The Journal of Systems and Software 107 (2015) 71–92

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A UML model-based approach to detect infeasible paths

Debasish Kundu [a], Monalisa Sarma [b], Debasis Samanta [a,*]

[a] School of Information Technology, Indian Institute of Technology Kharagpur, Kharagpur, West Bengal 721302, India
[b] Reliability Engineering Center, Indian Institute of Technology Kharagpur, Kharagpur, West Bengal 721302, India

ARTICLE INFO

Article history:
Received 26 March 2014
Revised 18 March 2015
Accepted 1 May 2015
Available online 14 May 2015

Keywords:
Infeasible path detection
Model-based analysis
Softwaretesting
UML sequence diagram

ABSTRACT

UML model-based analysis is gaining wide acceptance for its cost effectiveness and lower overhead for processing compared to code-based analysis. A possible way to enhance the precision of the results of UML based analysis is by detecting infeasible paths in UML models. Our investigation reveals that two interaction patterns called *Null Reference Check* (NLC) and *Mutually Exclusive* (MUX) can cause a large number of infeasible paths in UML sequence diagrams. To detect such infeasible paths, we construct a graph model (called SIG), generate MM paths from the graph model, where an MM path refers to an execution sequence of model elements from the start to end of a method scope. Subsequently, we determine infeasibility of the MM paths with respect to MUX and NLC patterns. Our proposed model-based approach is useful to help exclude generation of test cases and test data for prior-detected infeasible paths, refine test effort estimation, and facilitate better test planning in the early stages of software development life cycle.

1. Introduction

To reduce development and maintenance costs of modern softwares, developers usually advocate the use of object-oriented analysis and design paradigm (Larman, 2002). To facilitate the use of standardized notations for object-oriented analysis and design, Object Management Group (OMG; OMG UML Specification, 2011) introduced Unified Modeling Language (UML; Booch et al., 2005; Pilone and Pitman, 2005). UML being a visual modeling language, supports modeling different views of software and has found wide acceptance among software practitioners. The UML design models which are primarily constructed to serve as software design specifications, have found applications in many software engineering activities such as testing (Ali et al., 2007; Bandyopadhyay and Ghosh, 2009; Briand and Labiche, 2002; Gallagher et al., 2006; Kundu et al., 2009), traceability analysis, maintenance etc. In this regard, we may note that UML model-based analysis often use message paths (Binder, 1999; Fraikin and Leonhardt, 2002; Nayak and Samanta, 2009; Pilskalns et al., 2003; Sarma et al., 2007) object-states (Ali et al., 2007; Bandyopadhyay and Ghosh, 2009; Gallagher et al., 2006), which are explicitly captured in UML sequence and statechart diagrams.

In the context of integration testing of object-oriented systems, all interaction paths need to be tested in different object-states to uncover integration faults. In practical situations, object-oriented softwares have a large number of interaction paths (which are basically message paths) and hence, testing all of them in different object-states may not be feasible due to release pressure from stakeholders. Further complicating situation, determination of message paths requires static/dynamic analysis of huge amount of code artifacts, adding another level overhead for test automation. This has motivated us to consider UML models which is the *de facto* standard in industry for modeling object-oriented systems. Further, UML model based message path coverage helps the testers to detect critical integration faults as reported in the literature (Ali et al., 2007).

Existing work (Ali et al., 2007; Bandyopadhyay and Ghosh, 2009; Nayak and Samanta, 2009; Pilskalns et al., 2003; Sarma et al., 2007) on UML based test coverage implicitly assume that every path generated from various UML models such as activity, sequence, communication diagrams is feasible. All these work ignore the presence of infeasible paths which cannot be executed with any test data (Ngo and Tan, 2007; 2008; Souter and Pollock, 2001). Several studies report that a significant number of paths in object-oriented programs can be infeasible (Ngo and Tan, 2007; Souter and Pollock, 2001). This has driven us to examine path infeasibility in UML models in the context of integration testing of object-oriented systems.

We find that two interaction patterns called *Null Reference Check* (NLC) and *Mutually Exclusive* (MUX) cause a large number of the infeasible paths in UML sequence diagrams. Note that an NLC interaction pattern consists of modeling both *null* and *non-null* return values for a return variable in one method and checking *nullify test* of the return variable in another method. On the other hand, MUX interaction

* Corresponding author. Tel.: +91 3222 282334, fax: +91 3222 255306.
E-mail address: debasis.samanta.iitkgp@gmail.com, dsamanta@iitkgp.ac.in (D. Samanta).

http://dx.doi.org/10.1016/j.jss.2015.05.007
0164-1212/

pattern consists of a set of actions (e.g. sending a message, defining a variable etc.) such that only one action can be executed at a time. The main contributions of this paper are as follows: finding two infeasibility patterns namely NLC and MUX, designing algorithms to detect infeasible paths arising out of these two patterns, and investigating effects of two patterns on the number of infeasible paths and test effort estimation.

To detect infeasible paths in UML sequence diagrams for MUX and NLC patterns, we first construct a graph model (called *Sequence Integration Graph*). Note that our graph model subsumes control flow graph and additionally, contains method scope information of interactions. This graph model is then used to generate a set of MM paths (Jorgensen and Erickson, 1994; Zhao and Lin, 2007), where an MM path is an execution sequence of model elements from the start to end of a method scope. Subsequently, we identify infeasible MM paths with respect to MUX pattern and infeasible MM pairs (pairs of MM paths) with respect to NLC pattern.

We study effect of MUX and NLC patterns on two systems of our own namely *Restaurant Automation System* (*RAS*) and *Auditorium Management System* (*AMS*) and three open-source applications: *Crimson, Soot, OpenGTS*. We then determine how much test effort we can save after excluding infeasible paths. Further, we investigate whether location of an infeasibility pattern has any effect on the number of infeasible paths arising out of the pattern.

The rest of the paper is organized as follows. We review the existing techniques for detecting infeasible paths in Section 2. Few basic definitions and concepts related to the area of our research is presented in Section 3. In this section, we also introduce two interaction patterns which cause infeasible paths in UML sequence diagrams. Next, we discuss our approach to build graph model, generate MM paths, and determine their infeasibility. Section 5 presents our experimental results. In Section 6, we compare our technique with the existing techniques. Finally, Section 7 concludes the paper.

2. Related work

In this section, we review the existing work on infeasible path detection techniques. The existing infeasible path detection techniques can be classified into two categories: *static* and *dynamic*. Static techniques are mainly based on the characteristics of program code such as branch correlation, code pattern, satisfiability of a set of predicates along a path etc. On the other hand, dynamic techniques are based on program's real execution where actual values are assigned to input variables and the program's execution flow is monitored. The existing *static* and *dynamic* techniques are briefly reviewed in the next subsections followed by the scope of work.

2.1. Static techniques

Clarke (1976) first reported about path infeasibility for Fortran programs. Main purpose of Clark's work was to generate test data. For this, Clarke (1976) used symbolic path execution where expressions are assigned to program variables instead of values. A set of constraints contained in each path is simplified and then checked for satisfiability using an inequality solver. If these constraints are found to be satisfiable, then the corresponding path is considered as feasible, otherwise it is considered as infeasible. The set of constraints so obtained for a feasible path is used to generate test data.

Based on experimental evidence, Hedley and Hennell (1985) point out that an infeasible path contains at least one infeasible LCSAJ (Linear Code Sequence And Jump) where an LCSAJ is "a linear sequence of executable code commencing either from the start of a program or from a point to which control may jump. It is terminated either by a specific control flow jump or by the end of the program". Hedley and Hennell (1985) performed a study on Fortran Nag Library (The NAG Fortran Library, 2011) consisting of 88 routines, which revealed that 12.5% of the total LCSAJs are infeasible. Their investigation shows that inadequate language constructs, poor programming style are the main causes for infeasible LCSAJs to occur in Fortran programs. However, Hedley et al.'s investigation does not rule out the possibility of existence of infeasible paths even when good programming practices are followed.

Yates and Malevris (1989) propose a path generation approach for branch testing with considering infeasible paths. The motivation of their approach stems from the observation that probability of the path feasibility decays exponentially with the number of traversed predicates.

Based on the work of Yates and Malevris (1989), Malevris (1995) proposes a path generation method to cover all LCSAJs in a program while minimizing the number of infeasible paths generated. The key idea of Malevris's work (Malevris, 1995) is that if an LCSAJ say *m* lies on an infeasible path, then another path with near minimum number of predicates that contains *m* is considered for a candidate feasible path.

Bodík et al. (1997b) refine *def-use* pair analysis (Mall, 2004) using information about infeasible paths. For this, Bodík et al. (1997b) compute branch correlation both intra- and inter-procedurally based on the concept for elimination of redundant conditional branches (Bodík et al., 1997a). However, this branch correlation analysis does not necessarily yield shortest infeasible paths. To identify those, control flow graph is further processed. Subsequently, *def-use* pairs that span the shortest infeasible sub paths are excluded.

Forgács and Bertolino (1997) propose an approach to select feasible test paths to reach a specified program point with a minimum number of *influencing predicates*. To identify those, Forgács and Bertolino (1997) compute *principal slice* using both control and data flow information. As a principal slice is a program slice to a specified program instruction with a minimum number of influencing predicates, the paths derived on *principal slice* are likely to be feasible.

Souter and Pollock (2001) propose a technique to detect *type infeasible call chains* by performing static analysis of call graph for object-oriented programs. A *type infeasible call chain* of a call graph is a call chain that contains at least one type infeasible edge. This type infeasible edge is an edge for which there exists a call subgraph *G*, leading to that edge and satisfying certain properties called *type infeasibility property* (TIP). Such type infeasible call chains can occur in call graphs due to static type *Object* as a formal parameter, polymorphic container (that can contain objects of different types), polymorphic field (that can be instantiated as different types).

Similar to Clark's approach (Clarke, 1976), Zhang and Wang (2001) present an approach to detect infeasible paths by symbolic execution. Zhang et al. first extract a set of constraints on a path after applying the concept of backward substitution starting from the final node to the start node of the path. Subsequently, the set of constraints obtained for a path are checked for satisfiability using a constraint solver named as *BoNuS*. This is an extension of a Boolean satisfiability checker and uses linear programming package. The advantage of using *BoNuS* is that it can accept variables of the types Boolean, integer, real, fixed-size arrays etc.

