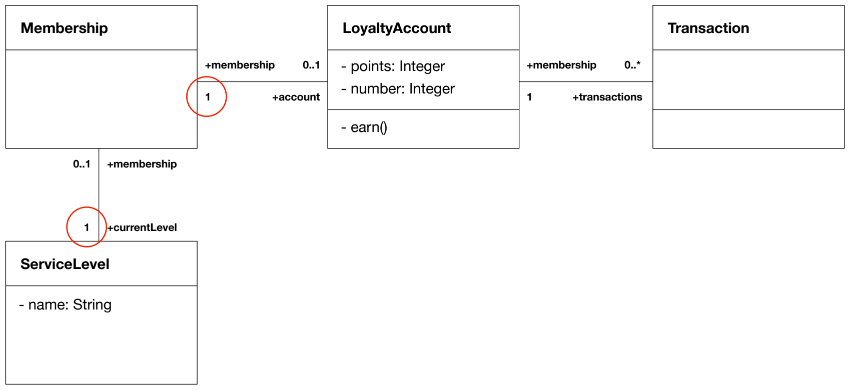


Figure 4.14: Required references for minimum test case value.

4.2.3 Methods Execution and Evaluation

After the execution of the method under test, we initialize the SAT solver based on the observed path conditions. The initialization includes variables, and variable domains, and the sub-expressions in the path conditions. To guarantee that the SAT solver only produces valid assignments in terms of multiplicity, we construct

```
algorithm: generate_test_values
input: Class diagram CD with OCL, method m, and init_values

self_class = class containing method m
self_object = instantiate self_class
set_required_references(self_class, self_object, CD)
if init_values == null
   set parameters and fields to default values
else
   use init_values to initialize parameters and fields
```

Listing IV.2: Test case generation algorithm.

```
algorithm: set_required_references
input: Class diagram CD, class, and object obj

for each association assoc in class
   if assoc multiplicity > 0
      a_class = assoc reference type
      a_object = instance a_class
      set assoc field on obj to a_object
      if assoc is bi-directional
         set opposite field on a_object to obj
      set_required_references(a_class, a_object, CD)
```

Listing IV.3: Set required references algorithm.

the variable domains used by the SAT solver in such a way that no value from the domain can violate the multiplicity constraints. A domain, in the context of a SAT solver, denotes the set of values allowed for a particular variable when trying to find an assignment for a given formula. To construct the domains, we first parse the path condition to select all the variables that correspond to an association end in the class diagram. Next, for each variable, we get its lower bound multiplicity, l, and upper bound multiplicity, h. Based on l and h, the variable domain is defined such that for each value in the domain, value $\geq l$ and value $\leq h$. If the upper bound is unlimited, we generate up to c values, where c is a constant. To illustrate this approach, consider the class diagram in 4.15 and the path condition "*cmpy.dept != null*", where *cmpy* is an instance of class *Company* and dept refers to the association end dept in the association between *Company* and *Department*. Based on the multiplicity of "*0..2*", we generate a domain with the values "*[0, 1, 2]*", where 0 represents the null value, and 1 and 2 indicate that the variable can be assigned with one or at most two instances of type *Department.* For variables that do not correspond to an association end, we generate a predefined number of values depending on the variable type.

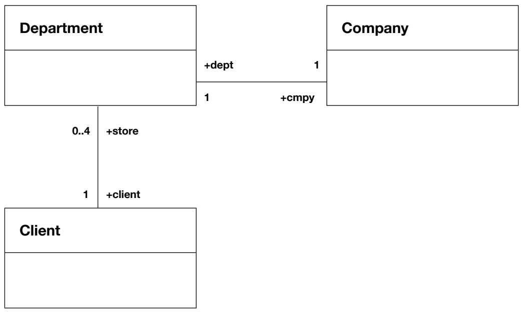


Figure 4.15: Company Class Diagram.

In some instances, multiple variables corresponding to an association end may share the same domain but have different multiplicity values. As a result, the values in the domain may not be enough to enforce the multiplicity constraints for all the association ends. In this case, we add one or more sub-expressions to the formula given to the SAT solver in order to enforce all the constraints. As an example, consider two variables corresponding to the association end dept in the association between *Company* and *Department* and a variable corresponding to the association end store in the association between *Client* and *Department*, with the multiplicities 0..2 and 0..4 respectively. The domain for both variables includes the values "*[0, 1, 2, 3, 4]*". In this example, the domain values only enforce the multiplicity constraint for the variable store. To enforce the multiplicity constraint for the variable dept, we add the sub-expression "*cpmy.dept* <= 2" to the formula sent to the SAT solver.

After the initial construction of a domain, we also dynamically update the domain values, if necessary, after the execution of the method under test and post method, based on the concrete observed values during the execution and the resulting path condition. The update aims at preventing trivially unsatisfiable formulas when flipping branch conditions, due to a domain not having enough values. As an example, consider the execution of the method $earn(0)$ and the resulting path condition $\neg(\$0.points + \$3 > 100) \wedge \$0.points + \$3 \geq 0$, with the initial domain values for variable \$3 including [-1, 0, 1]. If the branch $\neg(\$0.points + \$3 > 100)$ is flipped to cover a different execution path, i.e., $(\$0.points + \$3 > 100)$, and the formula is sent to the SAT solver, the formula will be trivially unsatisfiable since there is no domain value for \$3 that can make the condition $(\$0.points + \$3 > 100)$ evaluate to true. To prevent this, we analyze the path condition and dynamically update the domain based on the variable's relationship with other variables and concrete values in the path condition. In this example, we update the domain for variable \$3 to include [-1, 0, 1, 100, 101]. A similar approach is used with non-primitive types, for example, for an association end *e1* with a multiplicity 0..*, we initially generate a domain with a predetermined number of values, e.g., *domain(e1) = [null, obj1, obj2].* To update the number of objects in a non-primitive type domain, CCUJ requires the size property to be part of the path condition. For example, if the following path condition is observed, $\$0.e1.size() < 4$, we update the domain to include four or more objects. As a result, if the branch is flipped, an assignment is possible.

After the execution of the methods while performing symbolic execution, we obtain the path condition for the method under test, *pc*, and the path condition for the post method, pc_{post}, and a Boolean result, returned by the post method. If the result is false, we can determine that the test case has violated the OCL post-condition and we can report an error. If the result is true, we need to analyze the generated path conditions to decide how to generate the next test case value. Using a SAT solver, we determine whether $pc \implies pc_{post}$ is a tautology. On the first possibility, if $pc \implies pc_{post}$ is a tautology, it means that all test case values satisfying *pc* also satisfy pc_{post}. In this scenario, all the test case values that follow the same execution path on the method under test will also follow the same execution path on the post method. Since the result of the post method was true, we are interested on exploring a different execution path on the method under test to see if we can obtain a different result on the post method. To do this, we use the SAT solver to find a new assignment by negating a sub condition of *pc* using a backtracking technique to eventually achieve branch coverage. On the second possibility, if $pc \implies pc_{post}$ is not a tautology, it means that there is a test case value satisfying *pc* that does not satisfy pc_{post}. In other words, there is a test case value that would follow the same execution path on the method under test but would follow a different execution path on the post method. In this case, we are interested in finding such a test case, since a different execution path on post method could possibly return a different post-condition result. To do so, we use the SAT solver to find an assignment for the formula $pc \wedge \neg pc_{post}$. In both cases, whether $pc \implies pc_{post}$ is a tautology or not, after finding the assignment, the process is repeated until there are no more branches to explore on the method under test or an error is detected.

4.2.4 Multiplicity and Navigability as OCL Pre-Conditions

As introduced earlier, each test case value must correspond to a valid object diagram, i.e., the object diagram must be a valid instance of the class diagram in terms of the *multiplicity* and *navigability* constraints, and it must be a minimal test case, i.e., it should consist on the minimum number of objects needed to execute the method under test, starting with an instance of the class where the method is defined. A minimal valid test case can be specified with set of OCL expressions as *pre-conditions* for the method under test, both in terms of *multiplicity* and *navigability.*

As an example, the *multiplicity* constraint as a *pre-condition*, based on the diagram in Figure 4.1 (a) can be defined as shown in Listing IV.4. The constraint guarantees the method will be executed in a valid object diagram in terms of the *multiplicity.* In this example, the constraints specifies the test case will include an instance of *Membership*, and an instance of *ServiceLevel*, and that both instances will be set using the respective fields, *membership* and *serviceLevel.* In a similar way, the *navigability*

```
context LoyaltyAccount::earn(int i)
pre: self.membership != null and
     self.membership.serviceLevel != null
```

Listing IV.4: Multiplicity as pre-condition.

```
context LoyaltyAccount::earn(int i)
pre: self.membership.account == self and
     self.membership.serviceLevel.membership == self.membership
```

Listing IV.5: Navigability as pre-condition.

constraint ensures the links between instances is set according to the ones defined in the diagram. The *navigability* constraint is shown in Listing IV.5. If no OCL pre-conditions are defined for the method, our approach implicitly uses the *multiplicity* and *navigability* constraints as the method's *pre-conditions*.

4.2.5 Black Box Methods

When executing the method under test and post method, there may be calls to external methods where dynamic symbolic execution is not possible, i.e., we are unable to collect the path condition of their execution. Typically, this is related to cases where instrumentation is not possible due to licensing restrictions, or because the binary files are not directly accessible. We call these type of methods, *black box methods*. In our current implementation, we treat the Java Collection API and the EMF API as *black box* methods. We do not instrument these APIs since they are not part of the design class diagram and doing so would complicate the dynamic symbolic execution since many auxiliary fields introduced by the EMF forward engineering feature would have to be tracked during the execution. In order for our approach to work with these type of methods, we use a set of rules, based on the method's specification, to update the path condition and symbolic memory when a call to an EMF method or Java collection method is detected. In the following subsections we explain the type of scenarios where it is safe to ignore the *black box* path conditions.

Also, we explain how we use approximate symbolic execution to update the symbolic memory according to the semantics of the *black box* methods. In particular, we support approximate symbolic execution for the Java collection methods, i.e., the *List* implementation *ArrayList*, and for the EMF methods *eContents()*, and *eContainer()*.

4.2.5.1 Ignoring Path Conditions of Black Box Methods

In general, the scenario where the path condition of a black box method, bb_{pc}, can be safely ignored, is when the set of test cases satisfying the path condition of the method under test, wb_{pc}, is a subset of the path condition of the black box method, i.e., $wb_{pc} \implies bb_{pc}$ is a tautology, as shown in Figure 4.16. In this scenario, it is safe to ignore bb_{pc} and only consider $wb_{pc} \implies pc_{post}$, since all the test inputs satisfying wb_{pc} will result on the same black box method behavior. On the other hand, if the set of test cases satisfying wb_{pc} is not a subset of bb_{pc}, i.e., $wb_{pc} \implies bb_{pc}$ is not a tautology, as shown in Figure 4.17, the path condition of the black box method cannot be safely ignored, since a test input satisfying wb_{pc} could result on a different black box behavior. In relationship with pc_{post}, there are four possible cases, two for each of the previous scenarios. The cases are shown in the following list:

1. $wb_{pc} \implies bb_{pc}$ is a tautology and $wb_{pc} \implies pc_{post}$ is a tautology
2. $wb_{pc} \implies bb_{pc}$ is a tautology and $wb_{pc} \implies pc_{post}$ is NOT a tautology
3. $wb_{pc} \implies bb_{pc}$ is *NOT* a tautology and $wb_{pc} \implies pc_{post}$ is a tautology
4. $wb_{pc} \implies bb_{pc}$ is *NOT* a tautology and $wb_{pc} \implies pc_{post}$ is NOT a tautology

On the first case, when $wb_{pc} \implies bb_{pc}$ is a tautology, and $wb_{pc} \implies pc_{post}$ is a tautology, our approach backtracks on wb_{pc}. The code in Figure 4.18 shows an example of this scenario. With *earn_1()* as the test method, *save_1()* as the black box method, and *post_earn_1()* as the post condition method. For the second case, where $wb_{pc} \implies bb_{pc}$ and $wb_{pc} \implies pc_{post}$ is not a tautology, our approach backtracks on pc_{post} to try to find a test case that could result on the post condition returning false. This example is also shown in Figure 4.18 with *post_earn_2()* as the post condition method. On the third and fourth cases, $wb_{pc} \implies bb_{pc}$ is NOT a tautology, and ignoring bb_{pc} would result in some test cases being missed. For example, with the code in Figure 4.19, and the test input $i = 101$, the implication between wb_{pc} $\$x > 100$ and pc_{post} is a tautology. However, by pruning all the test cases where $\$x > 100$ we would miss some test cases where the behavior of the black box method would affect the post condition result, as it is the case with $i = 200$ where x is updated to 100, causing the post condition method to return false.

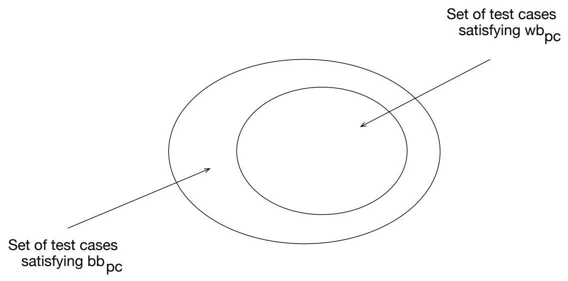

Set of test cases satisfying wb$_{pc}$
Set of test cases satisfying bb$_{pc}$

Figure 4.16: Relationship of a *safe to ignore* black box path condition and pc.

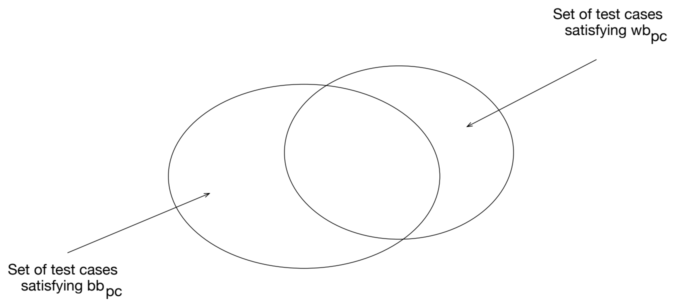

Set of test cases satisfying wb$_{pc}$
Set of test cases satisfying bb$_{pc}$

Figure 4.17: Relationship of a *NOT safe to ignore* black box path condition and pc.

```java
public int x = 0;

public String membership = "Silver";

// W implies B
public void earn_1(int i) {
    x = x + i;
    if (x > 100) {
        membership = "Gold";
    } else {
        if (x > 50) {
            membership = "Silver";
        } else {
            membership = "Inactive";
        }
    }
    save_1(x);
}

public void save_1(int x) {
    if (x > 100) {
        this.x = x + 10;
    } else {
        if (x > 50) {
            this.x = x + 1;
        }
    }
}
```

```java
// W+B implies POST
public boolean post_earn_1() {
    if (x > 100) {
        return (membership == "Gold");
    } else {
        if (x > 50) {
            return (membership == "Silver");
        } else {
            return (membership == "Inactive");
        }
    }
}

// W+B does NOT implies POST
public boolean post_earn_2() {
    if (x > 200) {
        return (membership == "Platinum");
    } else {
        if (x > 100) {
            return (membership == "Gold");
        } else {
            if (x > 50) {
                return (membership == "Silver");
            } else {
                return (membership == "Inactive");
            }
        }
    }
}
```

Figure 4.18: Safe to ignore black box path condition example.

```java
public int x = 0;

public String membership = "Silver";

// W does NOT implies B
public void earn_2(int i) {
    x = x + i;
    if (x > 100) {
        membership = "Gold";
    } else {
        if (x > 50) {
            membership = "Silver";
        } else {
            membership = "Inactive";
        }
    }
    save_2(x);
}

public void save_2(int x) {
    if (x > 190) {
        this.x = x - 100;
    } else {
        if (x > 90) {
            this.x = x + 10;
        } else {
            if (x > 40) {
                this.x = x + 1;
            }
        }
    }
}
```

```java
// W+B implies POST
public boolean post_earn_3() {
    if (x > 100) {
        return (membership == "Gold");
    } else {
        if (x > 50) {
            return (membership == "Silver");
        } else {
            return (membership == "Inactive");
        }
    }
}

// W+B does NOT implies POST
public boolean post_earn_4() {
    if (x > 180) {
        return (membership == "Platinum") || (membership == "Gold");
    } else {
        if (x > 100) {
            return (membership == "Gold");
        } else {
            if (x > 50) {
                return (membership == "Silver");
            } else {
                return (membership == "Inactive");
            }
        }
    }
}
```

Figure 4.19: Not safe to ignore black box path condition example.