Ngo and Tan (2007) propose an approach to detect infeasible paths in Java programs with respect to four infeasibility patterns namely *identical/complement-decision, mutually-exclusive-decision, check-then-do*, and *looping-by-flag*. They study effects of these four patterns on several object-oriented systems and find a large number of infeasible paths. To detect them, Ngo and Tan (2007) process control flow graph of class methods using the characteristics of the infeasibility patterns.

Yan and Zhang (2008) propose an approach to generate a set of feasible basis paths. For this, Yan et al. select a basis path on-the-fly with increasing path length starting from one, check whether the path is linearly independent on other paths, and subsequently, verify satisfiability of the predicates on the path using a constraint solver.

If the selected path is found to be infeasible, then the whole process is repeated to find another feasible path of the same length or next higher length until path length exceeds an upper bound (which is required to avoid infinite execution of the program).

2.2. Dynamic techniques

Bueno and Jino (2000) determine the likelihood of a path's infeasibility while generating test data dynamically. For this, Bueno et al. apply genetic algorithm and combine control and data flow information to compute a fitness function, which is subsequently used to guide searching test data for an intended path. If lack of search progress is detected over successive generations, then the intended path is considered to be potentially infeasible.

Ngo and Tan (2008) decide path infeasibility with respect to few empirical properties for correlated conditional statements. Note that two conditional statements are correlated when outcome of later can be implied from the outcome of earlier. Ngo et al. use program traces to check validity of the empirical properties. They (Ngo and Tan, 2008) consider the information of infeasible paths while generating path-oriented test data dynamically as follows. If an infeasible path is found, then the test data generation process for the infeasible path is terminated immediately, thereby avoiding any further effort.

Gong and Yao (2010) propose a dynamic technique using branch correlations, which may not be determined timely and accurately by static analysis (Bodík et al., 1997b). For this reason, Gong and Yao (2010) combine static and dynamic analysis and hence, collect sample data of reasonable size containing the outcomes of branches. Subsequently, Gong et al. use the sample data to calculate the values of statistics required for estimating the conditional probability of branch's outcome (i.e. *true*|*false*). This helps to determine type of branch correlations and path infeasibility.

Observation. If we see research trend for infeasible path detection techniques, we can observe that earlier researchers (Hedley et al., Yates et al.) have considered broader contexts of infeasibility like non-satisfiability of all predicates along a path, branch correlations. Verification of these infeasibility conditions require expensive computation and also have limitations of symbolic evaluation in handling pointers, arrays and function calls etc. For this reason, researchers (Souter et al., Ngo et al.) have shifted focus from broader contexts to specific contexts (Mutually-Exclusive-Decision, Check-then-do, Looping-by-flag, type-infeasible call chain). As per their observations, benefits can be realized only when large number of pattern instances (specific contexts) are found in real life applications and these patterns can be detected easily.

2.3. Scope of work

Infeasible path detection is an undecidable problem and hence, it does not have general solution. To solve this problem, the researchers have considered different infeasibility patterns which are applicable in particular situations and are based on symbolic evaluation (Clarke, 1976; Zhang and Wang, 2001), control and data flow (Bueno and Jino, 2000; Forgács and Bertolino, 1997), branch correlation (Bodík et al., 1997b; Ngo and Tan, 2007; 2008), polymorphic call (Souter and Pollock, 2001) etc. Majority of these infeasibility patterns (Bodík et al., 1997b; Bueno and Jino, 2000; Clarke, 1976; Forgács and Bertolino, 1997; Yan and Zhang, 2008; Zhang and Wang, 2001) are considered for C programs and few patterns (Ngo and Tan, 2007; 2008; Souter and Pollock, 2001) are for Java programs. To cause path infeasibility by the existing pattern(s) (Ngo and Tan, 2007; 2008; Souter and Pollock, 2001), there must have a message path that calls the method containing the infeasibility pattern(s). This fact implies two possibilities for infeasibility of message paths (1) the message path calls a method containing some infeasibility pattern and (2) the message path is itself infeasible. However, the existing techniques have considered first kind of infeasibility of message paths, whereas other type of infeasibility has not been investigated as per our best knowledge.

UML model captures well the message paths of object-oriented systems. But, the question arises which UML diagram(s) we should consider as a source model since various UML diagrams such as sequence, communication diagrams are known for modeling the message paths of object-oriented systems. After through study on these UML diagrams, we decide to use sequence diagram because (1) sequence diagrams resemble code at object interaction level, (2) object interactions and their order are well captured by means of fragments and message sequence in sequence diagrams, and (3) sequence diagrams are usually designed with lower level of abstraction.

UML sequence diagrams can have significant role in integration testing of object-oriented systems. Note that for commercial softwares, it is primarily important to detect critical integration faults, whose leakage can cause serious reliability issue. However, these integration faults are neither detected during structural testing (statement/decision/ condition coverage by tools like Bullseye; Bullseye (2015), *LCOV*; LCOV (2015); *GCOV*) nor they are aimed to uncover during functional testing. This can be addressed by covering method sequence corresponding to message paths in sequence diagrams.

If we target to consider design level sequence diagrams as source model, then its infeasible paths can be detected by analyzing either sequence diagrams or corresponding code. Thus, we need to justify why we should use UML models for detecting infeasible paths. UML-based infeasibility detection has the following advantages over code-based infeasibility detection. *First*, with UML based infeasibility detection, we need to process less amount of information instead of voluminous code. *Second*, we can identify which test scenarios are infeasible after analyzing UML design models at the early stage of software development. This facilitates software developers to refine test effort estimation and hence, make an effective test plan (in a model-driven software development environment) even before the start of coding phase.

Considering aforementioned advantages of model based infeasibility detection, we have planned our objectives as follows: (a) identify infeasible patterns in UML sequence diagrams, (b) analyze the characteristics of these infeasibility patterns, and (c) devise an approach to detect infeasible paths.

3. Proposed approach

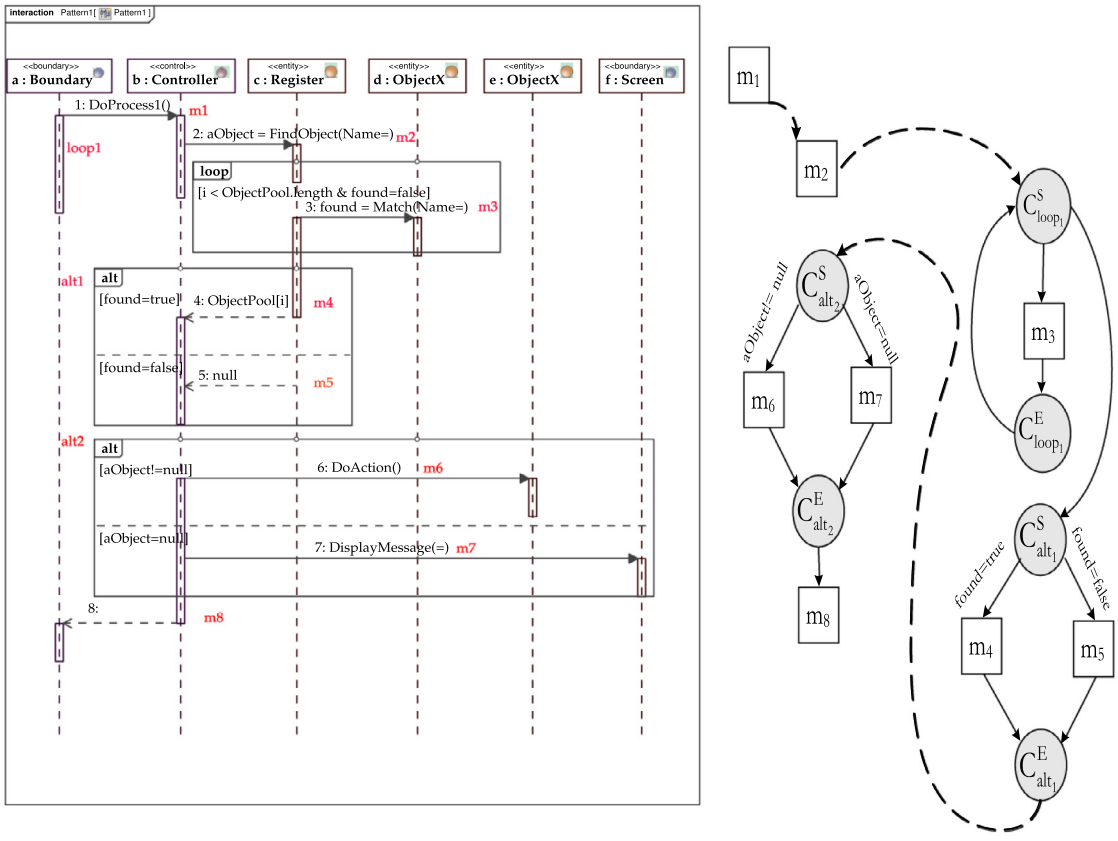
In this section, we first introduce two interaction patterns namely NLC and MUX, which cause infeasible paths in sequence diagrams and subsequently, describe our techniques to detect infeasible paths.

3.1. Two interaction patterns

(a) *Null Reference Check (NLC)*. An NLC interaction pattern consists of modeling both *null* and *non-null* return values for a return variable in one method and checking *nullify test* of the return variable in another method. The return variable must remain unmodified between the point of its return and *nullify test*. If a path has both *null* and *non-null* return values of the return variable under *nullify test* of an NLC pattern, then the path would be infeasible because the value of the return variable cannot be different unless it is modified after return.

Example: Consider an example sequence diagram containing an NLC interaction pattern as shown in Fig. 1.

Here, we can see that *DoProcess*1() calls *FindObject*(), which in turn returns either *null* or *non-null* (ObjectPool[i]) value depending on availability of an object reference in the list. This return value is assigned to the return variable *aObject*, which is subsequently used for *nullify test* by the fragment alt_2 in *DoProcess*1().



(a) Sequence diagram. (b) SIG.

Fig. 1. Sequence diagram with NLC interaction pattern and its SIG.

Let us identify infeasible path(s) caused by NLC interaction pattern present in the sequence diagram (Fig. 1). For this, we consider its SIG (Fig. 1). The SIG has four paths P_1, P_2, P_3, P_4 as shown below. Please note that we have considered only one execution of loop for path generation. In other words, non-execution of loop has been excluded for path generation and therefore, the path not going through $m_3()$ is not a missing path.

$$P_1 = m_1 \rightarrow m_2 \rightarrow C^S_{loop_1} \rightarrow m_3 \rightarrow C^E_{loop_1} \rightarrow C^S_{loop_1} \rightarrow C^S_{alt_1} \rightarrow m_4 \rightarrow C^E_{alt_1} \rightarrow C^S_{alt_2} \rightarrow m_6 \rightarrow C^E_{alt_2} \rightarrow m_8$$

$$P_2 = m_1 \rightarrow m_2 \rightarrow C^S_{loop_1} \rightarrow m_3 \rightarrow C^E_{loop_1} \rightarrow C^S_{loop_1} \rightarrow C^S_{alt_1} \rightarrow m_4 \rightarrow C^E_{alt_1} \rightarrow C^S_{alt_2} \rightarrow m_7 \rightarrow C^E_{alt_2} \rightarrow m_8$$

$$P_3 = m_1 \rightarrow m_2 \rightarrow C^S_{loop_1} \rightarrow m_3 \rightarrow C^E_{loop_1} \rightarrow C^S_{loop_1} \rightarrow C^S_{alt_1} \rightarrow m_5 \rightarrow C^E_{alt_1} \rightarrow C^S_{alt_2} \rightarrow m_6 \rightarrow C^E_{alt_2} \rightarrow m_8$$

$$P_4 = m_1 \rightarrow m_2 \rightarrow C^S_{loop_1} \rightarrow m_3 \rightarrow C^E_{loop_1} \rightarrow C^S_{loop_1} \rightarrow C^S_{alt_1} \rightarrow m_5 \rightarrow C^E_{alt_1} \rightarrow C^S_{alt_2} \rightarrow m_7 \rightarrow C^E_{alt_2} \rightarrow m_8$$

Let us consider a path say P_2 in the SIG. For the path P_2 to be executable, its two predicates ($found = true$) and ($aObject = null$) must be true. The satisfiability of the predicate ($found = true$) implies that *aObject* would be assigned with *non-null* value returned from *FindObject*(), whereas the satisfiability of other predicate (i.e. $aObject = null$) contradicts the former implication. Further, we can observe that *aObject* has not been modified between message 2 and alt_2, which confirms the infeasibility of P_2. Similarly, we can find that the path P_3 is also infeasible.

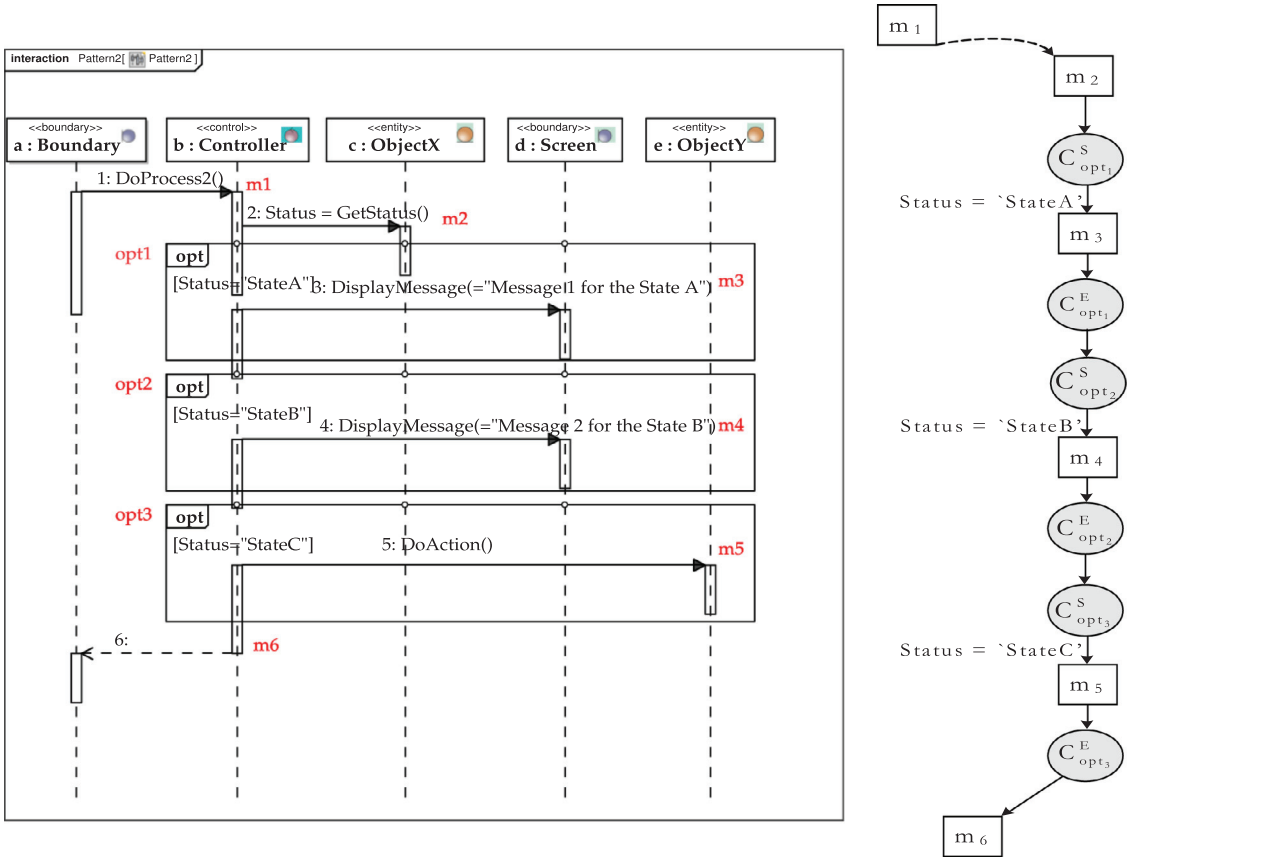
Characteristics: From the above discussions, we can summarize the characteristics of NLC interaction pattern as follows.

1. An NLC interaction pattern consists of modeling both *null* and *non-null* return values for a return variable in one method and checking *nullify test* of the return variable in another method.
2. The return variable under *nullify test* must not be modified between the point of its return and nullify test.
3. The paths on which the return variable is supposed to have both *null* and *non-null* values are infeasible.

Context of occurrence. Let us now investigate the situations where NLC interaction patterns can occur. There may arise some situations where an object is required to be selected on the basis of user input from a list and this responsibility may be delegated to an object *A* by another object *B*. While performing the responsibility, the delegated object *A* returns a valid object reference (i.e. *non-null* value), if it is found, otherwise *null* reference is returned. On receiving the object reference, the object *B* needs to perform *nullify test* of that reference before using it. This causes some paths to have both *null* and *non-null* values of the object reference, which imply their infeasibility. In essence, when object selection is modeled in sequence diagrams by means of delegation from one object to another, NLC interaction patterns are likely to occur.

(b) *Mutually Exclusive (MUX)*. An MUX interaction pattern consists of state-based interaction of an object that has many choices of taking actions (e.g. sending a message, defining/redefining a variable etc.) based on its own current state or other object's state, but the object is permitted to take one action at a time, that is, all actions are mutually exclusive. In this situation, a set of paths that cover at least two mutually exclusive actions are infeasible.

Example: To illustrate MUX interaction pattern, let us consider the sequence diagram shown in Fig. 2. Here, the object *b* gets state information of the object *c* by invoking the message *GetStatus*. Depending on the state of the object *c* (which is stored in the variable



(a) Sequence diagram. (b) SIG.

Fig. 2. Sequence diagram with MUX interaction pattern and its SIG.

Status), the object *b* executes one *opt* fragment as per following: opt_1 for *Status* = 'StateA'; opt_2 for *Status* = 'StateB'; opt_3 for *Status*='StateC'. Since the guard conditions of three *opt* fragments do not satisfy each other and the common influencing variable (i.e. *Status*) has not been modified between opt_1 and opt_3, only one *opt* fragment will be executed at a time, that is, *opt* fragments are mutually exclusive. Note that all three *opt* fragments belong to the method scope of *DoProcess2*.

Let us find infeasible paths in the sequence diagram due to the MUX interaction pattern. For this, we consider SIG for the sequence diagram (Fig. 2) as shown in Fig. 2. The SIG has eight paths $P_1, P_2, \ldots, P_8$ as follows.

$$P_1 = m_1 \to m_2 \to C^S_{opt_1} \to m_3 \to C^E_{opt_1} \to C^S_{opt_2} \to m_4 \to C^E_{opt_2} \to C^S_{opt_3} \to m_5 \to C^E_{opt_3} \to m_6$$

$$P_2 = m_1 \to m_2 \to C^S_{opt_1} \to C^E_{opt_1} \to C^S_{opt_2} \to m_4 \to C^E_{opt_2} \to C^S_{opt_3} \to m_5 \to C^E_{opt_3} \to m_6$$

$$P_3 = m_1 \to m_2 \to C^S_{opt_1} \to m_3 \to C^E_{opt_1} \to C^S_{opt_2} \to C^E_{opt_2} \to C^S_{opt_3} \to m_5 \to C^E_{opt_3} \to m_6$$

$$P_4 = m_1 \to m_2 \to C^S_{opt_1} \to m_3 \to C^E_{opt_1} \to C^S_{opt_2} \to m_4 \to C^E_{opt_2} \to C^S_{opt_3} \to C^E_{opt_3} \to m_6$$

$$P_5 = m_1 \to m_2 \to C^S_{opt_1} \to m_3 \to C^E_{opt_1} \to C^S_{opt_2} \to C^E_{opt_2} \to C^S_{opt_3} \to C^E_{opt_3} \to m_6$$

$$P_6 = m_1 \to m_2 \to C^S_{opt_1} \to C^E_{opt_1} \to C^S_{opt_2} \to m_4 \to C^E_{opt_2} \to C^S_{opt_3} \to C^E_{opt_3} \to m_6$$

$$P_7 = m_1 \to m_2 \to C^S_{opt_1} \to C^E_{opt_1} \to C^S_{opt_2} \to C^E_{opt_2} \to C^S_{opt_3} \to m_5 \to C^E_{opt_3} \to m_6$$

$$P_8 = m_1 \to m_2 \to C^S_{opt_1} \to C^E_{opt_1} \to C^S_{opt_2} \to C^E_{opt_2} \to C^S_{opt_3} \to C^E_{opt_3} \to m_6$$

We observe that the paths P_1, P_2, P_3, P_4 cover at least two mutually exclusive *opt* fragments, that is, they include at least one node between the start and end of each covered fragment. Therefore, P_1, P_2, P_3, P_4 are infeasible paths. If we consider two *if-then-else*, some infeasible paths can be avoided. But, it is possible if we violate *open-closed* design principle for *else part*. It is not desirable to violate the design principle for the sake of avoiding some infeasible paths.