4.2.5.2 Approximate Symbolic Execution of Java Collection Methods

add(Object obj), *addAll(Collection c)*

When the method *add()* is executed, the symbolic memory representing the *size* property of the collection is updated to be *1* larger than its original value. As an example, consider the list *l* for the method under test shown in Figure 4.20. The symbolic value of the list and its *size* property can be represented as l_0, and $l_0.size$ respectively. For the statement *l.add(p)* in Line 2, the symbolic value of the *size* property is updated to be $l_0.size + 1$. After the symbolic memory is updated, the path condition for the statement *if (l.size() > 2)* in Line 4, with the initial list being empty, is represented as $l_0.size + 1 <= 2$. Based on this path condition, our algorithm recognizes that in order to flip the branch we need to add at least *2* elements to the collection. Similar to the *add()* method, when *addAll()* is executed, the symbolic memory of the size property for the collection is updated, this time to be size of the original collection plus the number of objects on the collection used as a parameter. For example, the execution of $l1_0.addAll(l2_0)$ will update the the symbolic memory of the *size* property of $l1_0$ to be $l1_0.size + l2_0.size$.

```
0   public boolean addExample(Person p) {
1       if (p != null) {
2           l.add(p);
3       }
4       if (l.size() > 2) {
5           return true;
6       } else {
7           return false;
8       }
9   }
```

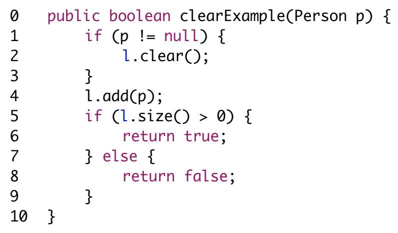
Figure 4.20: Collection black box methods *add()* and *size()* example.

size(), *isEmpty()*

When the method *size()* appears in a path condition, our algorithm recognizes that the observed concrete value corresponds to the number of elements in the collection. For example, for the statement *if (l.size() > 2)* in Figure 4.20, the resulting symbolic path condition is l_0*.size* $<=$ *2*, assuming the input list is empty at the moment of execution. Based on this path condition, our algorithm recognizes that we need to add at least two elements to the collection in order to flip the branch. When working with the *size()* method, we never flip a branch in such a way that the size would be given a negative value, since this is not a valid program state. As an example, the branch l_0*.size* $==$ *0* is only flipped as l_0*.size* $>$ *0*. In a similar way, the operation *isEmpty()* is treated as an special case of the method *size()*. If the path condition contains l_0*.isEmpty()* $==$ *true*, it is replaced with l_0*.size* $==$ *0*. Otherwise, if the path condition is l_0*.isEmpty()* $==$ *false*, it is replace with l_0*.size* $>$ *0*. After the update, the algorithm proceeds as with the normal *size()* operation.

clear()

When the operation *clear()* is called, the symbolic memory for the *size* property is set to *0* using its original symbolic value. For example, using Figure 4.21 as an example, the statement *l.clear()* in Line 2 updates the symbolic memory for the *size* property as l_0*.size* - l_0*.size*. Subsequent calls that would affect the *size* property, such as *add()* in Line 4, will further update the symbolic value accordingly. As an example, after the execution of the statement *l.add(obj)*, the resulting symbolic memory would be l_0*.size* - l_0*.size* $+$ *1*, and a statement such as *if (l.size()* $>$ *0)* would result in the path condition l_0*.size* - l_0*.size* $+$ *1* $>$ *0*.



```
0   public boolean clearExample(Person p) {
1       if (p != null) {
2           l.clear();
3       }
4       l.add(p);
5       if (l.size() > 0) {
6           return true;
7       } else {
8           return false;
9       }
10  }
```

Figure 4.21: Collection black box method *clear()* example.

contains(Object obj), containsAll(Collection c)

Using the example in Figure 4.22, with list *l*, and object *p*, the execution of the statement *l.contains(p)* is mapped to the symbolic path condition p_0 *in* l_0, if the result of the concrete execution is *true*, or p_0 *not in* l_0 if *false*. Our algorithm recognizes this pattern and either adds the object *p* to the collection or removes it in order to flip the branch. In this example, the resulting path condition is p_0 *!= null and* p_0 *not in* l_0. Similar to the method *contains()*, the method *containsAll()* is mapped as *list* $l2_0$ *in* $l1_0$, with the difference that the first argument in the expression refers to a second list of objects. For these type of branches, our algorithm recognizes that in order to flip the branch, all the objects in *l2* must be added to the target list *l1*, or not, depending on how the branch is being flipped.

```java
public boolean containsExample(Person p) {
    if (p != null && l.contains(p)) {
        return true;
    } else {
        return false;
    }
}
```

Figure 4.22: Collection black box method *contains()* example.

indexOf(Object obj), lastIndexOf(Object obj)

With the list *l* and object *p* as input, the statement *if (l.indexOf(p) == 3)* is mapped as the symbolic path condition p_0 *in* l_0 *== 3* if the concrete execution is *true*, or p_0 *in* l_0 *!= 3* if the result of the concrete execution is *false*. Our approach recognizes this as the object *p* being contained on the list at the specific position *3*, if the concrete execution result is true, not being on the list at that position if the concrete execution result is false. To flip this branch, our algorithm adds the object exactly at position *3* by adding two other objects followed by the object *p*. In a similar way to *indexOf()*, the method *lastIndexOf()* is mapped as the symbolic path condition *last* p_0 *in* l_0 *== 3*, with the addition of the keyword *last*, to indicate that on the event of the list having the same object more than once, the index of the last occurrence must be equal to *3*. For these type of method, to flip a branch, the argument object is placed at the specified position and we make sure that no occurrences of the same object are found afterwards.

equals(Object obj)

The method equals on a collection type returns *true* if the parameter object is also a list, both lists are of the same size, and all the objects in the target list are equal to the elements in the parameter list and they are at the same position, e.g., first element on *l1* is equal to the first element on *l2*, second element *l1* is equal to the second element on *l2*, etc. The execution of the statement *if (l1.equals(l2)* in the method under test, with *l1*, and *l2* as input, is mapped to the symbolic path condition $list\ l1_0 == list\ l2_0$. For these type of branches, flipping the branch requires having exactly the same set objects on both lists and at the same positions, if the required outcome is *true*.

***remove(Object obj)*, *remove(int index)*,**

The method *remove(Object obj)*, as the opposite of method *add()*, decreases the value of the *size* property of the list, but only if the parameter object was initially contained in the list. To determine if the *size* property must be updated, during dynamic symbolic execution, we look at the concrete value returned by the operation. Only if the execution of the method returns *true*, the memory is updated. If the result is *true*, the *size* property is mapped to the symbolic value $l_0.size$ - *1*. In the case of the method remove(int index), the symbolic value of the *size* property is always updated. The reason is that if the index is out of range, the program would trow an exception, otherwise and element is always removed.

4.2.5.3 Approximate Symbolic Execution of EMF Methods

eContents()

The EMF method *eContents()* returns an unmodifiable list containing all owned elements of an object, determined by its *containment features* [21]. The features are set during the forward engineering process and are defined as part of the static field *OWNED_ELEMENT_ESUBSET*. To generate the list of owned elements, the method uses the *feature id's* in the static field. Each *feature id* corresponds to each of the *fields* in the class containing the owned elements. For each *feature id*, the method calls the corresponding *get()* method to retrieve the objects. The execution of the method, which includes accessing *feature id's*, and the logic determining which *get()* method to call, is part of the *black box* method, but the execution of the *get()* methods is not. As an example, consider the method in Figure 4.23, where *c* is an instance of UML *Class*. Following our approach, on the second execution *c* is not *null* and the statement in line 8 with the black box method is executed. The resulting path condition in this example is (c_0 *!= null and* c_0*.eContents().size() == 3 and* c_0*.ownedAttributes == null and* c_0*.ownedComments and* c_0*.owned...*). As mentioned earlier, the execution of the *black box* method determines which *get()* method is executed but the execution of the *get()* method itself is white and it becomes part of the path condition, allowing us to see the fields access. In this example, *eContents()* executes all the *get()* methods of fields that correspond to owned elements of *Class*. Based on the path condition, we know we need to add 3 objects in one of the fields that are used to construct the list returned by *eContents()*. To determine what fields in the path condition are part of the execution of *eContents()*, we look at the static field *OWNED_ELEMENT_ESUBSET* using Java reflection and use the feature id's to determine the name of the fields that conform the *owned set* returned by *eContents()*. After identifying the fields, we instantiate the required number of objects and add them to the corresponding list. The reason we need to look at the static field *OWNED_ELEMENT_ESUBSET* to identify the fields is that we do not have other way of knowing which of the fields that appear in the path condition are part of the call to *eContents()* since we loose track of the execution while in the black box method.

```java
public void blackBoxExample(Class c) {

    if (c == null) {

        return;
    }

    if (c.eContents().size() > 3) {
        // ...
    }

}
```

Figure 4.23: Example with black box EMF method *eContents()*.

eContainer()

The EMF method *eContainer()* returns a reference to its containing object. An object, *a*, is said to be contained by an object, *b*, if *a* is part of *b's* contents, as determined by its *containment features*. In other words, *b* is *a's* container if *b.eContents().contains(a)* is true. The implementation of *eContainer()* returns the value of an auxiliary field containing the reference. This field is set when the contained object is added to the a particular container. In our approach, *eContainer()* is considered a *black box* method. To support it, when the method is executed, we need to determine the *feature id* of the field in the *container* object in order to be able to set it. For example, for the branch *obj.eContainer() == null*, we need to identify the *owner* type and the field that can be used to set the *containment* relationship. To support this operation, we use information from the class diagram to identify the owner types where the owned element is of the same type of the object executing the *eContainer()* operation. The class diagram also gives us information about the name of the field that needs to be set to establish the containment relationship. Using this information, our approach can flip the branch *obj.eContainer()* $==$ *null* by making *obj* a contained object of the owner type via the identified field.

CHAPTER V

EVALUATIONS

5.1 Bounded Exhaustive Testing

Our experiments include the checking of three UML composition implementations: The first one is a University example, the second one is the UML metamodel implementation v2.2.1, and the third one is the UML metamodel implementation v1.1.1. Both UML metamodel implementations are from the IBM's Eclipse Project and are based on the Eclipse Modeling Framework (EMF). The implementation of the University example was provided by us.

Also, we manually applied the OCL constraints defined in Section 3.1.3 to code automatically generated by several forward engineering tools. We found that most tools fail or, in some cases, completely ignore the implementation of basic association properties such as navigability. Only two tools, Fujaba [46] and EMF [6], correctly implement bidirectional navigability, but, similar to the rest of the tools, they leave out the implementation for the subsetting, union, redefinition, and multiplicity properties.

To conduct our experiments, we used the Java bytecode manipulation and analysis framework ASM [47]. We instrumented the programs to monitor the fields derived from a class diagram. To this purpose, we use as an input a class diagram to gather information about the association properties to instrument the relevant fields. In the following sections we discuss our results.

5.1.1 Effectiveness

We used our approach to check two UML metamodel implementations: UML2 v1.1.1, and UML2 v2.2.1. We detected violations of the lifetime property in UML2 v1.1.1, while the shareability property was correctly implemented. UML2 v2.2.1 did not present any problems in both the lifetime and shareability property.

The fault in UML2 v1.1.1 was caused by the *destroy()* method that expects the composition relationship information to be stored in a resource object. To implement composition, the *destroy()* method iterates over every object contained in a resource looking for links to the owner object or its parts, and any link found during the check is removed. The problem with this approach is that a resource object in EMF does not provide such information. A resource object in EMF is intended to provide a mechanism to persist and reference other persisted object by serializing classes in a class diagram. When the set of objects involved in a composition relationship are not part of a resource, the *destroy()* method does not correctly enforces the lifetime property.

The fault in UML2 v1.1.1 was originally confirmed in the Eclipse UML2 forum, and was later corrected in versions such as UML2 v2.2.1. In later versions, EMF attaches an adapter to each EObject that tracks several changes, such as adding or removing references to the object. The *destroy()* method in UML v2.2.1 uses the adapters to locate and remove all links to the owner or part objects. Following this approach, UML v2.2.1 correctly enforces the lifetime property.

Based on our results with the two UML composition implementations, we conclude with a high degree of confidence that UML composition was correctly implemented in v2.2.1. Also we conclude that our approach can effectively detect faults, as shown with the detection of the fault in UML2 v1.1.1.

Table 5.1 shows the results of testing the implementation of seven forward engineering tools for the navigability, multiplicity, subsetting, union, and redefinition properties, as well as the shareability and lifetime properties for composition. Only unidirectional navigability is fully supported, bidirectional navigability is supported by two tools. Composition properties are only supported by (EMF).

Table 5.1: Forward engineering tools test.

Tool	Unidirectional Navigability	Bidirectional Navigability	Link	Subsetting	Union	Redefinition	Shareability	Lifetime
Fujaba	OK	OK	Fail	Fail	Fail	Fail	Fail	Fail
EMF	OK	OK	Fail	Fail	Fail	Fail	OK	OK
Argo UML	OK	Fail	Fail	Fail	Fail	Fail	Fail	Fail
Visual Paradigm	OK	Fail	Fail	Fail	Fail	Fail	Fail	Fail
Omondo	OK	Fail	Fail	Fail	Fail	Fail	Fail	Fail
Astah*	-	-	-	-	-	-	-	-
Altova*	-	-	-	-	-	-	-	-
* Unable to test, the tools do not provide getter/setter implementations								

5.1.2 Efficiency

To show our tools efficiency in the generation of the test input space, we compare it with *Korat*, a tool for test inputs generation for Java programs [48]. *Korat* requires an invariant method, called *repOK()* to generate all the test inputs. In our case, for the test of composition the invariant method should be based on the two constraints, namely the multiplicity and navigability constraints. As an example, in Figure 3.12, *repOK()* for class Company checks the multiplicity and navigability of class Division's end. For class Division, *repOK()* checks the multiplicity and navigability of the ends for classes Company and Client. For class Client, *repOK()* checks the ends form classes Division and University. And finally, *repOK()* for class University checks the multiplicity and navigability of class Client's end.

Since *Korat* uses the all-field-read tracking technique, when *repOK()* for class

Company is executed and the end for class Division is read, the execution of the method *repOK()* in class Division is triggered. In a similar way, the method *repOK()* is executed for classes Client and University. Following this technique, *Korat* generates test cases in terms of all the classes that can be reached from the owner class via an association end.

We initially include objects of type University, but the field that references the objects of type University is quickly identified as a do not care field, as explained in Section 0, reducing the size of the test case and consequentially the size of the input space.

Since *Korat* does not support the use of collection types in the generation of test inputs, we wrote a simulator to support the UML implementation. There are more than two hundred test cases; we show five of them in Table 5.2. We ran the test cases in an Intel(R) Pentium(R) M 1.60GHz processor with 750Mb of RAM. The first test case is our own implementation of the University example. The last four test cases are from the UML2 v2.2.1 metamodel implementation. From our results that we can reduce between 96% and 99% of *Korat's* test input space, where larger finitization shows greater reductions. We can also see that *Korat* can easily run timeout even with a small number of objects because *Korat* considers several non-relevant classes.

Even without considering non-relevant classes, if the number of relationships related to the composition relationship is large, the test space can still be considerably large. One of the reasons the test input space we generate remains small in most cases is thanks to the principle of high decoupling in most software engineering projects. This principle requires a moderate number of relationships between two classes. Previous experiments based on eight industry standard software systems showed that most classes have between 1 and 10 relationships with the average number of relationships being 2.86 [49].

5.2 Mutation Testing

Mutation testing is a technique used to evaluate the quality of a test suite [9]. In mutation testing, small modifications that imitate programmers errors are introduced in a program, and the resulting program is called a *mutant*. After generating a mutant, we evaluate the program using our test cases with the objective of detecting a difference in the behavior between the original program and the mutant just generated. When a difference in behavior is detected, it is said the mutant has been killed [9]. A high rate of killed mutants indicates a strong set of test cases. There are different types of mutation operations that can be applied to a program to generate a mutant. Table 5.3 shows the type of mutation operations that we applied when evaluating our test cases. The table also shows an example of the mutation produced by the operator.

```java
public static Property getOpposite(Property property) {

    if (property.isNavigable()) {
        Association association = property.getAssociation();

        if (association != null) {
            EList<Property> memberEnds = association.getMemberEnds();

            if (memberEnds.size() == 2) {
                int index = memberEnds.indexOf(property);

                if (index != -1) {
                    Property otherEnd = ((InternalEList<Property>) memberEnds)
                        .basicGet(Math.abs(index - 1));

                    if (!association.getOwnedEnds().contains(otherEnd)
                        || association.getNavigableOwnedEnds().contains(
                            otherEnd)) {

                        return otherEnd;
                    }
                }
            }
        }
    }

    return null;
}
```

Figure 5.1: Property getOpposite() method.

We applied mutation testing to 21 methods, 2 of them from the UML specification, *getName()*, *getOpposite()*, and 19 automatically generated methods from the OCL constraints introduced by Booch et al. [8]. We generated a total of 383 mutants. To evaluate each mutant, the first step is to execute a set of test cases against the original program, and then on each of of the generated mutants. The procedure to generate the set of test cases follows the approach described in Chapter IV. For each method, we take as input the program's class diagram, with the method under test, and its OCL post condition, from which a post-method is generated to serve as test oracle. A key difference in mutation testing is that even if no error is detected by the method's post condition, if the output differs from the output of the original program the mutant is still considered to be killed.