Characteristics: From the above discussion, we can summarize the characteristics of an MUX interaction pattern as follows.

(1) MUX interaction pattern consists of a set of control blocks which are executed in mutually exclusive manner.
(2) To be mutually exclusive, the control blocks must satisfy the following conditions.
  (i) They must belong to the same method scope.
  (ii) They must be independent of each other, that is, one must not contain another.
  (iii) They must be influenced by at least one common variable which is not modified between these control blocks.
  (iv) The predicates associated with the control blocks must not be true altogether.
(3) A path that covers at least two mutually exclusive control blocks of MUX interaction pattern is infeasible.

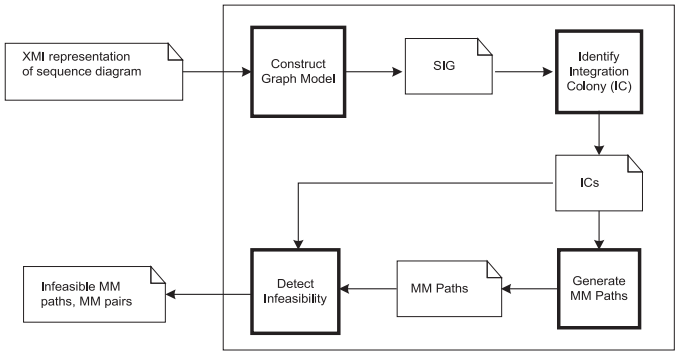


Fig. 3. The framework: *UiD*.

Context of occurrence: Let us investigate the situations where MUX interaction patterns can occur. The basic artifacts of an MUX interaction pattern is a set of control blocks encapsulating a set of actions of an object for its different states. Such control blocks can be modeled in sequence diagrams by means of a set of operands of an *alt* fragment (*if-else if-else*) (IBMRationalUML, 2014) or a sequence of *opt* (*if*) fragments. In case of extending behavior of an object, a new control block (i.e. new operand) would require to be added into *alt* fragment, otherwise *else* part would behave unexpected way. This means that the behavior encapsulated in *alt* fragment does not remain closed upon the addition of new object state, thus violating the *open-closed* principle, that is, "software entities (classes, modules, functions etc.) should be open for extension, but closed for modification" (OCD Principle, 2011). However, such violation would not arise in case of modeling with *opt* fragments, because behavior encapsulated in the existing *opt* fragments would not get affected due to the addition of new object state. In essence, when state-dependent interactions of an object are modeled in sequence diagrams satisfying the *open-closed* principle, MUX interaction patterns are likely to occur.

3.2. Infeasibility detection technique

We propose a framework called *UiD* (*UML-based Infeasible path Detection*) to identify infeasible paths in sequence diagrams. Our proposed framework consists of four steps as shown in Fig. 3.

Taking XMI representation of a given UML 2.*x* sequence diagram as an input, we first construct a graph model (called *SIG*). In the next step, we identify a set of integration colonies (*IC*s) from the graph model. Note that an integration colony is a subgraph of SIG, whose all model elements are in the same method scope. In the third step, all paths of an integration colony are mapped into MM paths. Finally, we detect infeasible MM paths and MM pairs (pairs of MM paths) caused by MUX and NLC patterns.

We now discuss four steps with the help of an example use case *Manage Show* of *AMS* (NID IIT KGP Web, 2010). In *Manage Show* use case, user is asked to enter *show title, show timings, show days*, and *option*. Depending on *option* which can be either *add show* or *update show* or *delete show*, different scenarios can occur. These scenarios are modeled by means of fragments in a sequence diagram (see Fig. 4). For simplicity, we refer to all messages by message numbers and fragments by fragment labels (see Fig. 4).

3.2.1. Constructing graph model

We propose a novel graph model of sequence diagram, called *Sequence Integration Graph* (SIG). The SIG consists of two types of nodes: *control node* and *message node* and two types of edges: *control edge* and *scope edge* (Kundu et al., 2013). A control node represents the start/end of a fragment, whereas a message node represents a message. On the other hand, control edge and scope edge represent control flow and change of method scope, respectively. To construct an SIG for a sequence diagram, we follow the steps given below.

(a) **Determining nodes**. We export sequence diagrams from Magic Draw 16.0 tool by means of XMI representation and then parse the XMI using SAX parser (Kundu et al., 2012) to extract the following: (1) for *message*. sender and receiver objects, parameters, message type; (2) for *fragment*. messages contained in a fragment, fragment type etc. We map model elements (message, fragment start, fragment end) of the sequence diagram into nodes of SIG as follows: (i) a message into a *message node*, (ii) fragment start to a *control node* representing the start of the fragment, (iii) fragment end to a *control node* representing the end of the fragment.

(b) **Determining edges**. We determine hierarchy structure of fragments, that is, outermost fragments (not contained in other fragments) and their inner fragments. This fragment structure is used to determine *precedence relation* (Kundu et al., 2013) between two model elements (x, y), which implies that x executes immediately before y. Finally, *precedence relation* among the model elements are mapped into the edges of their corresponding nodes in SIG.

(c) **Capturing method scope information**. We capture method scope information by means of scope edge. An edge in SIG is considered as a scope edge if its connected model elements are executed in different method scopes (a model element must be executed in some method scope as per Java language).

In our example of *Manage Show*, we obtain the set of nodes as shown in Fig. 5(A), where a circular node represents a control node and a square node represents a message node. Five scope edges (m_1, $C^S_{opt_1}$), (m_6, $C^S_{loop_1}$), ($C^E_{alt_1}$,$C^S_{alt_2}$), (D_{12}, $C^S_{loop_2}$) , ($C^E_{alt_3}$,$C^S_{alt_4}$) are shown as dotted edges.

3.2.2. Identifying integration colonies

Our graph model (SIG) subsumes control flow graph and additionally contains method scope information of model elements. Using method scope information, we identify a subgraph of SIG, whose model elements are in the same method scope. We refer each such subgraph of SIG as an *integration colony*. To identify a set of integration colonies from a given SIG, we process it as follows.

Step 1: Finding a list of scope edge pairs

An integration colony is enclosed by a pair of incoming and outgoing scope edges representing the start and end of a method scope. In fact, integration colonies can be nested, that is, an integration colony may contain another, which may contain other and so on. It is therefore necessary to identify scope edge pairs for the integration colonies starting from the innermost one to the next outer level. Following *depth-first-search* traversal of *SIG*, we identify a set of scope edge pairs such that each pair represents the start and end of same method scope. In our example SIG (in Fig. 5(A)), we find two scope edge pairs as [(m_6, $C^S_{loop_1}$),($C^E_{alt_1}$,$C^S_{alt_2}$)], [(m_{12}, $C^S_{loop_2}$),($C^E_{alt_3}$,$C^S_{alt_4}$)].

Step 2: Finding the subgraph of SIG for a scope edge pair

For each scope edge pair, we identify the start and end nodes of a subgraph of *SIG* as the target node of first scope edge and origin node of second scope edge, respectively. After identifying start and end nodes of the subgraph, we continue to include adjacent nodes and connected edges until only scope edge or no new edge is found. This completes the identification of the subgraph, referred as integration colony. In this way, we identify a set of integration colonies and subsequently, exclude them from SIG resulting a residue graph of SIG, which we refer to as *main integration colony* (MIC).

Two integration colonies and main integration colony for *Manage Show* are shown in Fig. 5(C), (D), (B), respectively.