Among the 383 mutants generated, 360 were killed, for a 93.8% killing rate. In our experiments, non killed mutants are equivalent to the original code. For example, *index != -1* is equivalent to *index > -1*, since, in the context of the method *getOpposite*, index cannot take a value less than -1. Specifically, the non killed mutants are, from Table 5.4, mutant *4*, *6*, *7*, and *8*. For mutant *4*, the behavior of the program remains the same as long as *memberEnds* is equal or greater than 2. In the case of *6*, *7*, and *8*, the variable *index*, as explained earlier, cannot take a value less than -1. In all cases, the behavior of the program remains unaffected. It is important to note that when applying a mutation operator, if the generated mutant is not valid, i.e., it cannot compile, the mutant is discarded. The code for the method *getOpposite()* is shown in Figure 5.1.

We also applied the small scope hypothesis based method to test the 383 mutants, with 5 as the number of possible values for each of the variables. The small scope hypothesis based method killed 242, for a 73.8% killing rate. One of the reasons for the lower killing rate is the reduced branch coverage achieved by the small scope hypothesis based method. While our approach can achieve 100% coverage, the small scope hypothesis method achieves only 70% coverage. Table 5.6 shows a summary of the results with the list of the tested methods, number of mutants, number of killed mutants, and branch coverage achieved by each method.

5.3 Path Condition Analysis

We evaluated our approach with two kinds of experiments: effectiveness and efficiency. Effectiveness can be evaluated by whether our approach is able to find real faults in some industry strength software systems. To evaluate the efficiency, we compared our approach to related techniques that can be applied to do consistency checking. The project we chose to do the experiments is Eclipse's UML2 project [50], an EMF based UML implementation. One of the reasons for choosing the project is that it is large, with more than 300 meta- classes with large number of OCL constraints. These constraints include post conditions and well-formed rules. Also, being an open source project, it has a good bug report system which allowed us to confirm our findings.

5.3.1 Effectiveness

When we studied the UML specification [7], we found that many existing approaches that claimed to recover UML composition by reverse engineering from a Java program do not strictly follow the semantics of UML composition [7]. The UML specification requires that "*If a composite is deleted, all of its parts are normally deleted with it. Note that a part can (where allowed) be removed from a composite before the composite is deleted, and thus not be deleted as part of the composite...*" p. 41. However, many existing approaches require that all part objects cannot be accessed by any object except for its owner object. In fact, this is not the case. For instance, the class diagram excerpted from the UML specification in Figure 5.2 shows that an object of class Property, which is owned by an object of class Class, can be accessed by an object of class Association. Therefore, when an owner object does not exist, all of its owned objects should not exist. Namely, all the links to the owned objects from other live objects should be removed. Assume method *destroy()* intends to implement the deletion of an owner object, Figure 5.3 (a) shows the property as a post-condition after method *destroy()* is called on an owner object.

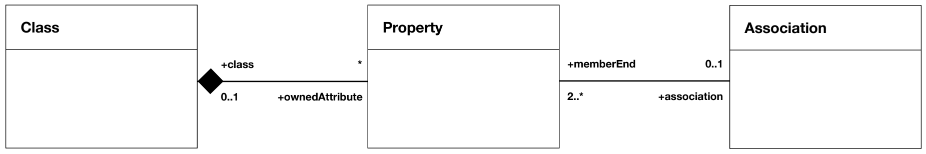


Figure 5.2: Partial UML metamodel.

```
context Class::destroy(): Boolean
post: self.ownedAttribute@pre->forAll(p |
        p.association@pre.memberEnd->excludes(p))
```

(a) Post-condition for destroy method

```
context Property::isAttribute(p: Property): Boolean
post: result = Classifier.allInstances->exists(c |
        c.attribute->includes(p))
```

(b) Post-condition for isAttribute method

Figure 5.3: UML metamodel post conditions.

After the above observation, we applied our approach on one of the UML2 projects, i.e. the UML2 v1.1.1 implementation. We detected the implementation error of all fields derived from UML composition and was confirmed with one of UML2 project members. The root cause of the implementation error is that the *destroy()* method iteratively checks each object contained in the resource, which is supposed to contain all the instantiated objects, and remove their links to the owned objects being destroyed. But the resource object, as part of EMF metamodel, did not automatically store all instantiated owned objects in the resource object appropriately. For this OCL constraint, we found a total of 999 error instances. We also applied our approach to a later version UML2 v2.2.1 and the previous fault has been fixed.

We also used our approach to check a set of 55 well-formedness rules on the UML2 project v4.0.2. We detected a software fault, related to the OCL post condition for the method *isAttribute()* defined in the UML specification (p.125 [7]). Figure 5.4 shows the OCL post condition and the automatically generated Java Boolean method used as post method. The generated method includes the auxiliary methods OclOperations.allInstances() that implements the OCL allInstances operation, and OclOperations.includes() that implements the OCL operation includes. As an initial minimal test case, CCUJ instantiates class Property and null is used as the default value for the parameter p. After the initial minimal test case generation, CCUJ executes the method under test, *isAttribute()* and its post method, *post_isAttribute()*. The rest of the process follows as explained in Chapter IV. The reason of the fault was because the implementation only checks the non-navigable inverse references to property p, this is, the references in which an object (*obj1*) can access p, but p cannot directly access the object *obj1*. Since the reference attribute in class Classifier is a navigable inverse reference, it was ignored, and the method failed to return true when c.attribute->includes(p) is true. The problem was confirmed and fixed by the developers.

```
context Property::isAttribute(p: Property): Boolean
post: result = Classifier.allInstances->exists(c |
        c.attribute->includes(p))
```

(a) Post-condition for isAttribute method

```
public static Boolean post_isAttribute(PropertyImpl p) {
     Iterator<?> it0 = ((Collection<?>) OclOperations
                                        .allInstances("Classifier"))
                                        .iterator();
     Boolean r1 = false;
     if (((Collection<?>) OclOperations
          .allInstances("Classifier"))
         .size() > 0) {
          while (it0.hasNext()) {
               Classifier c = (Classifier) it0.next();
               Boolean r0 = false;
               if (OclOperations.includes(
                         (Collection<Property>) c.getAttributes(), p)) {
                    r0 = true;
               }
               r1 = r0;
               if (r0) {
                    break;
               }
          }
     } else {
          r1 = true;
     }
     Boolean r2 = false;
     if (r1 == self.isAttribute(p)) {
          r2 = true;
     }
     return r2;
}
```

(b) Generated Java Boolean method

Figure 5.4: OCL post condition and generated Java Boolean method.

5.3.2 Efficiency

To determine the efficiency, we compared our approach with a glass box testing approach [11], and *Korat* [48], a black box testing approach, which are two prominent approaches, in terms of the number of generated test cases. One reason for this selection is that these two approaches consider different methods to generate test case values. The number of test cases determines the number of times that the method under test must be executed. Since our approach achieves branch coverage, the smaller the number of necessary test cases, the greater the efficiency. Both of *Korat* and the glass box testing approach use finitization to limit the number of values that can be assigned to a field. While the small scope hypothesis is the assumption of both approaches, they cannot be as effective as our approach since we explore different test cases to cover the branch coverage criteria. In the case of the glass box testing approach, the search space is pruned by ignoring fields not accessed during the execution of the method under test.

To compare our approach with *Korat* on the generation of the test input state space, we consider the Multiplicity and Navigability in a class diagram and convert these constraints into an invariant method *repOK()* that *Korat* uses to generate all test cases. For example, in Figure 5.2, *repOK()* in Class checks the multiplicity and navigability on Property's end. Method *repOK()* in Property checks the multiplicity and navigability on both Class and Association's end.

The results of the comparison for four methods from the UML2 project are shown in Table 5.7. The second column in the table shows the number of classes involved on each of the tests, which represents a fragment of the UML metamodel. Considering all the metamodel classes would greatly increase the number of test cases generated by the black bock approach, since it would recursively call *repOK()* in all the classes involved. On the third column we have different finitization values, a 3 indicates that both the black box and glass box testing approaches would consider up to 3 different values for a given field during the test case generation. As opposed to our approach that considers all possible values. The fourth column shows the number of test cases generated by the glass box testing approach. For the method *maySpecializeType()*, since the method under test has only one parameter that can take up to 3-5 different values, depending on the finitization, and no extra fields are touched during the execution, the number of generated test cases is equal to the finitization number.

For the second method, extra fields are touched during the execution of the method under test, increasing the number of generated test cases. The fifth column shows the number of test cases generated by the black box testing approach. The number of test cases is determined by number of parameters, number of classes, and the associations between them, where each of the fields can take up to 3-5 values depending on the finitization. This results in a large number of test cases since the black box testing approach tries all possible combinations for all the fields involved. Finally, the last column shows the number of test cases generated by our approach.

Table 5.2: Comparison with *Korat*.

	Finitization	Number of Fields Read	Number of Test Inputs	Time to Generate Test Inputs	Total Time to Test Inputs (ms)	Number of Fields Read by Korat	Number of Test Inputs by Korat	Time to Generate Test Inputs by Korat (ms)	Total Time to Test Inputs by Korat (ms)
1	(1,1,2,2)	4	3	2133.25 ms	2275.07 ms	8	25	2048.96 ms	2407.49 ms
	(1,1,2,3)	4	3	2129.19	2275.48	9	81	2151.7	3308.38
	(1,1,3,2)	5	4	2172.04	2315.42	10	125	2429.84	4344.84
	(1,1,4,2)	6	5	2186.85	2328.28	12	625	2361.85	11668.1
	(1,1,4,3)	6	5	2196.66	2339.11	13	6561	2642.19	95677.17
2	(1,1,1,1,1,2)	7	8	2134.12	2855.57	10	125	2256.15	4159.92
	(1,1,1,1,2,2)	8	12	2214.36	2935.72	12	625	2343.75	12262.5
	(1,1,1,1,2,3)	8	12	2230.18	2951.33	13	6561	3062.41	108825.73
	(1,1,1,2,2,3)	10	18	2353.65	3077.47	15	59049	time out	time out
	(1,1,1,2,2,4)	10	18	2397.33	3119.18	16	1419857	time out	time out
3	(1,1,1,1,1,1,2)	7	8	2156.17	2707.14	12	729	2335.9	13803.07
	(1,1,1,1,2,1,2)	8	12	2296.73	2943.62	14	6561	2963.54	107152.22
	(1,1,1,1,2,2,2)	8	12	2284.25	2951.57	16	83521	time out	time out
	(1,1,1,2,2,2,2)	10	18	2313.58	3136.75	18	1419857	time out	time out
	(1,1,1,2,2,3,2)	10	18	2383.76	3122.19	20	39135393	time out	time out
4	(1,1,1,1,1,1,1,1)	7	8	2246.18	2683.42	13	729	2551.63	14018.8
	(1,1,1,1,1,1,1,2)	8	8	2312.01	2893.8	15	4913	2673.09	81870.65
	(1,1,1,1,2,1,1,2)	8	12	2293.27	2912.47	17	83521	time out	time out
	(1,1,1,1,2,1,2,2)	10	12	2243.13	3284.42	18	1185921	time out	time out
	(1,1,1,2,2,1,2,2)	10	18	2351.37	3189.14	20	39135393	time out	time out
5	(1,1,1,1,1,1,1,1,1)	7	8	2237.71	2897.47	15	4913	2783.85	90753.16
	(1,1,1,1,1,1,1,1,2)	7	8	2215.31	2838.09	16	35937	time out	time out
	(1,1,1,1,2,1,1,1,2)	8	12	2240.2	3015.76	18	1185921	time out	time out
	(1,1,1,1,2,1,1,2,2)	8	12	2236.18	3093.24	20	17850625	time out	time out
	(1,1,1,2,2,1,1,2,2)	10	18	2312.49	3121.1	22	1160290625	time out	time out

Table 5.3: Test case generation comparison.

Mutation Operator	Original Code	Mutation Example
Absolute Value Insertion	x = 3 * a;	x = 3 * abs (a);
Arithmetic Operator Replacement	x = a + b;	x = a - b;
Relational Operator Replacement	if (m > n)	if (m >= n)
Conditional Operator Replacement	if (a && b)	if (a || b)
Assignment Operator Replacement	x += 3;	x -= 3;
Unary Operator Insertion	x = 3 * a;	x = 3 * -a;

Table 5.4: Mutations for method getOpposite().

#	Line	Mutation	Type
1	3	if (!property.isNavigable()) {	Unary Operator Insertion
2	5	if (association == null) {	Relational Operator Replacement
3	8	if (memberEnds.size() != 2) {	Relational Operator Replacement
4	8	if (memberEnds.size() > 2) {	Relational Operator Replacement
5	8	if (memberEnds.size() < 2) {	Relational Operator Replacement
6	11	if (index == -1) {	Relational Operator Replacement
7	11	if (index > -1) {	Relational Operator Replacement
8	11	if (index < -1) {	Relational Operator Replacement
9	16	if (_association.getOwnedEnds().contains(otherEnd) ...	Unary Operation Deletion
10	16	... contains(otherEnd) && association.getNavigableOwnedEnds() ...	Conditional Operator Replacement
11	13	... memberEnds).basicGet(Math.abs(index + 1)); ...	Arithmetic Operator Replacement
12	13	... memberEnds).basicGet(Math.abs(index * 1)); ...	Arithmetic Operator Replacement
13	13	... memberEnds).basicGet(Math.abs(index / 1)); ...	Arithmetic Operator Replacement

Table 5.5: Mutations for method earn().

#	Line	Mutation	Type
1	2	points -= i	Assignment Operator Replacement
2	2	points *= i	Assignment Operator Replacement
3	2	points /= i	Assignment Operator Replacement
4	2	points %= i	Assignment Operator Replacement
5	2	points >= i	Assignment Operator Replacement
6	2	points |= i	Assignment Operator Replacement
7	2	points ^= i	Assignment Operator Replacement
8	2	points <<= i	Assignment Operator Replacement
9	2	points >>= i	Assignment Operator Replacement
10	2	points >>>= i	Assignment Operator Replacement
11	2	points += abs(i)	Absolute Value Insertion
12	2	points += -abs(i)	Absolute Value Insertion
13	2	points += failOnZero(i)	Absolute Value Insertion
14	3	if (points >= 100)	Relational Operator Replacement
15	3	if (points < 100)	Relational Operator Replacement
16	3	if (points <= 100)	Relational Operator Replacement
17	3	if (points != 100)	Relational Operator Replacement
18	6	if (points < 0)	Relational Operator Replacement
19	6	if (points <= 0)	Relational Operator Replacement
20	6	if (points != 0)	Relational Operator Replacement

Table 5.6: Mutation testing comparison.

	% Branch Coverage			# Mutants Killed		% Mutants Killed	
Method	**CCUJ**	**Small Scope**	**# of Mutants**	**CCUJ**	**Small Scope**	**CCUJ**	**Small Scope**
earn()	100%	66%	20	19	19	95%	95%
getOpposite()	100%	64%	13	10	10	76%	76%
earn2()	100%	82%	116	110	42	95%	36%
ofAge()	100%	71%	11	10	5	91%	45%
checkDates()	100%	100%	8	8	8	100%	100%
knownServiceLevel()	100%	70%	4	4	2	100%	50%
correctCard()	100%	100%	4	4	4	100%	100%
levelAndColor()	100%	78%	22	22	10	100%	45%
minServices()	100%	58%	11	10	10	91%	91%
sizesAgree()	100%	55%	18	17	17	94%	94%
noAccounts()	100%	54%	34	32	32	94%	94%
numberOfParticipants()	100%	58%	11	10	10	91%	91%
numberOfParticipants2()	100%	58%	11	10	10	91%	91%
firstLevel()	100%	70%	4	4	2	100%	50%
isEmpty()	100%	58%	11	10	10	91%	91%
birthdayHappens()	100%	58%	11	10	10	91%	91%
upgradePointsEarned()	100%	58%	11	10	10	91%	91%
totalPoints()	100%	79%	11	10	7	91%	64%
totalPointsEarning()	100%	71%	33	31	20	94%	61%
totalPointsEarning2()	100%	90%	19	19	4	100%	21%
Average	100.00%	70.02%	383	360	242	93.78%	73.87%

Table 5.7: Test case generation comparison.