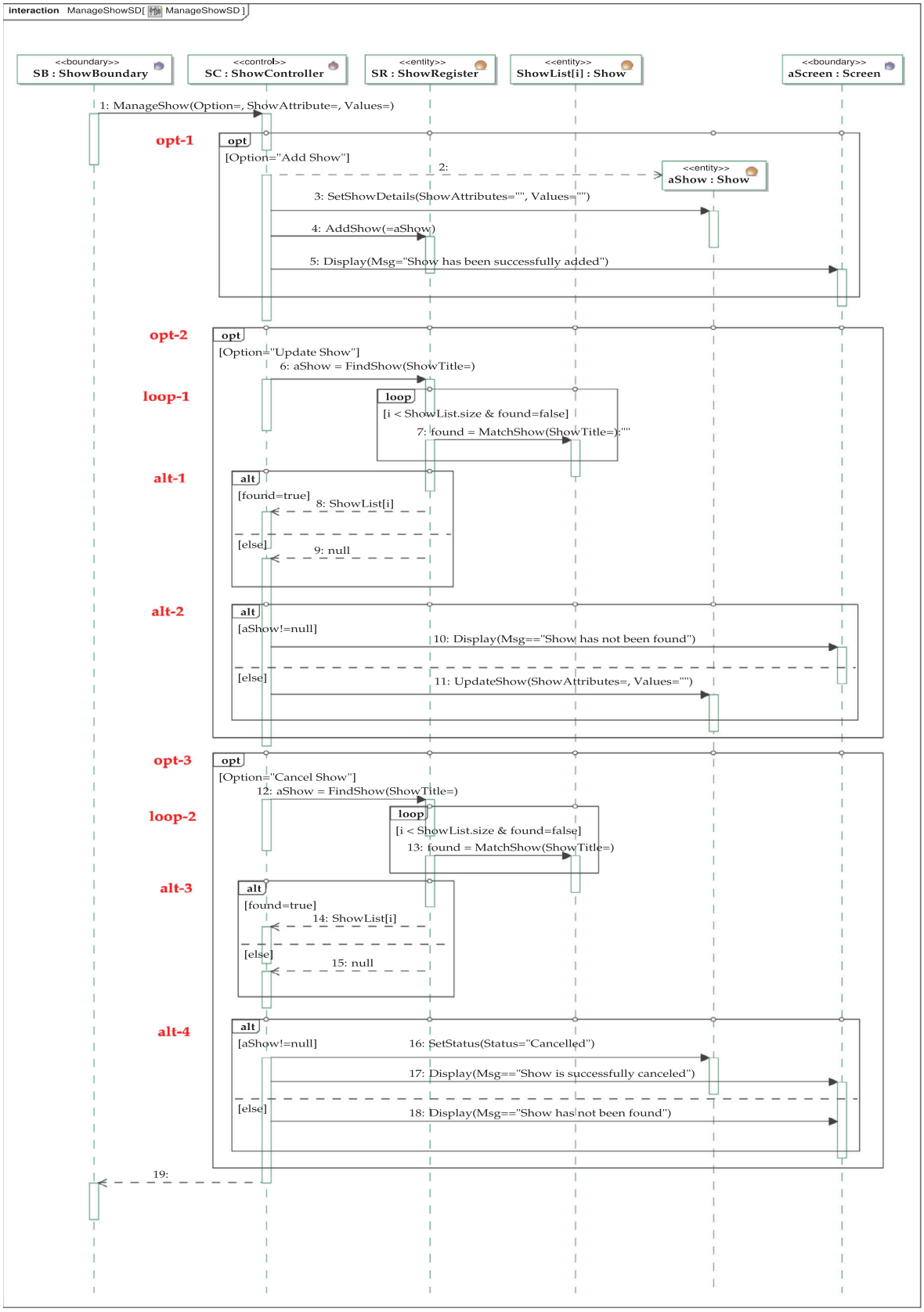


Fig. 4. Sequence diagram of *Manage Show* use case.

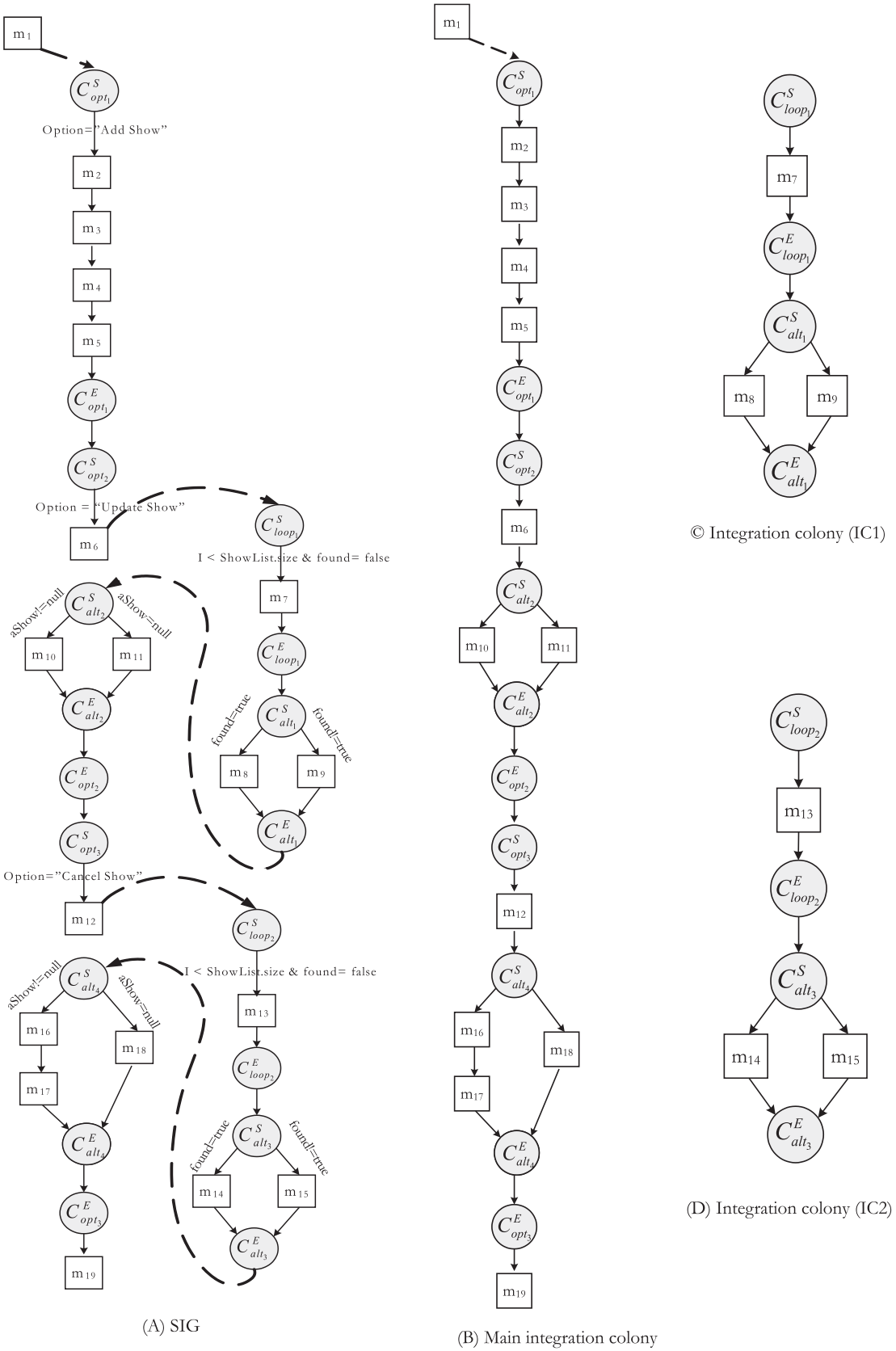


Fig. 5. SIG and three integration colonies for *Manage Show* use case.

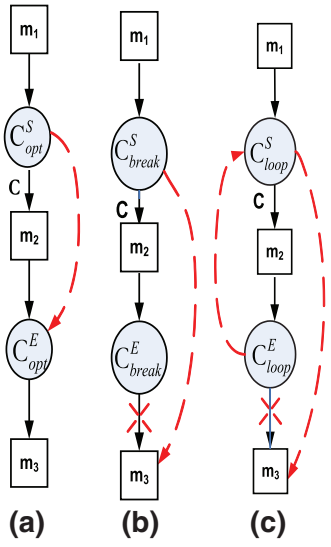


Fig. 6. Three graph models with implicit control flow.

3.2.3. Generating MM paths

An MM path for object-oriented systems refers to a sequence of method execution paths linked by messages (Jorgensen and Erickson, 1994). However, execution of the methods can be conditional, that is, they can be executed only when certain conditions are satisfied. A message invocation of a sequence diagram implies execution of a method with same name as the message and a fragment specifies certain conditions under which methods are executed. Considering this, we generalize the concept of an MM path for UML sequence diagram as an execution sequence of model elements where a model element refers to either a message or start of a fragment or end of a fragment. However, MM paths can be too lengthy because they may contain an arbitrary large sequence of model elements. The main problem of lengthy MM path is that more debugging effort is required to locate faults even if we know that they exist in particular MM path. For this reason, we have considered reduced length of MM path, whose start and end is marked by means of scope edges in our SIG model. Taking this into account, we define an MM path as an execution sequence of model elements from the start to end of a method scope. Because of inherent characteristics of an integration colony that all its model elements are in the same method scope, we choose to use integration colonies for generating MM paths. MM path generation consists of the following two steps.

Step 1: Generating execution sequences of model elements. We enumerate all basic paths from a given integration colony using standard path generation algorithm and then map them into execution sequences of model elements. But, problem arises when integration colonies have few control flows which are not explicit. We refer such control flows as *implicit control flow*s (see dotted lines and crosses in Fig. 6), which may also cause some hidden paths. The reason of existence of implicit flow in our graph model is as follows. For construction of SIG model, we have directly mapped model elements and their precedence relations (which model element occurs before what model element) into nodes and edges, respectively. But, few control flows (e.g. flow from the start of an *opt* fragment to its end) do not map to any precedence relation between model elements in sequence diagram and hence, remain implicit in our SIG model. The advantage of having implicit control flow in our SIG is to avoid loop back edge and to make path generation algorithm simpler.

We now look at few example graph models with implicit control flow. In Fig. 6, we can observe that first graph model (Fig. 6(a)) has implicit control flow (C^S_{opt}, C^E_{opt}) and hence, a hidden path $m_1 \rightarrow C^S_{opt} \rightarrow C^E_{opt} \rightarrow m_3$, that is, the path without executing *opt* fragment. On the other hand, the second graph model (Fig. 6(b)) has implicit information such as stopping control flow at C^E_{break}, transferring control flow from C^S_{break} to m_3, which are the paths with and without executing *break* fragment, respectively. For this, we find two hidden paths: $m_1 \rightarrow C^S_{break} \rightarrow m_2 \rightarrow C^E_{break}$ and $m_1 \rightarrow C^S_{break} \rightarrow m_3$. Similarly, third graph model (Fig. 6(c)) has implicit control flows such as (C^E_{loop}, C^S_{loop}) and (C^S_{loop}, m_3). Taking these control flows into account, we find actual execution path as $m_1 \rightarrow C^S_{loop} \rightarrow m_2 \rightarrow C^E_{loop} \rightarrow C^S_{loop} \rightarrow m_3$ from the third graph model.

Table 1
Three rules.