			Number of Test Cases			
Test	**No. Classes**	**Finitazation**	**Glass Box**	**Black Box**	**Finitazation**	**Our Approach**
Property::isAttribute()	2	3	26	6561	∞	2
	2	4	64	65536	∞	2
	2	5	125	390625	∞	2
StateMachine::ancestor()	8	3	27	6561	∞	10
	8	4	64	65536	∞	10
	8	5	125	390625	∞	10
Classifier::isTemplate()	4	3	108	2916	∞	6
	4	4	256	16384	∞	6
	4	5	500	62500	∞	6
Element::destroy()	3	3	27	531441	∞	3
	3	4	64	16777216	∞	3
	3	5	125	244140625	∞	3

CHAPTER VI

CONCLUSIONS

This dissertation presents two consistency checking approaches. The first is based on a bounded exhaustive testing technique with test input space pruning. Our pruning technique is based on *do not care fields* that detects fields that are not relevant for the method under test. Our technique was able to reduce the number of required test cases and successfully detect errors in two industry strength UML projects [50]. However, our exhaustive testing technique includes some drawbacks, namely, it does not guarantees a full exploration of the possible execution paths in the method under test, which may result on missing test cases that could expose a problem in the implementation, and a lack of efficiency compared to an approach that leverages information about previous executions when generating test cases, e.g., test case generation based on branch coverage, bounded exhaustive test case generation is more inefficient. Our second approach address these concerns. In our second approach we introduce a consistency checking technique that uses a testing-based validation technique that combines coverage criteria and dynamic symbolic execution with path conditions analysis to reduce the number of input states without sacrificing accuracy. It extends traditional lazy initialization for minimum test case generation from a class diagram, and extends universal symbolic execution to work with Java bytecode for consistency checking. Our approach was applied to multiple methods with invariants and post conditions in UML implementation projects, and was able to detect multiple errors while efficiently generating a small number of test cases. Some of the drawbacks of our approach are related to the limitations on SAT solver. Our approach only supports as data types basic arrays, objects, and integer types, and basic path condition constructs. Floating type numbers are not supported. As a future work we intend to expand support for multiple data types and path conditions.

APPENDIX A

UML ASSOCIATION PROPERTIES FORMALIZATION

In the following subsections we present each of the association properties in more detail and give examples of its use in a class diagram. Also we use the Object Constraint Language (OCL) to strictly define each of the properties at M2 and M1 level. The formalization of the association properties allows us to find deviations between a UML class diagram and its implementation in terms of the different types of association relationships. The following sections are organized as follows: for each property, we present its definition based on the UML specification, a formal definition using the OCL language at M2 level, the algorithm used to generate the properties constraints at M1 level, and an example of the generated M1 level constraints.

A.1 Navigability

As previously introduced, navigability on an association relationship is used to specify the accessibility of class instances. Navigability is represented by an arrow on the navigable end, if both ends are navigable, the arrows are normally omitted. The class diagram in Figure A.1(a) shows an example of unidirectional navigability,

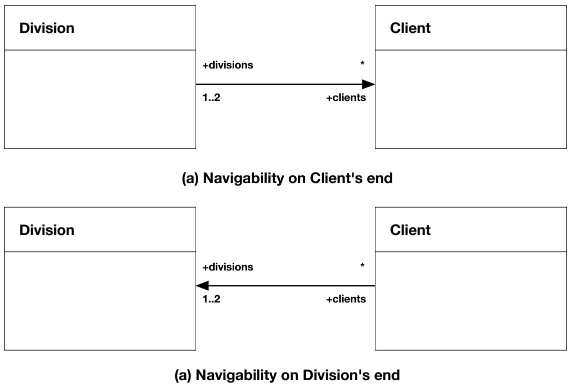


Figure A.1: Navigability on a class diagram.

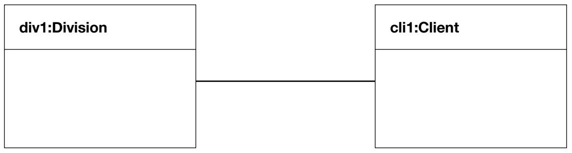


Figure A.2: An object diagram for class diagram in Figure A.1(a)

where an instance of class Division can efficiently access an instance of class Client but not the opposite way. On Figure A.1(b) the navigability is bidirectional.

A.1.1 Navigability Constraint at M2 Level

The navigability constraint is defined for the Association context. Using Figure A.1(a) as an example, for each member end on the association, m1 = clients, m2 = divisions, if member end m1 is navigable, the instance associated with the opposite member end m2 must have a reference to the instance associated with the navigable member end m1. In Figure A.2, the instance div1 is associated with the navigable member end m1, and instance cli1 is associated with the opposite member end m2. The instance div1 must have a reference to instance cli1.

```
context Association
inv: self.memberEnd->forAll(m1, m2 | m1 <> m2 implies
     (m1.isNavigable(self) implies m2.hasReferenceTo(m1, self)))
```

In the constraint, the method isNavigable() returns true if the member end is navigable, e.g., the method would return true for member end m1 (clients), and false for member end m2 (divisions). The method hasReference() returns true if the instance associated with member end m2 (div1 in Figure A.2) has a reference to the instance associated with the member end m1 (cli1 in Figure A.2). More details about auxiliary operations, such as isNavigable(), can be found in Section 0.

A.1.2 Algorithm to Generate Navigability Constraint at M1 Level

The algorithm to generate navigability constraints at M1 works as follows: for every association in the model that is not unidirectional (it has at least two navigable member ends) the algorithm selects its navigable member ends and for each pair-wise combination it generates a pair of constraints, one for each member end.

```
for each association ``a'' in the model

// selects navigable member ends
let ends : Set(Property) = a.memberEnd->reject(m | a.ownedEnd->
    includes(m))

if ends size >= 2
ends->forAll(e1, e2 | e1 <> e2, generate constraints )

    // values in square brackets are resolved from the model
        context e1:[e1 type]
        inv: e1.[e2 name]->forAll(e2 | e2.[e1 name]->includes(e1))

        context e2:[e2 type]
        inv: e2.[e1 name]->forAll(e1 | e1.[e2 name]->includes(e2))
```

A.1.3 Navigability Constraint at M1 Level

Using the class diagram in Figure A.1(b) as an example, we automatically generate two OCL constraint for each of the navigable member ends in the diagram that are part of a bidirectional association. For this example a total of 2 OCL constraints are generated. Each constraint checks that, for each pair of classes, C1 and C2, that are part of a bidirectional association, if an object of type C1 has a reference to an object of type C2, the object of type C2 must also have a reference to the object of type C1.

```
context d:Division
inv: d.clients->forAll(c | c.divisions->includes(d))
```

context c:Client inv: c.divisions->forAll(d | d.clients->includes(c))

A.2 Multiplicity

The multiplicity of an association end constraints the number of references between the instances of the participating classes in the association. Multiplicity can be specified by a single number or a range in the form "a..b", where "a" represents the lower multiplicity and "b" represents the upper multiplicity. An "*" can be used to represent an unlimited upper multiplicity. As an example, Figure A.1(b), the multiplicity "0..5" on Client's end specifies that an instance of class Division can have a reference to 0 or at most 5 instances of class Client. The "*" on Division's end indicates that an instance of class Client can have a reference to 0 or an unlimited number of instances of class Division.

Some associations can also include a qualifier property, a property used to group a set of elements in a member end with a multiplicity greater than 1. For example, the class diagram in Figure A.3 shows the qualifying property LineId, this property is part of the class Items. This means that an instance of class Invoice can have a reference to no more than 5 instances of class Item with the same LineId, but it can have a reference to more than 5 if LineId is different.

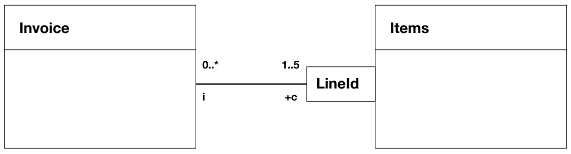


Figure A.3: Qualified association.

A.2.1 Multiplicity Constraint at M2 Level

The multiplicity constraint is defined for the meta-class Class context. For each instance i of class self, and for each association a where self is a participating class, we count the number of links or instances of a where i is referenced. The number of references must be within the multiplicity range for that particular association.

```
context Class
inv: self.instances()->forAll(i |
      Association.allInstances()->forAll(a |
      a.memberEnd->forAll(m1, m2 | m1.type = self and m1 <> m2 and
      a.links()->size() > 0 implies if m2.qualifier->size() = 0
      then checkCount(linksFor(a, i, m1)->size(), m2) else
      checkQualifying(a, i, m1, m2) endif )))
```

In the constraint, the operation allInstances() returns a collection with all the instances of meta-class Association. The operation checkCount() returns true if the number of links (linksFor()) is within the multiplicity range for the member end m2. The operation checkQualifying() is similar to checkCount() but it takes into consideration the qualifying property.

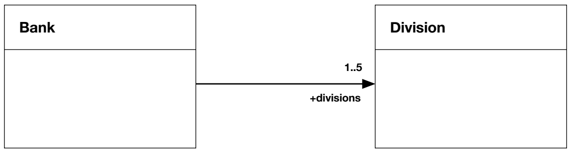


Figure A.4: Class diagram with multiplicity constraints.

A.2.2 Algorithm to Generate Multiplicity Constraint at M1 Level

For each association in the model, the algorithm selects the navigable member ends and generates a constraint for each. The algorithm considers two scenarios related to the multiplicity type, the first one when the multiplicity is set to exactly 1 and the second one when the multiplicity is given by a range with a non-infinity upper bound. The outline of the algorithm is as follows:

```
for each association ``a'' in the model
```

```
a.memberEnd->forAll(m1 | generate constraints)

if not a.ownedEnd->includes(m1) // if m1 is navigable
// the values inside square brackets are resolved from the model
case 1: m1.upperValue = 1 and m1.lowerValue = 1
 context c:[m1 ownerClass]
              inv: c.[m1 name] <> OclVoid
case 2: m1.upperValue > 1
              context c:[m1 ownerClass]
              inv: c.[m1 name]->size() <= m1.upperValue
                   and c.[m1 name]->size() >= m1.lowerValue
```

A.2.3 Multiplicity Constraint at M1 Level

A multiplicity constraint is automatically generated for each member end with a bounded multiplicity. As an example, in Figure A.4, one OCL constraint is generated for clients member end with a bounded multiplicity of 1..5.

```
context div:Division
inv: div.clients->size() <= 5 and div.clients->size() >= 1
```

A.3 Subsetting

A subsetting association is defined by the UML specification as follows:

“Subsetting represents the familiar set-theoretic concept. It is applicable to the collections rep-resented by association ends, not to the association itself. It means that the subsetting association end is a collection that is either equal to the collection that it is subsetting or a proper subset of that collection. (Proper subsetting implies that the superset is not empty and that the subset has fewer members.) Subsetting is a relationship in the domain of extensional semantics. [7]” p. 40. The class diagram in Figure A.5 shows an example where properties b1 and b2 on class A1 subset property b on class A. This means that the collection contained in property b1 and property b2, is a subset of the collection contained in property b.

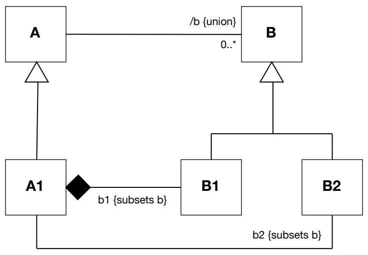


Figure A.5: Subsetting and Union associations.

A.3.1 Subsetting Constraint at M2 Level

The subsetting constraint is defined for the meta-class Class context. For each instance i of class self, if self is a participating class on a subsetting association, the instance i must be part of the union set of all subsetting association.

```
context Class
inv: self.instances()->forAll(i |
     self.subsettingAssociations()->forAll(a |
     self.valuesForAssociation(a, i)->forAll(l |
     self.isSubset(i, l, self.allSubsettedValues(a, i)))))
```

The constraint uses several auxiliary operations, namely, instances(), subsettingAssociations(), valuesForAssociation(), isSubset(), and allSubsettedValues(). For more information about these methods, please refer to Section 0.

A.3.2 Algorithm to Generate Subsetting Constraint at M1 Level

The algorithm to generate subsetting constraints at M1 works as follows: for every association in the model we select the navigable member ends that subset at least one property. The generated constraint checks if the values in the subsetting property are included in at least one of the subsetted properties.

```
for each association ``a'' in the model
a.memberEnd->forAll(m1 | generate constraints)

if not a.ownedEnd->includes(m1) and m1.redefinedProperty is not empty
let R = m1.redefinedProperty

// the values inside square brackets are resolved from the model
context c:[m1 ownerClass type]
```

```
Inv: c.[m1 name]->forAll(x |

x.oclAsType([R[0] owenerClass]).[R[0] name]->includes(x) or
x.oclAsType([R[1] owenerClass]).[R[1] name]->includes(x) or
...
x.oclAsType([R[n] owenerClass]).[R[n] name]->includes(x))
```

A.3.3 Subsetting Constraint at M1 Level

For the subset property, we generate an OCL constraint for each of the subsetting properties and the subsetted property. As an example, in Figure A.5 we generate an OCL constraint for the subsetting properties b1 and b2, and one for the subsetted property b. The constraints check that each element in the subsetting properties is also an element of the subsetted property.

```
context a:A1
inv: a.b1->forAll(x | a.oclAsType(A).b->includes(x))


context a:A1
inv: a.b2->forAll(x | a.oclAsType(A).b->includes(x))
```

A.4 Union

An association end can be marked as a union to show that the end is derived by the union of its subsets. On Figure A.5, property b on class A is the union of properties b1 and b2 on class A1.

A.4.1 Union Constraint

On the meta-class Class context, if class self is part of a union, every instance i of class self must be part of at least one of the subsetting sets.

```
context Class
inv: self.instances()->forAll(i |
      self.unionAssociations()->forAll(a |
     a.memberEnd->forAll(p | p.isDerivedUnion implies
      self.propertyValues(p, a, i)->includesAll(self.unionSubsetValues
         (p,i)) and
      self.unionSubsetValues(p,i)->includesAll(self.propertyValues(p,
         a, i)))))
```

The constraint uses the OCL method includesAll() that checks if a collection includes all the elements in a second collection. The constraint also uses three auxiliary operations, unionAssociations(), propertyValues(), and unionSubsetValues().

A.4.2 Algorithm to Generate Union Constraint at M1 Level

For each association in the model the algorithm selects the navigable member ends that are subsetted at least once and generate a constraint using the following template.

```
for each association ``a'' in the model
a.memberEnd->forAll(m1 | generate constraints)

if not a.ownedEnd->includes(m1)  // if m1 is navigable
let R be all properties redefining m1
// the values inside square brackets are resolved from the model
       context c:[R[0] type]
inv: c.oclAsType([R[0] ownerClass]).[m1 name]->forAll(x |
     c.[r[0] name]->includes(x)

context c:[R[n] type]
inv: c.oclAsType([R[1] ownerClass]).[m1 name]->forAll(x |
     c.[r[n] name]->includes(x)
```

A.4.3 Union Constraint at M1 Level

For the union property, we generate an OCL constraint for the union property b. The constraints checks that each element in the property must be part of at least one of the subsetting properties.

```
context a:A1
inv: a.oclAsType(A).b->forAll(b | a.b1->includes(b))


context a:A1
inv: a.oclAsType(A).b->forAll(b | a.b2->includes(b))
```

A.5 Redefinition

A redefining association is defined by the UML specification as follows:

“Redefinition is a relationship between features of classifiers within a specialization hierarchy. Redefinition may be used to change the definition of a feature, and thereby introduce a specialized classifier in place of the original featuring classifier, but this usage is incidental. The difference in domain (that redefinition applies to features) differentiates redefinition from specialization. [7]” 40.

A redefining association must remain consistent with the redefined association with respect of its type and multiplicity constraints. The type of the redefining property must be a subtype of the redefined property and the lower and upper multiplicity range must be contained on the lower and upper multiplicity range of the redefined property.

A.5.1 Redefinition Constraint

The redefinition constraint is defined for the Association context. For each member end m in the association, the constraint checks whether each redefining property r is consistent with the multiplicity values of m.

```
context Association
inv: self.memberEnd->forAll(m | m.redefinedProperty->size() > 0
    implies
      m.redefinedProperty->forAll(r | m.type.conformsTo(r.type) and if
      r.upperValue.oclAsType(LiteralUnlimitedNatural).value < 0 then
      m.upperValue.oclAsType(LiteralUnlimitedNatural).value < 0 else
      m.upperValue.oclAsType(LiteralUnlimitedNatural).value <=
      r.upperValue.oclAsType(LiteralUnlimitedNatural).value endif and
      m.lowerValue.oclAsType(LiteralInteger).value >=
      r.lowerValue.oclAsType(LiteralInteger).value ))
```

A.5.2 Algorithm to Generate Redefinition Constraint at M1 Level

For each pair of redefined and redefining properties, we use the algorithms previously introduced to generate the navigability, multiplicity, subsetting, and union constraints.

A.5.3 Redefinition Constraint at M1 Level

For a redefining property, we automatically generate a constraint for each of the properties previously defined, i.e. navigability, multiplicity, subsetting, and union. The constraints are generated following the same approach. A true evaluation of the redefining property implies a true evaluation of the redefined property. Also, a violation of the redefined property implies a violation of the redefining property.

A.6 Composition

According to the UML specification, a composite aggregation, or composition, is a strong form of aggregation. Composition is a subtype of the whole-part relationship. A whole-part relationship indicates that an element is composed by one or more parts (elements). A composition can be represented in a class diagram as a solid/filled black diamond and is differentiated from a regular aggregation by the aggregationKind property. Its representation is defined in the UML specification as follows:

"An association with aggregationKind = shared differs in notation from binary associations in adding a hollow diamond as a terminal adornment at the aggregate end of the association line. The diamond shall be noticeably smaller than the diamond notation for associations. An association with aggregationKind = composite likewise has a diamond at the aggregate end, but differs in having the diamond filled in" [7] p. 42.

The diagram in Figure A.6 shows the aggregate end as a filled diamond next to class Company. In this example, class Company is a composite class, and class Division is the part class of the composite aggregation.

The semantics of composition applies certain constraints on the relationship between instances of the composite and part classes, i.e., instances of class Company and instances of class Division respectively.

"If a composite is deleted, all of its parts are normally deleted with it. Note that a part can (where allowed) be removed from a composite before the composite is deleted, and thus not be deleted as part of the composite. Compositions may be linked in a directed acyclic graph with transitive deletion characteristics; that is, deleting an element in one part of the graph will also result in the deletion of all elements of the subgraph below that element. Composition is represented by the isComposite attribute on the part end of the association being set to true" [7] p. 41.

We call the above constraint the lifetime property, because it is related to the lifetime of the composite and its parts. In essence, this property requires that if a composite is removed, its parts should also be removed.

Also from the UML specification, we can see the concept of the lifetime property exemplified in the following sentences, where classifier refers to a composite class:

"A part declares that an instance of this classifier may contain a set of instances by composition. All such instances are destroyed when the containing classifier instance is destroyed" [7] p. 184.

The following fragment of the UML specification describes one more constraint:

"An association may represent a composite aggregation (i.e., a whole/part relationship). Only binary associations can be aggregations. Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time" [7] p. 41.

The above constraint is related to how a part can be shared among different composites. The constraint specifies that a part should not be shared by more than one composition at time. We call this constraint the shareability property.

A.6.1 Composition Constraints

A.6.1.1 Shareability

We define the global shareability constraint for the meta-class Class context. The constraint first checks if self is an owned class, if true, we check that for every instance of self oc, and for every two composite classes where self is the part class, there is not more than one link at a time from an owner object.

```
context Class
inv: self.isOwnedClass() implies self.instances()->forAll( oc |
     self.ownerCompositions()->forAll(comp1, comp2 | not (comp1 <>
        comp2 and
     oc.linkExist(comp1, comp2))))
```

A.6.1.2 Lifetime

For the lifetime property, our constraint first checks if self is a composite class, then for each instance of self i, if i is deleted, each instance of part classes oci owned by i should also be deleted.

```
context Class
inv: self.isCompositeClass() implies self.instances()->forAll(i |
     i.isUndefined() implies self.ownedClasses()->forAll(oc |
     oc.instances()->forAll(oci | i.ownsObject(oci)
     implies oci.isUndefined()))))
```

A.6.2 Algorithm to Generate Composition Constraint at M1 Level

The Lifetime Constraint is defined as post-condition for the method implementing the lifetime property. For each class in the model that is an owner class we generate two constraints as follows:

```
for each class c in the model that is an owner class
let P be all navigable association member ends of type c

// the values inside square brackets are resolved from the model
        context [c name]::[methodName]():Boolean
```

```
post:     [P[0] ownerClass].allInstances->forAll(x |
            x.[P[0] name]->excludes(self)) and
               [P[1] ownerClass].allInstances->forAll(x |
            x.[P[1] name]->excludes(self)) and
   ...
               [P[n] ownerClass].allInstances->forAll(x |
            x.[P[n] name]->excludes(self)) and

for each class c in the model that is an owner class
let M be the set of composite member ends owned by c, for each m in M
let P be the set of navigable association member ends of the same
   type of m

// the values inside square brackets are resolved from the model
      context [c name]::[methodName]():Boolean
   post: self.[m name]->forAll(x |
             [P[0] ownerClass].allInstances(y | y.[P[0] name]->
                excludes(x) and
             [P[1] ownerClass].allInstances(y | y.[P[1] name]->
                excludes(x) and
      ...
             [P[n] ownerClass].allInstances(y | y.[P[n] name]->
                excludes(x))
```

A.6.3 Composition Constraint at M1 Level

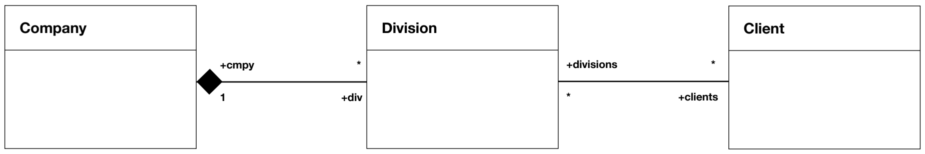


Figure A.6: Composition example.

For the class diagram in Figure A.6, the following constraint, as a post-condition of the method destroy, checks that no instance of class Client has a reference to the destroyed division object.

```
context Company::destroy()
post:self.div->forAll(x | Client.allInstances(y | y.divisions->
   excludes(x))
```

Also, for the shareability property we generate one OCL constraint for each possible combination of composition pairs that share the same part class. For the class diagram in Figure A.7, we generate the following four constraints.

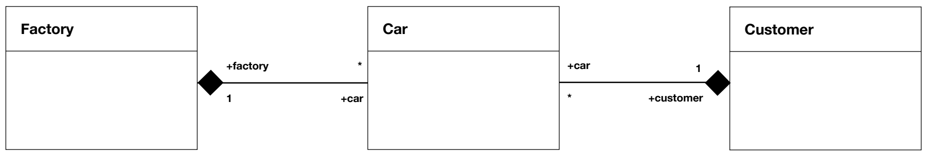


Figure A.7: Shared composition example.

```
context factory:Factory
inv: factory.car->forAll( car | Customer.allInstances()->forAll(c |
    not c.car->includes(car))
```

```
context customer:Customer
inv: customer.car->forAll( car | Factory.allInstances()->forAll(f |
    not f.car->includes(car))
```

A.7 Composition Constraint Auxiliary Operations

A.7.1 isComposition()

Returns true if self is a composite association, otherwise false.

```
context Association
def: isComposition() : Boolean =
        self.memberEnd->exists(me | me.isComposite)
```

A.7.2 isCompositeClass()

Returns true if self is a composite class, otherwise false.

```
context Class
def: isCompositeClass() : Boolean =
        self.superClasses()->exists(c |
        Property.allInstances()->exists(p |
        p.isComposite and p.getOtherEnd().type = c))
```

A.7.3 superClasses()

Returns a set of classes that are super classes of class self.

```
context Class
```

```
def: superClasses() : Set(Class) =
     Class.allInstances()->select(c | self.conformsTo(c))
```

A.7.4 ownedClasses()

Returns a set of classes owned (part classes) by class self.

```
context Class
def: ownedClasses() : Set(Class) =
     Class.allInstances()->select(c |
     Property.allInstances()->exists(p |
     p.isComposite and c.superClasses()->exists(s |
     p.type = s and p.getOtherEnd().type = self)))
```

A.7.5 isOwnedClass()

Returns true if class self is owned by another class, otherwise false.

```
context Class
def: isOwnedClass() : Boolean =
     Class.allInstances()->exists(c |
     c.ownedClasses()->includes(self))
```

A.7.6 compositions()

Returns the set of composite associations in the model.

```
context Class
def: compositions() : Set(Association) =
     Association.allInstances()->select(a |
     a.isComposition())
```

A.7.7 instances()

Returns a set with all the instances of class self.

```
context Class
def: instances() : Set(InstanceSpecification) =
     InstanceSpecification.allInstances()->select(i |
     i.classifier->includes(self))
```

A.7.8 isUndefined()

Returns true if the instance self is undefined.

```
context InstanceSpecification
def: isUndefined() : Boolean =
     InstanceSpecification.allInstances()->forAll(
     i | i.slot->forAll(s | Association.allInstances()->forAll(
     a | i.classifier->includes(a) implies s.value->forAll(
     v | v.oclAsType(InstanceValue).instance <> self))))
```

A.7.9 ownerCompositions()

Returns a set of composite associations where self is the owned class.

```
context Class
def: ownerCompositions() : Set(Association) =
     compositions()->select(c1 |
     c1.memberEnd->exists( m |
    m.isComposite and m.type = self))
```

A.7.10 ownsObject(i:InstanceSpecification)

Returns true if the instance self owns the insntace i, otherwise false.

```
context InstanceSpecification
def: ownsObject(i:InstanceSpecification) : Boolean =
     self.slot->exists(s | s.value->exists(
     v | v.oclAsType(InstanceValue).instance = i))
```

A.7.11 linkExist(a1:Association, a2:Association)

Returns true if the instance self is referenced by a link of type a1 and a link of type a2 simultaneously.

```
context InstanceSpecification
def: linkExist(a1:Association, a2:Association) : Boolean =
     InstanceSpecification.allInstances()->exists(
     link1 , link2 | link1 <> link2 and
     link1.classifier->includes(a1) and
     link2.classifier->includes(a2) and
     link1.slot->exists(s | s.value-> exists(
     v | v.oclAsType(InstanceValue).instance = self ))
     and link2.slot->exists(s | s.value-> exists( v |
     v.oclAsType(InstanceValue).instance = self )))
```

RIGOROUS ANALYSIS OF THE ALGORITHM IN PATH CONDITION ANALYSIS

Structural Operational Semantics of Concolic Execution

SR-SymVar-Seq:

$$\frac{v \text{ is a symbolic expression}}{< v; e_1, \omega > \longrightarrow < e_{1,}\omega >}$$

SR-Read-Field

$$\frac{\omega.\gamma(v) \neq null}{< v.f, \omega > \longrightarrow < \omega.\Gamma(v.f), \omega[\mathrm{C}: (\omega.\mathrm{C})(v! = null)^{<l,true>}] >}$$

SR-Field-Assignment

$$\frac{\omega.\gamma(v) \neq null}{<v.f = v_1, \omega> \longrightarrow <v_1, \omega\left[\gamma: \omega.\gamma[(v,f) \longmapsto v_1], \Gamma: \omega.\Gamma[(v,f) \longmapsto v_1], \mathrm{C}: [(\omega.\mathrm{C})(v! = null)^{<l,true>}]\right]>}$$

SR-If

$$\frac{.}{<if\ true\ then\ e_1\ else\ e_2, \omega> \longrightarrow <e_1, \omega>}$$

$$\frac{.}{<if\ false\ then\ e_1\ else\ e_2, \omega> \longrightarrow <e_2, \omega>}$$

SR-While

$$\frac{.}{< while_k\ e_0\ do\ e_1, \omega> \longrightarrow < if\ e_0\ then\ e_1; while_{k-1}\ e_0\ do\ e_1\ else\ skip, \omega>} \quad \text{if } k > 0$$

SR-New

$$\frac{\alpha \notin \mathrm{HOs} \quad \bar{f}\ is\ a\ set\ of\ fields\ of\ C}{<new\ C, \omega> \longrightarrow <\alpha, \omega\left[\gamma: \omega.\gamma[(\alpha,\bar{f}) \longmapsto \{null\}], \Gamma: \omega.\Gamma[(\alpha,\bar{f}) \longmapsto \{null\}]\right]>}$$

SR-Method-Call

$$\frac{\omega.\gamma(v) \neq null \quad mbody(m,v) = \bar{x}.e}{<v.m(\bar{v}_1), \omega> \longrightarrow <e[\bar{v}_1/\bar{x},\ \ v/this], \omega[C: (\omega.C)(v! = null)^{<l,true>}]>}$$

SR-EQ-T

$$\frac{\omega.\gamma(v_0) == \omega.\gamma(v_1)}{< v_0 == v_1, \omega > \longrightarrow < \boldsymbol{true}, \omega[C: \omega.\mathrm{C}(v_0 == v_1)^{<l,true>}] >}$$

SR-EQ-F

$$\frac{\omega.\gamma(v_0)\ ! = \omega.\gamma(v_1)}{< v_0 == v_1, \omega > \longrightarrow < \boldsymbol{true}, \omega[C: \omega.\mathrm{C}(v_0 == v_1)^{<l,false>}] >}$$

SR-AND-T

$$\frac{\omega.\gamma(v_0) == true}{< v_0 \&\& e_1, \omega > \longrightarrow < e_1, \omega[C: \omega.C(v_0 == true)^{<l,true>}] >}$$

SR-AND-F

$$\frac{\omega.\gamma(v_0) == false}{< v_0 \&\& e_1, \omega > \longrightarrow < \boldsymbol{false}, \omega[C: \omega.C(v_0 == true)^{<l,false>}] >}$$

SR-OR-T

$$\frac{\omega.\gamma(v_0) == true}{< v_0 \,||\, v_1, \omega > \longrightarrow < \boldsymbol{true}, \omega[C: (v_0 == true)^{<l,true>}] >}$$

SR-OR-F

$$\frac{\omega.\gamma(v_0) == false}{< v_0 \,||\, v_1, \omega > \longrightarrow < v_1, \omega[C: (v_0 == true)^{<l,false>}] >}$$

SR-NOT-T

$$\frac{\omega.\gamma(v_0) == true}{< !\, v_0 \,, \omega > \longrightarrow < \boldsymbol{false}, \omega[C: (v_0 == true)^{<l,true>}] >}$$

SR-NOT-F

$$\frac{\omega.\gamma(v_0) == false}{< !\, v_0 \,, \omega > \longrightarrow < \boldsymbol{true}, \omega[C: (v_0 == true)^{<l,false>}] >}$$

Structural Operational Semantics of Concrete Execution

S-Const-Seq:

$$\frac{v \text{ is constant}}{< v; e_1, s > \longrightarrow < e_{1,} s >}$$

S-Read_Field

$$\frac{v \neq null}{< v.f, s > \longrightarrow < \mathrm{s}(v, f), \mathrm{s} >}$$

S-Field-Assignment

$$\frac{v \ \neq null}{< v.f = v_1, s > \longrightarrow < v_1, s\,[(v, f) \mapsto v_1] >}$$

S-If

$$\frac{.}{< if\ true\ then\ e_1\ else\ e_2, s > \longrightarrow < e_1, s >}$$

$$\frac{.}{< if\ false\ then\ e_1\ else\ e_2, s > \longrightarrow < e_2, s >}$$

S-While

$$\frac{.}{< while_k\ e_0\ do\ e_1, s > \longrightarrow < if\ e_0\ then\ e_1; while_{k-1}\ e_0\ do\ e_1\ else\ skip, s >}\ \text{if } k > 0$$

S-New

$$\frac{\alpha\ \notin \text{HOs} \quad \bar{f}\ is\ a\ set\ of\ fields\ of\ C}{< new\ C, s > \longrightarrow < \alpha, s\left[\left(\alpha, \bar{f}\right) \mapsto \{null\}\right] >}$$

S-Method-Call

$$\frac{v \neq null \quad mbody(m, v) = \ \bar{x}.e}{< v.m(\bar{v}_1), s > \longrightarrow < e[\bar{v}_1/\bar{x}\ ,\ \ v/this],\ \ s >}$$

S-EQ-T

$$\frac{.}{< v_0 == v_0, s > \longrightarrow < true, s >}$$

S-EQ-F

$$\frac{v_0 != v_1}{< v_0 == v_1, s > \longrightarrow < false, s >}$$

S-AND-T

$$\frac{.}{< true\ \&\&\ e_1, s > \longrightarrow < e_1, s >}$$

S-AND-F

$$\frac{.}{< false\ \&\&e_1, s > \longrightarrow < false, s >}$$

S-OR-T

$$\frac{.}{< true\ ||\ v_1, s > \longrightarrow < \ true, s >}$$

S-OR-F

$$\frac{.}{< false\ ||\ v_1, s > \longrightarrow < v_1, s >}$$

S-NOT-T

$$\frac{.}{<\,!\,true\;,s\;> \longrightarrow <\,false, s\;>}$$

S-NOT-F

$$\frac{.}{<\,!\,false\;,s\;> \longrightarrow <\,true, s\;>}$$

Next we need prove the following lemmas and theorems.

Lemma 1: For any state s , $\boldsymbol{\omega}_s$=(<$\boldsymbol{\omega}_s.\boldsymbol{\gamma}$, $\boldsymbol{\omega}.\boldsymbol{\Gamma}$, $\boldsymbol{\omega}_s.C$>) is the corresponding concolic state, and e is an expression, if <e, $\boldsymbol{\omega}_s$ > ⟶ < e', $\boldsymbol{\omega}_s$'> then

i) If < e', $\boldsymbol{\omega}_s$' > is an error configuration ⊥ in concolic execution, then <s(e'), $s(\boldsymbol{\omega}_s'.\,\boldsymbol{\Gamma})$> is an error configuration ⊥ in the concrete execution.

ii) If $\boldsymbol{\omega}_s.C$ is satisfied by s then $\boldsymbol{\omega}_s'.C$ is satisfied by s.

iii) For any state σ, if $\boldsymbol{\omega}_s'.C$ is satisfied by σ, then $\boldsymbol{\omega}_s.C$ is satisfied by σ.

iv) For any state σ whose concolic state is < $\boldsymbol{\omega}_\sigma.\boldsymbol{\gamma}$, $\boldsymbol{\omega}.\boldsymbol{\Gamma}$, $\boldsymbol{\omega}_\sigma.C$ > such that $\boldsymbol{\omega}_\sigma.C$ = $\boldsymbol{\omega}_s.C$, if $\boldsymbol{\omega}_s'.C$ is satisfied by σ, then 1) <σ(e), $\sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma})$>⟶ <$\sigma$(e'), $\sigma(\boldsymbol{\omega}'.\boldsymbol{\Gamma})$ >; and 2) $\boldsymbol{\omega}_\sigma'.C$ = $\boldsymbol{\omega}_s'.C$

Proof: If < e', $\boldsymbol{\omega}_s$' > is an error heap configuration ⊥ then e' is one of the following forms: i) null.f; ii)null.f=$v_{\mathcal{A}}$; iii) null.m($\bar{v}_{\mathcal{A}}$); or iv) $while_0\; e_{0_{\mathcal{A}}} do\; e_{1_{\mathcal{A}}}$. Therefore, the corresponding concrete expression s(e') should be one of the above four forms and so <s(e'), $s(\boldsymbol{\omega}'.\,\boldsymbol{\Gamma})$> is an error heap configuration ⊥ in the concrete execution (no rules can be applied). We conclude the proof of i).