C^S_f, C^E_f: Start and end nodes of a fragment f

a, b : Two message/fragment nodes

$C^S_f \xrightarrow{c} a$: An edge from C^S_f to a labeled with predicate c

$C^S_f \rightsquigarrow a$: Add an edge from C^S_f to a

$C^E_f \nrightarrow a$: Delete an edge from C^E_f to a

R1 : $(C^S_{loop} \xrightarrow{c} a \;\&\; C^E_{loop} \rightarrow b) \Rightarrow (C^E_{loop} \rightsquigarrow C^S_{loop} \overset{!c}{\rightsquigarrow} b \;\&\; C^E_{loop} \nrightarrow b)$

R2 : $(C^S_{break} \xrightarrow{c} a \;\&\; C^E_{break} \rightarrow b)$
$\Rightarrow (C^E_{break} \nrightarrow b) \;||\; (C^S_{break} \overset{c}{\rightsquigarrow} a \;\&\; C^E_{break} \overset{c}{\rightsquigarrow} b \;\&\; C^S_{break} \overset{!c}{\rightsquigarrow} b)$

R3 : $(C^S_{opt} \xrightarrow{c} a \;\&\; b \rightarrow C^E_{opt}) \;////\; (C^S_{opt} \overset{!c}{\rightsquigarrow} C^E_{opt} \;\&\; C^S_{opt} \nrightarrow a \;\&\; b \;/////\; C^E_{opt})$

To generate hidden paths and transform generated paths into actual execution sequences of model elements, we propose three rules *R*1, *R*2, and *R*3 as mentioned in Table 1. They are applied to the paths containing a fragment of types: *loop, break*, and *opt*, respectively. Applying *R*1, we add two edges: loop back edge (C^E_{loop}, C^S_{loop}), loop exit edge (C^S_{loop}, b) in a path and delete the edge (C^E_{loop}, b) from the path. On the other hand, *R*2 reduces C^E_{break} in a path as *sink node* and introduces another path by excluding *break* fragment. In the same way, *R*3 introduces a new path by excluding the *opt* fragment. Note that *R*2 and *R*3 may need to be applied repeatedly in order to exclude each *break*/*opt* fragment from a path.

Step 2: Mapping execution sequences of model elements into MM paths

Let P be an execution sequence of model elements enabled by a message m and V be the corresponding message node. We map P onto an MM path say P' after concatenating V to P, which is symbolically represented as

$$P' \Rightarrow V \rightarrow P$$

In case of main integration colony, V is the start node and is already included in P. Therefore, this mapping is not necessary for the main integration colony.

By applying the above steps to our example, we obtain eighteen MM paths $M^1_b, M^2_b, M^3_b, \ldots, M^{18}_b$ from the main integration colony (Fig. 5(B)), two MM paths M^1_c, M^2_c from $IC1$ (Fig. 5(C)), two MM paths M^1_d, M^2_d from $IC2$ (Fig. 5(D)) as shown in Table 2.

3.2.4. Identifying infeasible MM paths

In this subsection, we discuss our technique to detect infeasibility of MM paths and MM pairs.

For MUX interaction pattern. We propose an algorithm named as *Detect_Infeasibility_MUX* (see Appendix A) to detect infeasibility of MM paths due to MUX interaction pattern. We process individual integration colonies by applying our algorithm given below.

Step 1: (Identify independent control blocks). We may note that independent control blocks are those which do not contain other control block(s). To identify them, we perform *depth-first-search* traversal of the given integration colony and mark the depth of hierarchy where a control block is found. For this, we use a variable called *level*, which is initialized as *zero*. Each time traversal enters/leaves a control block , the value of *level* is incremented/decremented

Table 2
MM paths of integration colonies of *Manage Show* use case.

Main integration colony	$M_1^b = m_1 \rightarrow C^S_{opt_1} \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow C^E_{opt_3} \rightarrow m_{19}$
	$M_2^b = m_1 \rightarrow C^S_{opt_1} \rightarrow m_2 \rightarrow m_3 \rightarrow m_4 \rightarrow m_5 \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow C^E_{opt_3} \rightarrow m_{19}$
	$M_3^b = m_1 \rightarrow C^S_{opt_1} \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow m_6 \rightarrow C^S_{alt_2} \rightarrow m_{10} \rightarrow C^E_{alt_2} \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow C^E_{opt_3} \rightarrow m_{19}$
	$M_4^b = m_1 \rightarrow C^S_{opt_1} \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow m_6 \rightarrow C^S_{alt_2} \rightarrow m_{11} \rightarrow C^E_{alt_2} \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow C^E_{opt_3} \rightarrow m_{19}$
	$M_5^b = m_1 \rightarrow C^S_{opt_1} \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow m_{12} \rightarrow C^S_{alt_4} \rightarrow m_{16} \rightarrow m_{17} \rightarrow C^E_{alt_4} \rightarrow C^E_{opt_3} \rightarrow m_{19}$
	$M_6^b = m_1 \rightarrow C^S_{opt_1} \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow m_{12} \rightarrow C^S_{alt_4} \rightarrow m_{18} \rightarrow C^E_{alt_4} \rightarrow C^E_{opt_3} \rightarrow m_{19}$
	$M_7^b = m_1 \rightarrow C^S_{opt_1} \rightarrow \cdots \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow \cdots \rightarrow m_{10} \rightarrow \cdots \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow C^E_{opt_3} \rightarrow m_{19}$
	$M_8^b = m_1 \rightarrow C^S_{opt_1} \rightarrow \cdots \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow \cdots \rightarrow m_{11} \rightarrow \cdots \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow C^E_{opt_3} \rightarrow m_{19}$
	$M_9^b = m_1 \rightarrow C^S_{opt_1} \rightarrow \cdots \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow \cdots \rightarrow m_{16} \rightarrow m_{17} \rightarrow \cdots C^E_{opt_3} \rightarrow m_{19}$
	$M_{10}^b = m_1 \rightarrow C^S_{opt_1} \rightarrow \cdots \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow \cdots \rightarrow m_{18} \rightarrow \cdots \rightarrow C^E_{opt_3} \rightarrow m_{19}$
	$M_{11}^b = m_1 \rightarrow C^S_{opt_1} \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow \cdots m_{10} \rightarrow \cdots \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow \cdots \rightarrow m_{16} \rightarrow m_{17} \rightarrow \cdots C^E_{opt_3} \rightarrow m_{19}$
	$M_{12}^b = m_1 \rightarrow C^S_{opt_1} \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow \cdots m_{11} \rightarrow \cdots \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow \cdots \rightarrow m_{16} \rightarrow m_{17} \rightarrow \cdots C^E_{opt_3} \rightarrow m_{19}$
	$M_{13}^b = m_1 \rightarrow C^S_{opt_1} \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow \cdots m_{10} \rightarrow \cdots \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow \cdots \rightarrow m_{18} \rightarrow \cdots C^E_{opt_3} \rightarrow m_{19}$
	$M_{14}^b = m_1 \rightarrow C^S_{opt_1} \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow \cdots m_{11} \rightarrow \cdots \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow \cdots \rightarrow m_{18} \rightarrow \cdots C^E_{opt_3} \rightarrow m_{19}$
	$M_{15}^b = m_1 \rightarrow \cdots \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow \cdots m_{10} \rightarrow \cdots \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow \cdots \rightarrow m_{16} \rightarrow m_{17} \rightarrow \cdots C^E_{opt_3} \rightarrow m_{19}$
	$M_{16}^b = m_1 \rightarrow \cdots \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow \cdots m_{11} \rightarrow \cdots \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow \cdots \rightarrow m_{16} \rightarrow m_{17} \rightarrow \cdots C^E_{opt_3} \rightarrow m_{19}$
	$M_{17}^b = m_1 \rightarrow \cdots \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow \cdots m_{10} \rightarrow \cdots \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow \cdots \rightarrow m_{18} \rightarrow \cdots C^E_{opt_3} \rightarrow m_{19}$
	$M_{18}^b = m_1 \rightarrow \cdots \rightarrow C^E_{opt_1} \rightarrow C^S_{opt_2} \rightarrow \cdots m_{11} \rightarrow \cdots \rightarrow C^E_{opt_2} \rightarrow C^S_{opt_3} \rightarrow \cdots \rightarrow m_{18} \rightarrow \cdots C^E_{opt_3} \rightarrow m_{19}$
IC1	$M_1^c = m_6 \rightarrow C^S_{loop_1} \rightarrow m_7 \rightarrow C^E_{loop_1} \rightarrow C^S_{loop_1} \rightarrow C^S_{alt_1} \rightarrow m_8 \rightarrow C^E_{alt_1}$
	$M_2^c = m_6 \rightarrow C^S_{loop_1} \rightarrow m_7 \rightarrow C^E_{loop_1} \rightarrow C^S_{loop_1} \rightarrow C^S_{alt_1} \rightarrow m_9 \rightarrow C^E_{alt_1}$
IC2	$M_1^d = m_{12} \rightarrow C^S_{loop_2} \rightarrow m_{13} \rightarrow C^E_{loop_2} \rightarrow C^S_{loop_2} \rightarrow C^S_{alt_3} \rightarrow m_{14} \rightarrow C^E_{alt_3}$
	$M_2^d = m_{12} \rightarrow C^S_{loop_2} \rightarrow m_{13} \rightarrow C^E_{loop_2} \rightarrow C^S_{loop_2} \rightarrow C^S_{alt_3} \rightarrow m_{15} \rightarrow C^E_{alt_3}$

Table 3
Independent control blocks of graph model for *Manage Show*.

Integration colony	Level	Independent control blocks	S_v	Correlated
MIC	One	$(C^S_{opt_1}, C^E_{opt_1})$, $(C^S_{opt_2}, C^E_{opt_2})$, $(C^S_{opt_3}, C^E_{opt_3})$	{*Option*}	Yes
	Two	$(C^S_{alt_2}, C^E_{alt_2})$, $(C^S_{alt_4}, C^E_{alt_4})$	{*aShow*}	Yes
*IC*1	One	$(C^S_{loop_1}, C^E_{loop_1})$, $(C^S_{alt_1}, C^E_{alt_1})$	{*found*}	Yes
*IC*2	One	$(C^S_{loop_2}, C^E_{loop_2})$, $(C^S_{alt_3}, C^E_{alt_3})$	{*found*}	Yes

by one. All control blocks which are in the same depth of hierarchy (represented by the value of *level*) are the independent control blocks.