To prove ii), we observe that the rules of the structural operational semantics of concolic execution can be divided into two groups based on how *P'* is generated

1) P' ≡ P
2) P' ≡ P ∧ r

For group 1, the rules include SR-SymVar-Seq. Since $\boldsymbol{\omega}_s'.C$ is $\boldsymbol{\omega}_s.C$, the conclusion is valid.

For group 2, the rules include SR-Read_Field. *s* satisfies r since it is part of the premise of the rule.

To prove iii), we observe that $\boldsymbol{\omega}_s.C$ is part of $\boldsymbol{\omega}_s'.C$ and the conclusion is valid.

We prove iv) by the structural induction on the expression e. Let state *s* be an any state and its concolic state is $\boldsymbol{\omega}_s$.

1. Rule SR-SymVar-Seq i.e. e is v; e_1: we have $< v; e_1, \boldsymbol{\omega}_s > \longrightarrow < e_1, \boldsymbol{\omega}_s >$ and if $\boldsymbol{\omega}_s.\, C_s$ is satisfied by σ then we prove < $\sigma(v; e_1)$, $\sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma})$>⟶ <$\sigma(e_1)$, $\sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma})$>. Let state σ be such a state and its concolic state is < $\boldsymbol{\omega}_{\sigma}.\boldsymbol{\gamma}$, $\boldsymbol{\omega}.\boldsymbol{\Gamma}$, $\boldsymbol{\omega}_{\sigma}.$C > where $\boldsymbol{\omega}_{\sigma}.C = \boldsymbol{\omega}_s.C$ and $\boldsymbol{\omega}_{\sigma}.\boldsymbol{\gamma}=\boldsymbol{\sigma}$. Then <$\sigma(v; e_1)$, ($\boldsymbol{\omega}$.)>= <$\sigma(v)$; $\sigma(e_1)$, $\sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma})$>. Since v is a symbolic expression, $\sigma(v)$ is a constant too. By R-Cons-Seq, we have <$\sigma(v)$; $\sigma(e_1)$, ($\boldsymbol{\omega}.\boldsymbol{\Gamma}$)>⟶ <$\sigma(e_1)$, ($\boldsymbol{\omega}.\boldsymbol{\Gamma}$)>, and so this concludes the first part. Since the C part of the symbolic state is not changed in this transition, this concludes the second part.

2. Rule SR-Read_Field i.e. e is *v.f:* we have

   $< v.f, \boldsymbol{\omega}_s > \longrightarrow < \boldsymbol{\omega}.\boldsymbol{\Gamma}(v.f), \boldsymbol{\omega}_s[\mathrm{C}: (\boldsymbol{\omega}_s.\mathrm{C})(v! = null)^{<l,true>}] >$ and if $\boldsymbol{\omega}_s{}'.C$ is satisfied by σ and its concolic state is $< \boldsymbol{\omega}_{\boldsymbol{\sigma}}.\boldsymbol{\gamma}, \boldsymbol{\omega}.\boldsymbol{\Gamma}, \boldsymbol{\omega}_{\boldsymbol{\sigma}}.\mathrm{C} >$ where $\boldsymbol{\omega}_\sigma.C = \boldsymbol{\omega}_s.C$ and $\boldsymbol{\omega}_{\boldsymbol{\sigma}}.\boldsymbol{\gamma}=\boldsymbol{\sigma}$, then we prove $< \sigma(v.f),\ \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma}) > \longrightarrow < \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma}(v.f)), \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma}) >$. Since $\boldsymbol{\omega}_s{}'.C$ is satisfied by σ, $\sigma(v) \neq null$. By rule R-Read_Field, we have $< \sigma(v.f), \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma}) > \longrightarrow < \sigma(\mathrm{v,f}), \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma}) >$. Then, we have $\sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma}(v.f)) = \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma}(v).f) = \sigma(v, f)$. This concludes the first part. Since $\boldsymbol{\omega}_\sigma.C = \boldsymbol{\omega}_s.C$, both $\boldsymbol{\omega}_\sigma{}'.C$ and $\boldsymbol{\omega}_s{}'.C$ add $(v! = null)^{<l,true>}$ to the original ones. So this concludes the second part.

3. Rule SR-Field-Assignment ie.e is *v.f:* we have $< v.f = v_1, \boldsymbol{\omega}_s > \longrightarrow < v_1, \boldsymbol{\omega}_s[\boldsymbol{\gamma}: \boldsymbol{\omega}_s.\boldsymbol{\gamma}[(v,f) \longmapsto v_1], \boldsymbol{\Gamma}: \boldsymbol{\omega}.\boldsymbol{\Gamma}[(v,f) \longmapsto v_1], \mathrm{C}: (\boldsymbol{\omega}_s.\mathrm{C})(v! = null)^{<l,true>}] >$ and if $\boldsymbol{\omega}_s{}'.C$ is satisfied by σ and its concolic state is $< \boldsymbol{\omega}_{\boldsymbol{\sigma}}.\boldsymbol{\gamma}, \boldsymbol{\omega}.\boldsymbol{\Gamma}, \boldsymbol{\omega}_{\boldsymbol{\sigma}}.\mathrm{C} >$ where $\boldsymbol{\omega}_\sigma.C = \boldsymbol{\omega}_s.C$ and $\boldsymbol{\omega}_{\boldsymbol{\sigma}}.\boldsymbol{\gamma}=\boldsymbol{\sigma}$, then we prove $< \sigma(v.f = v_1), \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma}) > \longrightarrow < \sigma(v_1), \sigma(\boldsymbol{\omega}'_{\boldsymbol{\sigma}}.\Gamma) >$ where $\boldsymbol{\omega}'_{\boldsymbol{\sigma}} = \boldsymbol{\omega}_{\boldsymbol{\sigma}}[\boldsymbol{\gamma}: \boldsymbol{\omega}_{\boldsymbol{\sigma}}.\boldsymbol{\gamma}[(v,f) \longmapsto v_1], \boldsymbol{\Gamma}: \boldsymbol{\omega}.\boldsymbol{\Gamma}[(v,f) \longmapsto v_1], \mathrm{C}: (\boldsymbol{\omega}_{\boldsymbol{\sigma}}.\mathrm{C})(v\,! = null)^{<l,true>}]$. We have $< \sigma(v.f = v_1), \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma}) > = < \sigma(v.f) = \sigma(v_1), \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma}) > = < \sigma(v).f = \sigma(v_1), \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma}) >$. Since $\boldsymbol{\omega}_s{}'.C$ is satisfied by σ, $\sigma(v)$!=null, by rule R-Field-Assignment, we have $< \sigma(v).f = \sigma(v_1), \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma}) > \longrightarrow < \sigma(v_1), \sigma((\boldsymbol{\omega}.\boldsymbol{\Gamma})[(v,f) \longmapsto \sigma(v_1)] >$. Since $\sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma}(v)) = \sigma(v)$, we conclude the first part. Since $\boldsymbol{\omega}_\sigma.C = \boldsymbol{\omega}_s.C$, both $\boldsymbol{\omega}_\sigma{}'.C$ and $\boldsymbol{\omega}_s{}'.C$ add $(v! = null)^{<l,true>}$ to the original ones. So this concludes the second part.

4. Rule SR-While i.e. e is $while_k\ e_0\ do\ e_1$: we have $< while_k\ e_0\ do\ e_1, \boldsymbol{\omega}_s > \longrightarrow < if\ e_0\ then\ e_1; while_{k-1}\ e_0\ do\ e_1 else\ skip, \boldsymbol{\omega}_s >$ and if $\boldsymbol{\omega}_s{}'.C$ is satisfied

by σ and its concolic state is < $\boldsymbol{\omega_\sigma}.\boldsymbol{\gamma}$, $\boldsymbol{\omega}.\boldsymbol{\Gamma}$, $\boldsymbol{\omega_\sigma}$. C > where $\boldsymbol{\omega_\sigma}.C = \boldsymbol{\omega_s}.C$ and $\boldsymbol{\omega_\sigma}.\boldsymbol{\gamma}=\boldsymbol{\sigma}$, then we prove <$\sigma(while_k\ e_0\ do\ e_1), \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma})$>

⟶<$\sigma(if\ e_0\ then\ e_1; while_{k-1}\ e_0\ do\ e_1 else\ skip), \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma})$> assuming $k>0$.

<$\sigma(while_k\ e_0\ do\ e_1), \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma})$> =<$while_k\ \sigma(e_0)\ do\ \sigma(e_1), \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma})$>. By R-While, we have

<$while_k\ \sigma(e_0)\ do\ \sigma(e_1), \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma})$>⟶<

$if\ \sigma(e_0)\ then\ \sigma(e_1); while_{k-1}\ \sigma(e_0)\ do\ \sigma(e_1)\ else\ skip, \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma}) >$. So this concludes the first part. Since the C part of the symbolic state is not changed in this transition, this concludes the second part.

5. Rule SR-New, i.e. e is *new C*: we have $< new\ C, \boldsymbol{\omega}_s > \longrightarrow < \alpha, \boldsymbol{\omega}_s' >$

   $where\ \boldsymbol{\omega}_s' = \boldsymbol{\omega}_s\left[\boldsymbol{\gamma}: \boldsymbol{\omega}_s.\boldsymbol{\gamma}\left[\left(\alpha, \bar{f}\right) \mapsto \{null\}\right], \boldsymbol{\Gamma}: \boldsymbol{\omega}.\boldsymbol{\Gamma}\left[\left(\alpha, \bar{f}\right) \mapsto \{null\}\right]\right]$ and

   if $\boldsymbol{\omega}_s'.C$ is satisfied by σ and its concolic state is <$\boldsymbol{\omega_\sigma}.\boldsymbol{\gamma}$, $\boldsymbol{\omega}.\boldsymbol{\Gamma}$, $\boldsymbol{\omega_\sigma}$. C > where $\boldsymbol{\omega}_\sigma.C = \boldsymbol{\omega}_s.C$ and $\boldsymbol{\omega_\sigma}.\boldsymbol{\gamma}=\boldsymbol{\sigma}$, then we prove< σ (new C),

   $\sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma})$ >⟶< $\sigma(\alpha), \sigma(\boldsymbol{\omega}'.\boldsymbol{\Gamma})$>. Since $\alpha \notin$ HOs holds, we take the same α used by SR-New and we have < σ (new C), $\sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma})$ >=<new C,

   $\sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma})$ >⟶< α, $\sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma})\left[\left(\alpha, \bar{f}\right) \mapsto \{null\}\right]$ >=< $\sigma(\alpha)$,($\boldsymbol{\omega}'.\boldsymbol{\Gamma}$) >. So this concludes the first part. Since the C part of the symbolic state is not changed in this transition, this concludes the second part.

6. Rule SR-Method_Call, i.e. e is $v_0.m(\bar{v}_1)$: we have $< v.m(\bar{v}_1), \boldsymbol{\omega}_s > \longrightarrow <$

   $e[\bar{v}_1/\bar{x}\ ,\ \ v/this],\ \boldsymbol{\omega}_s' > where\ \boldsymbol{\omega}_s' = \boldsymbol{\omega}_s[C: (\boldsymbol{\omega}_s.C)(v! = null)^{<l,true>}]$

   and, if $\boldsymbol{\omega}_s'.C$ is satisfied by σ and its concolic state is < $\boldsymbol{\omega_\sigma}.\boldsymbol{\gamma}$, $\boldsymbol{\omega}.\boldsymbol{\Gamma}$, $\boldsymbol{\omega_\sigma}$. C > where $\boldsymbol{\omega}_\sigma.C = \boldsymbol{\omega}_s.C$ and $\boldsymbol{\omega_\sigma}.\boldsymbol{\gamma}=\boldsymbol{\sigma}$, then we prove

   $< \sigma(v.m(\bar{v}_1)), \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma})$>⟶$< \sigma(e[\bar{v}_1/\bar{x}\ ,\ \ v/this],\ \sigma(\boldsymbol{\omega}'.\boldsymbol{\Gamma}) >$. Since σ

satisfies $\boldsymbol{\omega}_s{}'.C$, $\sigma(v)$!= null holds. $< \sigma(v.m(\bar{v}_1)), \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma}) > =<$ $\sigma(v).m(\sigma(\bar{v}_1)), \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma}) > \longrightarrow < e[\sigma(\bar{v}_1)/\bar{x}\ ,\ \sigma(v)/this],\ \sigma(\boldsymbol{\omega}.\boldsymbol{\Gamma}) > \longrightarrow <$ $\sigma(e[\bar{v}_1/\bar{x}\ ,\ v/this],\ \sigma(\boldsymbol{\omega}'.\boldsymbol{\Gamma}) >$ ($\boldsymbol{\omega}'.\boldsymbol{\Gamma}$ is not updated in the target configuration). So this concludes the first part. Since $\boldsymbol{\omega}_\sigma.C$ = $\boldsymbol{\omega}_s.C$, both $\boldsymbol{\omega}_\sigma{}'.C$ and $\boldsymbol{\omega}_s{}'.C$ add $(v! = null)^{<l,true>}$ to the original ones. So this concludes the second part.

7. Rule SR-EQ-T, i.e. e is $v_0 == v_1$: we have $< v_0 == v_1, \boldsymbol{\omega}_s > \longrightarrow <$ $\boldsymbol{true}, \boldsymbol{\omega}_s{}' > where\ \boldsymbol{\omega}_s{}' = \boldsymbol{\omega}_s[C\colon \boldsymbol{\omega}_s.\mathrm{C}(v_0 == v_1)^{<l,true>}]$ and, if $\boldsymbol{\omega}_s{}'.C$ is satisfied by σ and its concolic state is $<\boldsymbol{\omega}_{\boldsymbol{\sigma}}.\boldsymbol{\gamma}, \boldsymbol{\omega}.\boldsymbol{\Gamma}, \boldsymbol{\omega}_{\boldsymbol{\sigma}}.\mathrm{C} >$ where $\boldsymbol{\omega}_\sigma.C$ = $\boldsymbol{\omega}_s.C$ and $\boldsymbol{\omega}_{\boldsymbol{\sigma}}.\boldsymbol{\gamma}=\boldsymbol{\sigma}$, then we prove $< \sigma(v_0 == v_1), \sigma(\omega.\Gamma) > \longrightarrow <$ $\sigma(\boldsymbol{true}), \sigma(\omega'.\Gamma) >$. $< \sigma(v_0 == v_1), \sigma(\omega.\Gamma) > =< \sigma(v_0) ==$ $\sigma(v_1), \sigma(\omega.\Gamma) >$. Since σ satisfies $\boldsymbol{\omega}_s{}'.C$, $\sigma(v_0) == \sigma(v_1)$ holds, by S-EQ-T, we have $< \sigma(v_0) == \sigma(v_1), \sigma(\omega.\Gamma) > \longrightarrow < true, \sigma(\omega.\Gamma) =<$ $\sigma(\boldsymbol{true}), \sigma(\omega'.\Gamma) >$ ($\omega'.\Gamma$ is not updated in the target configuration). So, this concludes the first part. Since $\boldsymbol{\omega}_\sigma.C$ = $\boldsymbol{\omega}_s.C$, both $\boldsymbol{\omega}_\sigma{}'.C$ and $\boldsymbol{\omega}_s{}'.C$ add $(v_0 == v_1)^{<l,true>}$to the original ones. So this concludes the second part. Same proof for the SR-EQ-F. In the same manner, we can prove SE-EQ-F.