Applying *step1* to all integration colonies for *Manage Show*, we find the four sets of independent control blocks shown in Table 3. From this table, we can see that the main integration colony has two sets of independent control blocks, whereas *IC*1 and *IC*2 have one set each.

Step 2 (Determine correlated control blocks). For each set of independent control blocks, we check whether there exists a non-empty set of common predicate variables. If such non-empty set is found, the associated control blocks are considered as correlated.

In our example with four sets of independent control blocks listed in Table 3, we find that $(C^S_{opt_1}, C^E_{opt_1})$, $(C^S_{opt_2}, C^E_{opt_2})$, $(C^S_{opt_3}, C^E_{opt_3})$ are correlated. This is because, they have a common predicate variable *Option* as evident from their predicates shown in Table 4. Similarly, other three sets have one common predicate variable each and hence, they are also correlated (see Table 4).

Step 3 (Identify mutually exclusive control blocks). For this, we check whether a set of correlated control blocks of an integration colony satisfies the following two conditions.

(i) The integration colony must not have a node defining/redefining a common predicate variable of the correlated control blocks.
(ii) All predicates associated with the correlated control blocks must not be true altogether for a domain value of the common predicate variable(s).

To verify the second condition, we consider whether the predicates have conflicting operators with the same right operands or vice-versa. If these conditions are satisfied, then the correlated control blocks would be mutually exclusive, that is, only one of them would be executed. An MM path is considered as infeasible if it covers at least two mutually exclusive control blocks.

In our example, we verify four sets of correlated control blocks for mutual exclusiveness. In main integration colony of example SIG, we find $(C^S_{opt_1}, C^E_{opt_1})$, $(C^S_{opt_2}, C^E_{opt_2})$, $(C^S_{opt_3}, C^E_{opt_3})$ as mutually exclusive because their associated predicates *Option='Add Show'*, *Option='Update Show'*, *Option='Delete Show'* contradict each other and there does not exist a single node between $C^S_{opt_1}$ and $C^E_{opt_3}$, which modifies the variable *Option*. On the other hand, such condition has not been found true for other three sets of correlated control blocks (see Table 4). After checking the coverage of MM paths in the main integration colony, we find that 12 MM paths namely M_7^b, M_8^b, M_9^b, M_{10}^b, M_{11}^b, M_{12}^b, M_{13}^b, M_{14}^b, M_{15}^b, M_{16}^b, M_{17}^b, M_{18}^b cover at least two out of three mutually exclusive control blocks $(C^S_{opt_1}, C^E_{opt_1})$, $(C^S_{opt_2}, C^E_{opt_2})$, $(C^S_{opt_3}, C^E_{opt_3})$ and hence, they are infeasible.

For NLC interaction pattern. We propose an algorithm named as *Detect_Infeasibility_NLC* (see Appendix A) to detect infeasibility of pair of MM paths arising out of the NLC pattern. *Detect_Infeasibility_NLC* has two major steps as discussed below.

Step 1 (Identify correlated integration colony pairs). We consider a pair of integration as correlated if one is nested under other by means of scope edges.

In our example (Fig. 5), we can observe that the integration colony IC_1 is nested under the Main Integration Colony (MIC) by means of

Table 4
Correlated control blocks of three integration colonies.

Set of correlated control blocks	Set of predicates	Satisfiability of predicates	Common variable modification	Mutually exclusive
$(C^S_{opt_1}, C^E_{opt_1})$	*Option='Add Show'*			
$(C^S_{opt_2}, C^E_{opt_2})$	*Option='Update Show'*	No	No	Yes
$(C^S_{opt_3}, C^E_{opt_3})$	*Option='Delete Show'*			
$(C^S_{alt_2}, C^E_{alt_2})$	*aShow* ≠ *Null*			
$(C^S_{alt_4}, C^E_{alt_4})$	*aShow* ≠ *Null*	Yes	Yes	No
$(C^S_{loop_1}, C^E_{loop_1})$	*i* < *ShowList.size & found=false*			
$(C^S_{alt_1}, C^E_{alt_1})$	*found=true, found!=true*	Yes	Yes	No
$(C^S_{loop_2}, C^E_{loop_2})$	*i* < *ShowList.size & found=false*			
$(C^S_{alt_3}, C^E_{alt_3})$	*found=true, found!=true*	Yes	Yes	No

scope edges (m_6, $C^S_{loop_1}$), ($C^E_{alt_1}$, $C^S_{alt_2}$) and hence they are correlated. Similarly, we find another pair (*MIC*, IC_2).

Step 2 (Check infeasibility for concatenation of MM paths). For each pair of correlated integration colonies, we identify the set of potentially infeasible MM pairs where first MM path contains a *nullify test* predicate and second MM path has return value for variable referred in *nullify test*. For each such MM pair, we now check whether they satisfy the following conditions.

(i) First MM path must not define/redefine the said predicate variable between the point of a method call and *nullify test* predicate.
(ii) The *nullify test* predicate must not be true for the return value from the method call.

If these two conditions are found to be true, then a contradictory situation arises where the expected value of the said predicate variable to satisfy the predicate does not match with return value from prior method call, which is not possible unless the predicate variable is modified. With this situation, the MM pair (i.e. concatenation of one with other) is infeasible.

Examining the coverage information of MM paths of integration colony pair (*MIC*, IC_1), we find that M^b_3, M^b_4, M^b_7, M^b_8, M^b_{11}, M^b_{12}, M^b_{13}, M^b_{14}, M^b_{15}, M^b_{16}, M^b_{17}, M^b_{18} of *MIC* have *nullify test* predicate. On the other hand, two MM paths M^c_1 and M^c_2 of IC_1 have the return values *ShowList*[*i*] and *null* for variable *aShow* referred in *nullify test*, respectively. Therefore, all MM pairs in the product set $\{M^b_3, M^b_4, M^b_7, M^b_8, M^b_{11}, M^b_{12}, M^b_{13}, M^b_{14}, M^b_{15}, M^b_{16}, M^b_{17}, M^b_{18}\} \times \{M^c_1, M^c_2\}$ are subject to infeasibility verification. Consider an MM pair (M^b_3, M^c_2) where M^b_3 has the *nullify test* predicate as *aShow* ≠ *null* and M^c_2 has *null* return value that cannot satisfy the predicate. The fact that *aShow* has not been modified between the point of prior method call and *nullify test* predicate confirms the infeasibility of the MM pair (M^b_3, M^c_2). Similar way, we find other infeasible MM pairs $\{M^b_4, M^b_8, M^b_{12}, M^b_{14}, M^b_{16}, M^b_{18}\} \times \{M^c_1\}$ and $\{M^b_7, M^b_{11}, M^b_{13}, M^b_{15}, M^b_{17}\} \times \{M^c_2\}$.

4. Experimental results and analysis

In this section, we discuss our experimental results followed by analysis.

4.1. Objectives

The objectives of our experiments are as follows.

(i) Investigate the extent to which identified MUX and NLC interaction patterns make MM paths, MM pairs, and scenarios infeasible.
(ii) Investigate how much test effort can be saved after excluding infeasible paths.
(iii) Investigate whether the locations of interaction patterns influences the number of infeasible paths.
(iv) Compare the computation overhead of detecting infeasible paths using SIG and control flow graph.

4.2. Subject programs

Since our approach targets to detect infeasible message paths in design level UML sequence diagrams, availability of UML design specifications is necessary to validate our approach. In product based industry, design level sequence diagrams are prepared with sufficient details and following which code is developed after keeping them consistent with sequence diagrams. However, such industrial designs are not accessible in public domain due to confidentiality reason and that is why, they could not be used in our research. For this reason, we had to use two systems of our own: *RAS* and *AMS* (NID IIT KGP Web, 2010). Note that *RAS* consists of 21 classes and its code size is 4.6 KLOC, whereas *AMS* has 18 classes and its code size is 3.8 KLOC. Following the standard design principles (Booch et al., 2005), we have prepared their design specifications which include one use case diagram per system, a use case description per use case, one class diagram per system, one activity diagram and one sequence diagram per use case, one statechart diagram for each important domain object. Since our approach uses only sequence diagrams, we focus on them while describing RAS and AMS in the following.

Restaurant Automation System (*RAS*). This system automates various functionalities of a restaurant such as make order, process order, deliver order, generate bill, pay bill, generate sale statistics etc. The design specification of *RAS* includes seven sequence diagrams for the following use cases: *Make Order, Process Order, Deliver Order, Generate Bill, Pay Bill, Manage Item, Generate Statistics*. These sequence diagrams contain three types of collaborating objects: (1) *controller objects*: ProcessOrderController, ManageItemController, MakeOrderController, GenerateStatisticsController, (2) *boundary objects*: SelectOrderForm, PaymentForm, ManageItemForm, GenerateStatisticsForm, OrderForm, MenuForm, PaymentSlip, (3) *entity objects*: ItemRegister, BillRegister, OrderRegister, Item, Bill, Order, Payment, ChequePayment, PaymentRegister. The number of messages, fragments, and collaborative objects contained in seven sequence diagrams of *RAS* are shown in Table 5.

Auditorium Management System (*AMS*). This system automates various functionalities of an auditorium such as managing movie-shows, buying tickets, cancelling tickets, computing sale commissions, generating sale statistics etc. The design specification of *AMS* includes six sequence diagrams for the following use cases: *Book Ticket, Cancel Ticket, Compute Sale Commission, Pay Commission, Manage Show, Generate Show Statistics*. These sequence

Table 5
Number of MUX and NLC patterns in the sequence diagrams for use cases of *RAS* and *AMS*.

System	Use case	Number of messages	Number of fragments	Number of objects	Number of integration colonies (IC)	Number of interaction patterns	
						MUX	NLC
RAS	Make Order	10	0	5	1	0	0
	Deliver Order	21	13	9	3	1	2
	Manage Item	17	9	6	3	1	2
	Process Order	34	18	9	4	1	3
	Generate Bill	19	6	9	2	0	1
	Generate Statistics	16	6	9	1	1	0
	Pay Bill	23	5	12	3	0	1
AMS	Book Ticket	20	4	11	2	0	1
	Cancel Ticket	22	4	11	2	0	1
	Compute Sale Commission	18	5	9	3	0	1
	Pay Commission	12	5	6	2	1	1
	Manage Show	19	9	6	3	1	2
	Generate Show Statistics	16	8	7	4	1	0

Table 6
Infeasibility of MM path, MM pairs, scenario paths.