8. Rule SR-AND-T, i.e. e is $v_0\&\&e_1$: we have $< v_0\&\&e_1, \boldsymbol{\omega}_s > \longrightarrow < e_1, \boldsymbol{\omega}_s{}' >$ $where\ \boldsymbol{\omega}_s{}' = \boldsymbol{\omega}_s[C\colon \boldsymbol{\omega}_s.C(v_0 == true)^{<l,true>}]$ and , if $\boldsymbol{\omega}_s{}'.C$ is satisfied by σ and its concolic state is $< \boldsymbol{\omega}_{\boldsymbol{\sigma}}.\boldsymbol{\gamma}, \boldsymbol{\omega}.\boldsymbol{\Gamma}, \boldsymbol{\omega}_{\boldsymbol{\sigma}}.\mathrm{C} >$ where $\boldsymbol{\omega}_\sigma.C$ = $\boldsymbol{\omega}_s.C$ and $\boldsymbol{\omega}_{\boldsymbol{\sigma}}.\boldsymbol{\gamma}=\boldsymbol{\sigma}$, then we prove $< \sigma(v_0\&\&e_1), \sigma(\omega.\Gamma) > \longrightarrow < \sigma(e_1), \sigma(\omega'.\Gamma) >$. Since σ satisfies $\boldsymbol{\omega}_s{}'.C$, $\sigma(v_0) == true$ holds, by S-AND-T, we have $< \sigma(v_0\&\&e_1), \sigma(\omega.\Gamma) > = < \sigma(v_0)\&\&\ \sigma(e_1), \sigma(\omega.\Gamma) > \longrightarrow <$

$\sigma(e_1), \sigma(\omega.\Gamma)$>=$< \sigma(e_1), \sigma(\omega'.\Gamma) >$ ($\omega'.\Gamma$ is not updated in the target configuration). So, this concludes the first part. Since $\boldsymbol{\omega}_\sigma.C$ = $\boldsymbol{\omega}_s.C$, both $\boldsymbol{\omega}_\sigma{}'.C$ and $\boldsymbol{\omega}_s{}'.C$ add $(v_0 == true)^{<l,true>}$to the original ones. So this concludes the second part. In the same manner, we can prove SE-OR-T and SE-OR-F.

9. Rule SR-NOT-T, i.e e is $!\,v_0$: we have $< !\,v_0\,, \boldsymbol{\omega}_s > \longrightarrow < \boldsymbol{false}, \boldsymbol{\omega}_s{}' >$ $where\ \boldsymbol{\omega}_s{}' = \boldsymbol{\omega}_s[C\colon \boldsymbol{\omega}_s.C(v_0 == true)^{<l,true>}]$ and, if $\boldsymbol{\omega}_s{}'.C$ is satisfied by σ and its concolic state is <$\boldsymbol{\omega_\sigma.\gamma}$, $\boldsymbol{\omega}$.**Γ**, $\boldsymbol{\omega_\sigma}$. C > where $\boldsymbol{\omega}_\sigma.C$ = $\boldsymbol{\omega}_s.C$ and $\boldsymbol{\omega_\sigma.\gamma}$=$\boldsymbol{\sigma}$, then we prove $\sigma(!\,v_0)\,, \sigma(\omega.\Gamma) > \longrightarrow^* < \sigma(\boldsymbol{false}), \sigma(\omega'.\Gamma) >$. Since σ satisfies $\boldsymbol{\omega}_s{}'.C,\ \ \sigma(v_0) == true$ holds, by S-NOT-T, we have $< \sigma(!\,v_0), \sigma(\omega.\Gamma) > = < !\ \sigma(v_0), \sigma(\omega.\Gamma) = < !\,true, \sigma(\omega.\Gamma) > \longrightarrow <$ $false, \sigma(\omega.\Gamma) >$ ($\omega'.\Gamma$ is not updated in the target configuration). So this concludes the first part. Since $\boldsymbol{\omega}_\sigma.C$ = $\boldsymbol{\omega}_s.C$, both $\boldsymbol{\omega}_\sigma{}'.C$ and $\boldsymbol{\omega}_s{}'.C$ add $(v_0 == true)^{<l,true>}$to the original ones. So this concludes the second part. In the same manner, we can prove SR-NOT-F.

Lemma 2: For any state s whose symbolic state is $\boldsymbol{\omega}_s$=< $\boldsymbol{\omega}_s.\boldsymbol{\gamma}$, $\boldsymbol{\omega}$.**Γ**, {}>, if <e, $\boldsymbol{\omega}_s$> $\longrightarrow^*$ < e',$\boldsymbol{\omega}_s{}'$> then

i) If < e',$\boldsymbol{\omega}_s{}'$> is an error concolic configuration ⊥, then <**s**(e'),**s**($\boldsymbol{\omega}'.\boldsymbol{\Gamma}$)> is an error heap configuration ⊥in the concrete execution

ii) State **s** satisfies $\boldsymbol{\omega}_s{}'$.C.

iii) For state $\boldsymbol{\sigma}$, if $\boldsymbol{\sigma}$ satisfies $\boldsymbol{\omega}_s{}'$.C and $\boldsymbol{\omega_\sigma}$=< $\boldsymbol{\omega}_\sigma.\boldsymbol{\gamma}$, $\boldsymbol{\omega}$.**Γ**, {}> is the corresponding concolic state, then $\boldsymbol{\omega}_s{}'$.C=$\boldsymbol{\omega_\sigma}{}'$.C.

iv) For state $\boldsymbol{\sigma}$, if $\boldsymbol{\sigma}$ satisfies $\boldsymbol{\omega}_s{}'$.C, then < $\boldsymbol{\sigma}$(e), $\boldsymbol{\sigma}$ > ⟶* <$\boldsymbol{\sigma}$(e'), $\boldsymbol{\sigma}(\boldsymbol{\omega}'.\Gamma)$ >

Proof: We prove i) by induction on the length n of the chain of symbolic reductions.

1. Base case, when n =0, i.e. < e',$\boldsymbol{\omega}_s{}'$> =<e, $\boldsymbol{\omega}_s$> is ⊥ . Then < e',$\boldsymbol{\omega}_s{}'$> should be one of the following four configurations:

- < null.f, $\boldsymbol{\omega}_s{}'$>
- < null.f=e, $\boldsymbol{\omega}_s{}'$>
- < null.m($\bar{v}$), $\boldsymbol{\omega}_s{}'$>
- < $while_0\ e_0\ do\ e_1$, $\boldsymbol{\omega}_s{}'$>

Therefore <**s**(e'),**s**($\boldsymbol{\omega}'.\boldsymbol{\Gamma}$)> should be an error heap configuration too.

2. Induction step: assume i) is valid for any n = k -1. We consider the following two cases:

   - If <e, $\boldsymbol{\omega}_s$ > $\longrightarrow^{k-1}$ < e', $\boldsymbol{\omega}_s{}'$> and < e', $\boldsymbol{\omega}_s{}'$> is ⊥, then <**s**(e'),**s**($\boldsymbol{\omega}'.\boldsymbol{\Gamma}$)> is ⊥ by induction. So there is no kth reduction from < e', $\boldsymbol{\omega}_s{}'$>.
   - If <e, $\boldsymbol{\omega}_s$ > $\longrightarrow^{k-1}$ < e', $\boldsymbol{\omega}_s{}'$> and < e', $\boldsymbol{\omega}_s{}'$> is not ⊥ so we have a rule to be applied and the next configuration is < e', $\boldsymbol{\omega}_s{}'$> ⟶< e", $\boldsymbol{\omega}_s$"> and < e", $\boldsymbol{\omega}_s$"> is an error configuration. Then similar to the base case, < e", $\boldsymbol{\omega}_s$"> has one of the four error configurations. Therefore, <**s**(e"),**s**($\boldsymbol{\omega}''.\boldsymbol{\Gamma}$)> should be an error heap configuration too

We prove ii) by induction on the length n of the chain of symbolic reductions.

1. Base case when n =0, i.e. <e, $\boldsymbol{\omega}_s$ > where $\boldsymbol{\omega}$.C={}. State *s* satisfies $\boldsymbol{\omega}_s$.C.
2. Induction step: assume ii) is valid for any n=k-1. We will prove for the state *s*, if <e, $\boldsymbol{\omega}_s$ > $\longrightarrow^{k}$ < e'', $\boldsymbol{\omega}_s$"> then *s* satisfies $\boldsymbol{\omega}$".C. <e, $\boldsymbol{\omega}_s$ > $\longrightarrow^{k}$ < e'', $\boldsymbol{\omega}_s$">

can be broken into <e, $\boldsymbol{\omega}_s$> $\longrightarrow^{k-1}$< e', $\boldsymbol{\omega}_s$'> ⟶ < e'', $\boldsymbol{\omega}_s$''>. By induction, *s* satisfies $\boldsymbol{\omega}_s$'.C. By lemma 1 (ii), *s* satisfies $\boldsymbol{\omega}$''.C.

We prove iii) by induction on the length n of the chain of symbolic reductions

1. Base case when n = 0, i.e. <e, $\boldsymbol{\omega}_s$ >=< e',$\boldsymbol{\omega}_s{}'$> where $\boldsymbol{\omega}_s$.C={}. Then for all state $\boldsymbol{\sigma}$, we have $\boldsymbol{\omega}_s{}'$.C=$\boldsymbol{\omega}_{\boldsymbol{\sigma}}{}'$.C.
2. Induction Step: assume iii) hold for any n = k-1 . We will prove for any state $\boldsymbol{\sigma}$, if $\boldsymbol{\sigma}$ satisfies $\boldsymbol{\omega}_s{}'$.C, then < $\boldsymbol{\sigma}$(e), $\boldsymbol{\sigma}$ > ⟶* <$\boldsymbol{\sigma}$(e'), $\boldsymbol{\sigma}(\boldsymbol{\omega}'.\Gamma)$ >. <e, $\boldsymbol{\omega}_s$> $\longrightarrow^{k}$ < e', $\boldsymbol{\omega}_s$'> can be broken into <e, $\boldsymbol{\omega}_s$> $\longrightarrow^{k-1}$<e'', $\boldsymbol{\omega}_s$''> ⟶ <e', $\boldsymbol{\omega}_s$'>. Since $\boldsymbol{\sigma}$ satisfies $\boldsymbol{\omega}_s{}'$.C, by lemma 1 (iii) $\boldsymbol{\sigma}$ satisfies $\boldsymbol{\omega}_s$''.C. By induction for n=k-1, $\boldsymbol{\omega}_s$".C=$\boldsymbol{\omega}_\sigma$".$C$ holds. Next by lemma 1 (iv)(2) and <e'', $\boldsymbol{\omega}_s$''> ⟶ <e', $\boldsymbol{\omega}_s$'>, we have $\boldsymbol{\omega}_s{}'$.C=$\boldsymbol{\omega}_{\boldsymbol{\sigma}}{}'$.C. This concludes the case.

We prove iv) by induction on the length n of the chain of symbolic reductions

1. Base case when n = 0, i.e. <e, $\boldsymbol{\omega}_s$ > =< e',$\boldsymbol{\omega}_s{}'$> where $\boldsymbol{\omega}$.C={}. Then for all state $\boldsymbol{\sigma}$, we have < (e), $\boldsymbol{\sigma}$ > = <$\boldsymbol{\sigma}$(e'), $\boldsymbol{\sigma}(\boldsymbol{\omega}'.\Gamma)$ > immediately.
2. Induction Step: assume iv) hold for any n = k-1 . We will prove for any state $\boldsymbol{\sigma}$, if $\boldsymbol{\sigma}$ satisfies $\boldsymbol{\omega}_s{}'$.C, then < $\boldsymbol{\sigma}$(e) , $\boldsymbol{\sigma}$ > ⟶* <$\boldsymbol{\sigma}$(e'), $\boldsymbol{\sigma}(\boldsymbol{\omega}'.\Gamma)$ >. <e, $\boldsymbol{\omega}_s$> $\longrightarrow^{k}$ < e', $\boldsymbol{\omega}_s$'> can be broken into <e, $\boldsymbol{\omega}_s$> $\longrightarrow^{k-1}$<e'', $\boldsymbol{\omega}_s$''> ⟶ <e', $\boldsymbol{\omega}_s$'>. Since $\boldsymbol{\sigma}$ satisfies $\boldsymbol{\omega}_s{}'$.C, by lemma 1 (iii) $\boldsymbol{\sigma}$ satisfies $\boldsymbol{\omega}_s$''.C. By induction for n=k-1, < $\boldsymbol{\sigma}$(e), $\boldsymbol{\sigma}$ > ⟶* <$\boldsymbol{\sigma}$(e"), $\boldsymbol{\sigma}(\boldsymbol{\omega}".\Gamma)$> holds. Next by lemma 1 (iv)(1), from <e'', $\boldsymbol{\omega}_s$''> ⟶ <e', $\boldsymbol{\omega}_s$'> and $\boldsymbol{\omega}_s$".C=$\boldsymbol{\omega}_\sigma$".$C$ (using lemma 2(iii)), we have <$\boldsymbol{\sigma}$(e"), $\boldsymbol{\sigma}(\boldsymbol{\omega}".\Gamma)$ >⟶<$\boldsymbol{\sigma}$(e'), $\boldsymbol{\sigma}(\boldsymbol{\omega}'.\Gamma)$ >. This concludes the case.

This concludes our proof.

Lemma 3: For any state **s** whose symbolic state is $\boldsymbol{\omega}_s$=<$\boldsymbol{\omega}_s.\boldsymbol{\gamma}$, $\boldsymbol{\omega}.\boldsymbol{\Gamma}$, {}>, assume <$e_1; e_2$, $\boldsymbol{\omega}_s$> $\longrightarrow^*$ < e_2,$\boldsymbol{\omega}_{s_1}$ > $\longrightarrow^*$ <e,$\boldsymbol{\omega}_{s_2}$> and let pc_1 be $\boldsymbol{\omega}_{s_1}$.C and pc_2 be $\boldsymbol{\omega}_{s_2}$.C-$\boldsymbol{\omega}_{s_1}$.C. For any state σ whose concolic state is $\boldsymbol{\omega}_\sigma$=<$\boldsymbol{\omega}_\sigma.\boldsymbol{\gamma}$, $\boldsymbol{\omega}.\boldsymbol{\Gamma}$,{}> such that σ satisfies $\boldsymbol{\omega}_{s_1}$.C. If $pc_1 \Rightarrow pc_2$ is a tautology, then $\boldsymbol{\omega}_{s_2}$.C = $\boldsymbol{\omega}_{\sigma_2}$.C.

Proof: For any state **s** whose symbolic state is $\boldsymbol{\omega}_s$=<$\boldsymbol{\omega}_s.\boldsymbol{\gamma}$, $\boldsymbol{\omega}.\boldsymbol{\Gamma}$, {}>, assume <$e_1; e_2$, $\boldsymbol{\omega}_s$> $\longrightarrow^*$ < e_2,$\boldsymbol{\omega}_{s_1}$ >. Then any state σ whose concolic state is $\boldsymbol{\omega}_\sigma$=<$\boldsymbol{\omega}_\sigma.\boldsymbol{\gamma}$, $\boldsymbol{\omega}.\boldsymbol{\Gamma}$,{}> such that σ satisfies $\boldsymbol{\omega}_{s_1}$.C, we have $\boldsymbol{\omega}_{s_1}$.C = $\boldsymbol{\omega}_{\sigma_1}$.C by lemma 2 iii). Since $pc_1 \Rightarrow pc_2$ is a tautology and σ satisfies $\boldsymbol{\omega}_{s_1}$.C(=$pc_1$), σ satisfies pc_2. So σ satisfies $\boldsymbol{\omega}_{s_2}$.C. Consider <$e_1; e_2$, $\boldsymbol{\omega}_s$> $\longrightarrow^*$<e,$\boldsymbol{\omega}_{s_2}$> and σ satisfies $\boldsymbol{\omega}_{s_2}$.C, we have $\boldsymbol{\omega}_{s_2}$.C = $\boldsymbol{\omega}_{\sigma_2}$.C by lemma 2 (iii).

Theorem 1: At the end of the execution of *Concolic_CCUJ* on the expression α .m;*α .m_{post}*, the set **B** should satisfy the following properties:

- **i.** **B** contains the finite number of path constraints
- **ii.** **B** contains all path constraints in α .m;$\alpha .m_{post}$

Proof: We first prove i). We add a new path constraint to ***B*** after the concolic execution of α .m;$\alpha .m_{post}$. Note k is set for the execution bound of a loop structure and so the number of path conditions in a path constraint is bounded. And each path condition has at most two branches, either true or false. So, the number of path constraints in **B** is finite.

We next prove ii). Assume **B** misses one path constraint which is (l_1, b_1) $(l_2, b_2)\ldots$ $(l_i, b_i)\ldots$ (l_n, b_n) where $l_i \in LabelName, b_i$ is either *true* or *false*, and i={1,2,…n}. Then we find the longest path constraint lpc in **B** that shares the longest prefix of the missing path constraint. Formally, lpc is (l_1, b_1) (l_2, b_2)… $(l_i, !\, b_i)$… (l_k, b_k) where *i* is the largest number from 1 to n. Note algorithm *Conc_Symb_Exe* pops a top element from *stack_pc* if 1) the top element has been flipped, 2) the path constraint consisting of *stack_pc[1]⋀..⋀!stack_pc[top]⋀Pre* is not satisfiable, or the path constraint consisting of *stack_pc[1]⋀..⋀stack_pc[i]* is produced by method α .m, denoted as pc_1 and the path constraint consisting of *stack_pc[i+1]⋀..⋀stack_pc[top]* is produced by method α *.m*$_{post}$, denoted as pc_2, and $pc_1 \Rightarrow pc_2$ is tautology and the top element is any of path conditions from pc_2. Now back to our case. If pc_i's *flipped* value is *true*, from lpc in **B,** then there are two cases where pc_i's *flipped* value: 1) $(l_1, b_1) \wedge (l_2, b_2) \wedge \ldots \wedge (l_i, b_i)$ ∧*Pre* is satisfiable, 2) pc_i is not flappable, i.e. the path condition from one of SR-Read-Field,SR-Field-Assignment,and SR-Method-Call. Since (l_1, b_1) (l_2, b_2)… (l_i, b_i) is part of the missing path condition, we ensure (l_1, b_1) (l_2, b_2)… (l_i, b_i) is satisfiable. So, it should be in **B** and this contradicts our assumption that lpc is the longest path constraint in **B.** If (l_1, b_1) (l_2, b_2)… (l_i, b_i) is not satisfiable combined with *Pre*, then it contradicts the assumption of the missing path constraint. If (l_1, b_1) (l_2, b_2)… (l_{i-1}, b_{i-1}), denoted as pc_1, in lpc is the path constraint of α .m and $(l_i, !\, b_i)$… (l_k, b_k), denoted as pc_2, in lpc is the path constraint of α *.m*$_{post}$ and $pc_1 \Rightarrow pc_2$ is a tautology, then it contradicts lemma3 which shows both path constraints in α *.m*$_{post}$ are the same.

Theorem 2: Assume *Concolic_CCUJ* can terminate. If *Concolic_CCUJ* returns true if and only if for any state **s**, if **s** satisfies *Pre*(denoted as Pre(s)), then (<α *.m,* **s** >⟶* < e',

$\boldsymbol{s_m}$> (denoted as Method(**s**))) ∧ (< $\alpha.m_{post}$, $\boldsymbol{s_m}$ >⟶*<*true*, $\boldsymbol{s_{post}}$> (denoted as Post(**s**))) is true.

Proof: We first prove Concolic_CCUJ returns true only if for any state **s**, if **s** satisfies *Pre*(denoted as Pre(*s*)), then (<*α .m,* **s** >⟶* < e', $\boldsymbol{s_m}$> (denoted as Method(**s**))) ∧ (< $\alpha.m_{post}$, $\boldsymbol{s_m}$ >⟶*<*true*, $\boldsymbol{s_{post}}$> (denoted as Post(**s**))) is true.

Assume Concolic_CCUJ returns true. Notice S is initialized to Pre at line 1. At line 2, there are two possibilities for the condition for the while statement:

i) If S is not satisfiable, then there is no concrete state satisfying S. Then, the conclusion is valid.

ii) If S is satisfiable we need to prove that $\boldsymbol{\mathcal{J}}$ is a loop invariant.

- If S is satisfiable and let *s* be such a state that statisfies *S*, then Pre(*s).*
- If S is not satisfiable, then for any state *s* if Pre(s)⇒Method(s) ∧ Post(**s**)

The invariant holds at the beginning of the while loop statement since it is under the false branch of the if statement at line 2. Assume we find a state *s*, whose corresponding concolic state is < $\boldsymbol{\omega.\gamma}$, $\boldsymbol{\omega.\Gamma}$, {}> and at line 6, the concolic execution results in < α .m, < $\boldsymbol{\omega.\gamma}$, $\boldsymbol{\omega.\Gamma}$, {}>> ⟶* <e', <$\boldsymbol{\omega}_m.\boldsymbol{\gamma}$, $\boldsymbol{\omega}_m.\boldsymbol{\Gamma}$,$\boldsymbol{\omega}_m.C$>> . Since Concolic_CCUJ returns true, then the condition of line 7 is not true so line 8 is skipped. By Lemma 2, we have for any σ, including *s*, such that σ satisfies $\boldsymbol{\omega}_m.C$, <α .m, $\sigma(\boldsymbol{\omega.\Gamma})$> ⟶* <σ(e'), $\sigma(\boldsymbol{\omega}_m.\boldsymbol{\Gamma})$>

holds. At line 9, the symbolic execution continues to call m_{post}using $<$ $\boldsymbol{\omega}_m.\boldsymbol{\gamma}, \boldsymbol{\omega}_m.\boldsymbol{\Gamma}, \phi >$ as an initial concolic state. Again line 10 cannot be true; otherwise Concolic_CCUJ returns false. So, by Lemma 2, from < α . m_{post}, $<$ $\boldsymbol{\omega}_m.\boldsymbol{\gamma}, \boldsymbol{\omega}_m.\boldsymbol{\Gamma}, \phi$ >> $\longrightarrow^*$ <***true***, $\boldsymbol{\omega}_{post}$> we have $< \alpha.m_{post},\ \sigma(\boldsymbol{\omega}_m.\boldsymbol{\Gamma}) >$ $\longrightarrow^*$ < σ(**true**), σ($\boldsymbol{\omega}_{post}.\boldsymbol{\Gamma}$)>. Using the transitive property of ⟶ and lemma 2, for any σ such that σ satisfies $\boldsymbol{\omega}_m.C$ and $\boldsymbol{\omega}_{post}.C$, we have Pre($\sigma$) ⟹Method($\boldsymbol{\sigma}$) ∧ Post($\boldsymbol{\sigma}$) holds at line 13. Since there is no change Pre(σ), Method($\boldsymbol{\sigma}$), Post($\boldsymbol{\sigma}$) at line 13, we still have Pre(σ) ⟹Method($\boldsymbol{\sigma}$) ∧ Post($\boldsymbol{\sigma}$). At line 14, there are two scenarios:

1). if $\boldsymbol{\omega}_m$.C ⇒ $\boldsymbol{\omega}_{post}$.C is *not a* tautology then there exists state σ such that σ satisfies $\boldsymbol{\omega}_m$. C but not $\boldsymbol{\omega}_{post}$. C; so we need to explore a different path in m_{post}. Thus, Concolic_CCUJ tries to find a new_pc. Since there is no execution of methods α .m and α . m_{post}, Pre(σ) ⟹Method($\boldsymbol{\sigma}$) ∧ Post($\boldsymbol{\sigma}$) still holds at lines 14, and 15.

2). If $\boldsymbol{\omega}_m$.C ⇒ $\boldsymbol{\omega}_{post}$.C is a tautology, then for any state σ if σ satisfies $\boldsymbol{\omega}_m$.C then σ satisfies $\boldsymbol{\omega}_{post}$.C. By Lemma 3, for all these states have the same $\boldsymbol{\omega}_{post}$.C. In this case, we need to explore a different path in α .m. Since there is no execution of methods α .m and α.m_{post}, Pre(σ) ⟹Method($\boldsymbol{\sigma}$) ∧ Post($\boldsymbol{\sigma}$) holding at line 13 still holds at line 16,17, and 18.

At line 19, S is updated to *Pre* ∧ *new_pc*. There are two possibilities for new_pc. First, if new_pc is not false, then S is satisfiable because method backtracking pops all the top elements whose path constraint combined with

Pre is not satisfiable. Since *Pre* is part of S, so any state s satisfies S, s should satisfy *Pre*. If new_pc is false, then S is not satisfiable. Then, from the previous execution, we show for any state s we have Pre(s)⇒Method(s) ∧ Post(**s**). So the invariant holds at the end of the while statement.

Next, if the while statement can terminate, namely, S is not satisfiable, then for any state s, Pre(s)⇒Method(s) ∧ Post(**s**) holds. We conclude the "only if" part proof.

Next we prove the "if" part.

Assume Concolic_CCUJ returns false if there exists a state *s* such that Pre(s)⇒Method(s) ∧ Post(**s**) holds. There are two scenarios that return false:

i) If the condition is false at line 7, then <e', $\boldsymbol{\omega_m}$> in < α .m, $\boldsymbol{\omega}$>$\longrightarrow^*$<e', $\boldsymbol{\omega_m}$> is an error heap configuration . By Lemma 2, <e', $\boldsymbol{\omega}_m.\boldsymbol{\gamma}$> is an error heap configuration and this contradicts the following fact: Pre(*s*) ⟹Method(s) ∧ Post(*s*) where Method(s) is false. Otherwise, the condition is true at line 4. Then,

ii) If condition at line 10 is false and it means $<\ \alpha\,.\,\mathrm{m}_{post}, <\boldsymbol{\omega}_m.\gamma, \boldsymbol{\omega}_m.\Gamma, \phi >> \longrightarrow^*$ <***false***, $\boldsymbol{\omega}_{post}$ >. By Lemma 2, $<\ s(\alpha\,.\,\mathrm{m}_{post}), \mathrm{s}(\boldsymbol{\omega}_m.\gamma) > \longrightarrow^*$ <s(***false***), $s(\boldsymbol{\omega}_{post})$ > holds and *s(**false**)=false* which contradicts the fact: Pre(*s*) ⟹Method(s) ∧ Post(*s*) where Post(*s*) is false.

This concludes our proof.

Theorem 3: Algorithm Concolic_CCUJ should terminate.

Proof: To prove the termination of Algorithm Concolic_CCUJ, we need to prove that the while statement can terminate. By Theorem 1, we know **B** includes a finite number of path constraints of α .m;*α .m*$_{post}$. Next, we will show that during each loop execution, a new path constraint according to new_pc is added to **B**. If this is true, then number of the execution of the while statement is equal to the number of elements in **B**.

Method backtracking is the only place to return a path condition. When method backtracking returns a path condition other than *false*, it must satisfy the following conditions:

- stack_pc[top].flipped=false;
- the path condition consisting of stack_pc[1]∧..∧!stack_pc[top]∧Pre is satisfiable;
- top>0

Also, a path constraint including path conditions stack_pc[1],…,stack_pc[top] is in **B.** In this case, we can guarantee that there are no path constraints in **B** that have the path conditions stack_pc[1],.., stack_pc[top-1] and !stack_pc[top] as their prefix. To prove this, we assume there exists a path constraint in **B** that has the path conditions stack_pc[1],.., stack_pc[top-1] and !stack_pc[top] as its prefix. Then, stack_pc[top].flippped=true since another path constraint

including path conditions stack_pc[1],…,stack_pc[top] has been in **B.** This contradicts the first condition which is stack_pc[top].flipped=false. So, the path constraint new_pc consisting of stack_pc[1]∧..∧!stack_pc[top] is not a prefix of any path constraint in **B**. Thus, the Concolic_CCUJ path constraints generated by new_pc is new in **B**.

BIBLIOGRAPHY

[1] Jonathan Corbet, Greg Kroah, and Amanda McPherson. Linux kernel development. Technical report, 2010.

[2] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, 2002.

[3] N.G. Leveson and C.S. Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[4] Alain Abran, James Moore, Pierre Bourque, Robert Dupuis, and Leonard Tripp. Guide to the software engineering body of knowledge. Technical report, 2004.

[5] Daniel Dvorak et al. Nasa study on flight software complexity. *NASA office of chief engineer*, 2009.

[6] Joaquin Miller and Jishnu Mukerji. Mda guide version 1.0.1. Technical report, 2003.

[7] OMG. Unified modeling language: Superstructure, version 2.1.2. Technical report, 2007.

[8] OMG. Object constraint language. Technical report, 2006.

[9] Paul Ammann and Jeff Offutt. *Introduction to Software Testing.* Cambridge University Press, 2008.

[10] Darko Marinov, Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Martin Rinard. An evaluation of exhaustive testing for data structures. Technical report, MIT Computer Science and Artificial Intelligence Laboratory Report MIT -LCS-TR-921, 2003.

[11] Paul T. Darga and Chandrasekhar Boyapati. Efficient software model checking of data structure properties. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 363–382. ACM, 2006.

[12] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[13] Frank Truyen. The fast guide to model driven architecture. Technical report, 2006.

[14] Rational software architect, 2014.

[15] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[16] Koushik Sen. Concolic testing and constraint satisfaction. In *Proceedings of the 14th International Conference on Theory and Application of Satisfiability Testing*, SAT'11, pages 3–4, Berlin, Heidelberg, 2011. Springer-Verlag.

[17] Bowen Alpern and FredB. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[18] E. Allen Emerson. Handbook of theoretical computer science (vol. b). chapter Temporal and Modal Logic, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990.

[19] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):21, 2009.

[20] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 317–320. ACM, 1999.

[21] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2009.

[22] M. Khandai, A.A. Acharya, and D.P. Mohapatra. A novel approach of test case generation for concurrent systems using uml sequence diagram. In *Electronics Computer Technology (ICECT), 2011 3rd International Conference on*, volume 1, pages 157–161, April 2011.

[23] Bao-lin Li, Zhi-shu Li, Li Qing, and Yan-Hong Chen. Test case automate generation from uml sequence diagram and ocl expression. In *Computational Intelligence and Security, 2007 International Conference on*, pages 1048–1052, Dec 2007.

[24] M. Sarma, D. Kundu, and R. Mall. Automatic test case generation from uml sequence diagram. In *Advanced Computing and Communications, 2007. ADCOM 2007. International Conference on*, pages 60–67, Dec 2007.

[25] M. Beyer, W. Dulz, and Fenhua Zhen. Automated ttcn-3 test case generation by means of uml sequence diagrams and markov chains. In *Test Symposium, 2003. ATS 2003. 12th Asian*, pages 102–105, Nov 2003.

[26] Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong, and Zheng Guoliang. Generating test cases from uml activity diagram based on gray-box method. In *Software Engineering Conference, 2004. 11th Asia-Pacific*, pages 284–291, Nov 2004.

[27] Hang Zhou, Zhiqiu Huang, and Yi Zhu. Polymorphism sequence diagrams test data automatic generation based on ocl. In *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 1235–1240, Nov 2008.

[28] Clay E. Williams. Towards a test-ready meta-model for use cases. In *Workshop of the pUML-Group held together with the UML 2001 on Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists*, pages 270–287. GI, 2001.

[29] E.G. Cartaxo, F.G.O. Neto, and P.D.L. Machado. Test case generation by means of uml sequence diagrams and labeled transition systems. In *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, pages 1292–1297, Oct 2007.

[30] R Anbunathan and Anirban Basu. Dataflow test case generation from uml class diagrams. In *Computational Intelligence and Computing Research (ICCIC), 2013 IEEE International Conference on*, pages 1–9, Dec 2013.

[31] Aynur Abdurazik and Jeff Offutt. Using uml collaboration diagrams for static checking and test generation. pages 383–395, 2000.

[32] Abhik Roychoudhury, Ankit Goel, and Bikram Sengupta. Symbolic message sequence charts. *ACM Trans. Softw. Eng. Methodol.*, 21(2):12:1–12:44, March 2012.

[33] Key project: Integrated deductive software design, 2014.

[34] A.B.H. Ali, F. Boufares, and A. Abdellatif. Checking constraints consistency in uml class diagrams. In *Information and Communication Technologies, 2006. ICTTA '06. 2nd*, volume 2, pages 3599–3604, 2006.

[35] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik. Consistency checking of conceptual models via model merging. In *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pages 221–230, Oct 2007.

[36] Muhammad Usman, A. Nadeem, Tai hoon Kim, and Eun suk Cho. A survey of consistency checking techniques for uml models. In *Advanced Software Engineering and Its Applications, 2008. ASEA 2008*, pages 57–62, Dec 2008.

[37] Xiaoshan Li, Zhiming Liu, and Jifeng He. Consistency checking of uml requirements. In *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, pages 411–420, June 2005.

[38] Yang Liu, Yafen Li, and Pu Wang. Design and implementation of automatic generation of test cases based on model driven architecture. In *Information Technology and Computer Science (ITCS), 2010 Second International Conference on*, pages 344–347, July 2010.

[39] Yann-Gael Gueheneuc and Herve Albin-Amiot. Recovering binary class relationships: putting icing on the uml cake. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 301–314. ACM, 2004.

[40] Ana Milanova. Precise identification of composition relationships for uml class diagrams. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 76–85. ACM, 2005.

[41] David Akehurst, Gareth Howells, and Klaus M. Maier. Implementing associations: Uml 2.0 to java 5. *Software and Systems Modeling*, 6:3–35, 2007.

[42] Franck Barbier, Brian Henderson-Sellers, Annig Le Parc-Lacayrelle, and Jean-Michel Bruel. Formalization of the whole-part relationship in the unified modeling language. *IEEE Trans. Softw. Eng.*, 29:459–470, 2003.

[43] Grady Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA„ 2004.

[44] Sat4j model checker, 2010.

[45] Eclipse modeling tools - mdt, 2014.

[46] Fujaba tool suite 5, 2010.

[47] Java bytecode manipulation and analysis framework, 2014.

[48] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133. ACM, 2002.

[49] Wuwei Shen, Kun Wang, and Alexander Egyed. An efficient and scalable approach to correct class model refinement. *IEEE Trans. Softw. Eng.*, 35:515–533, 2009.

[50] Uml2 project, 2010.