System	Use case	MM path			MM pair			Scenario		
		Total (*A*)	Infeasible (*B*)	Percent of infeasible (*C*)	Total (*D*)	Infeasible (*E*)	Percent of infeasible (*F*)	Total (*G*)	Infeasible (*H*)	Percent of infeasible (*I*)
RAS	Deliver Order	17	4	23.52	30	19	63.33	30	21	70
	Manage Item	22	12	54.54	48	44	91.66	50	44	88
	Process Order	105	84	80.00	534	507	94.94	678	663	97.78
	Generate Bill	6	0	0	8	4	50	8	4	50
	Generate Statistics	8	4	50	0	0	0	8	4	50
	Pay Bill	6	0	0	8	3	37.5	6	3	50
AMS	Book Ticket	5	0	0	6	3	50	6	3	50
	Cancel Ticket	5	0	0	5	2	40	4	2	50
	Compute Sale Commission	6	0	0	6	2	33.33	6	3	50
	Pay Commission	7	1	14.28	10	6	60	10	6	60
	Manage Show	22	12	54.54	48	44	91.66	50	44	88
	Generate Show Statistics	13	4	30.76	20	15	75	18	12	66.67

diagrams contain three types of objects as follows: (1) *controller objects*: TicketController, CommissionController, ShowController, StatisticsController, (2) *boundary objects*: TicketBoundary, CommissionBoundary, ShowBoundary, StatisticsBoundary, Screen, (3) *entity objects*: IndexSeatRegister, SeatRegister, TicketRegister, Ticket, FundRegister, Fund, Refund, CommissionRegister, Commission, ShowRegister, Show. The characteristics of six sequence diagrams of *AMS* are given in Table 5.

To mitigate the subjective bias due to the usage of two systems of our own, we choose three open source applications namely *Crimson* (11.4 KLOC), *Soot* (274.5 KLOC), *OpenGTS* (139.4 KLOC). The purpose of using reverse engineered sequence diagrams for *Crimson, SOOT, OpenGTS* is to find the evidences of our infeasibility patterns in real-life software applications. The average numbers of [messages, fragments, objects] of three open source applications: *Crimson, Soot, OpenGTS* after reverse engineering are [12,6,4], [21,11,8], [16,9,6], respectively. We briefly discuss their functionalities in the following.

Crimson. Crimson is a Java XML parser which supports XML 1.0 (http://xml.apache.org/crimson/, 2013).

SOOT. Soot is an optimization framework for Java bytecode (Soot, 2013). Soot consists of three intermediate representations namely *Baf* (a streamlined representation of bytecode), *Jimple* (a typed 3-address intermediate representation), *Grimp* (an aggregated version of Jimple suitable for decompilation and code inspection) (Vallée-Rai et al., 1999). It supports transformations between these intermediate representations as well as optimization techniques on these representations.

OpenGTS. OpenGTS is an open source project designed to provide web-based GPS tracking services for vehicles (OpenGTS, 2013). OpenGTS supports the following features: (1) integration with mapping service providers such as Google Maps, Microsoft Virtual Earth; (2) customizable reports to show historical data for a specific vehicle; (3) customizable geofenced areas (geozones) to provide arrival/departure events on reports etc.

4.3. Effect of MUX and NLC interaction patterns on the infeasibility of MM paths, MM pairs, and scenarios

First, we discuss our experiments with RAS and AMS. Subsequently, we investigate the effects of MUX and NLC patterns in three open-source applications.

RAS and AMS. We count the number of MUX and NLC patterns found in different sequence diagrams of *RAS* and *AMS* as shown in Table 5. From this table, we can observe that majority sequence diagrams (except *Make Order* and *Generate Statistics* of *RAS, Generate Show Statistics* of *AMS*) contain one or more NLC pattern(s), whereas seven sequence diagrams have one MUX pattern each (see Table 5).

Following our approach, we build SIG models for all design level sequence diagrams of RAS and AMS, identify their integration colonies, apply our algorithm *Detect_Infeasibility_MUX* on the individual integration colonies to detect infeasible MM paths. We then measure how many of the total MM paths of an SIG are infeasible as shown in Table 6. From Table 6, we can observe that 14%–80% MM paths of different sequence diagrams are infeasible due to MUX pattern(s).

Now, we examine whether there exists some correlation between the total number of MM paths of the integration colony containing MUX pattern(s) and % infeasible MM paths. Since their distributions are not known, we apply Spearman's non-parametric

Table 7
Correlation of infeasibility of MM paths with the total number of MM paths.

Use case	Total number of MM paths (A)	Percent of infeasible MM paths (C)	A rank (AR)	C rank (CR)	d = CR − AR	d^2
Deliver Order	17	23.52	4	6	2	4
Manage Item	22	54.54	2.5	2.5	0	0
Process Order	105	80.00	1	1	0	0
Generate Statistics	8	50	6	4	−2	4
Pay Commission	7	14.28	7	7	0	0
Manage Show	22	54.54	2.5	2.5	0	0
Generate Show Statistics	13	30.76	5	5	0	0

Table 8
Correlation of infeasible MM pairs with its dependent factors.

Use case	Percent of infeasible MM path (C)	Total MM pairs (D)	Percent of infeasible MM pairs (F)	C Rank (CR)	D Rank (DR)	F Rank (FR)	d_1 = FR − CR	d_1^2	d_2 = FR − DR	d_2^2
Deliver Order	23.52	30	63.33	5	4	5	0	0	1	1
Manage Item	54.54	48	91.66	2.5	2.5	2.5	0	0	0	0
Process Order	80.00	534	94.94	1	1	1	0	0	0	0
Generate Bill	0	8	50.00	9	7.5	7.5	−1.5	2.25	0	0
Pay Bill	0	8	37.50	9	7.5	10	1	1	2.5	6.25
Book Ticket	0	6	50.00	9	9.5	7.5	−1.5	2.25	−2	4
Cancel Ticket	0	5	40.00	9	11	9	0	0	−2	4
Compute Sale Commission	0	6	33.33	9	9.5	11	2	4	1.5	2.25
Pay Commission	14.28	10	60.00	6	6	6	0	0	0	0
Manage Show	54.54	48	91.66	2.5	2.5	2.5	0	0	0	0
Generate Show Statistics	30.76	20	75	4	5	4	0	0	−1	1

method (Spearman, 2011) to compute the individual ranks of both the total number of MM paths and % infeasible MM paths (see Table 7), which are used to determine the correlation coefficient (r) as per the following equation.

$$r = 1 - \frac{\left(6 \times \sum_{i=1}^{n} d^2\right)}{n \times (n^2 - 1)} = 0.86 \qquad (n = 7) \tag{1}$$

Here, n refers to the number of items in sample data and d_i represents the difference between ranks of individual data of two data sets.. The high value of r indicates strong positive correlation between them at 0.025% significance level (Zar, 1972). From this, we can imply that the occurrence of MUX interaction pattern(s) in an integration colony with higher total of MM paths causes higher percentage of infeasible MM paths.

Next, we apply our algorithm *Detect_Infeasibility_NLC* on pairs of integration colonies of individual SIGs to detect infeasible MM pairs, that is, concatenation of pairs of MM paths are infeasible or not. We also take other infeasible MM pairs into account, whose one MM path is infeasible due to MUX pattern(s). The total MM pairs and their infeasibility percentage of all SIGs are reported in Table 6. From Table 6, we can find that different graph models have 33%–94% infeasible MM pairs. To identify potential dependent factors of % infeasible MM pairs, we compute Spearman's correlations (Spearman, 2011) (1) between % infeasible MM paths (C) and % infeasible MM pairs (F), (2) between total MM pairs (D) and % infeasible MM pairs (F). For them, we compute their individual ranks and obtain the correlation coefficient r as 0.96 at 0.005% significance level for the first pair (C and F) and 0.92 at 0.005% significance level for the second pair (D and F) (Zar, 1972) (see Table 8).

This indicates strong positive correlation for both pairs. From this, we can conclude that (1) higher percentage of infeasible MM paths in graph model induces higher percentage of infeasible MM pairs, (2) the occurrence of NLC/MUX interaction pattern in a graph model with higher number of MM pairs causes higher percentage of infeasible MM pairs.

Next, we identify infeasible scenarios after checking whether they contain some infeasible MM path or MM pair. From Table 6, we can observe that different graph models in RAS and AMS have 50%–97% infeasible scenarios. To identify potential dependent factors of the percentage of infeasible scenarios, we determine Spearman's correlations (Spearman, 2011) (1) between % infeasible MM paths (C) and % infeasible scenarios (I), (2) between % infeasible MM pairs (F) and % infeasible scenarios (I). The values of correlation coefficient for these two pairs are 0.88 at 0.001% significance level and 0.94 at 0.001% significance level, respectively (Zar, 1972). This implies that a graph model with larger number of infeasible MM paths or MM pairs results the higher percentage of infeasible scenarios (Table 9).

Open source applications. For experiments, we have obtained reverse engineered sequence diagrams for the class methods of three open source applications (*Crimson, Soot, OpenGTS*) and subsequently applied our tool on them to detect infeasible paths. As there are a huge number of classes in those applications (further, each class also has a large number of methods), we had to restrict ourselves in selecting a subset of all class methods for reverse engineering. For selection, we had reviewed code manually whether (i) each class method contains an MUX pattern instance and (ii) a pair of class methods contain an NLC pattern instance. The total efforts required to apply our technique on three open source applications were 12 Man-Days for *SOOT*, 7 Man-Days for *OpenGTS*, and 3 Man-Days for *Crimson*. Please note that these efforts were due to the following tasks: (a) code review, (b) selection of class methods, (c) reverse engineering of the selected class methods, (d) tool execution on reverse engineered sequence diagrams. Among these tasks, first three tasks have been

Table 9
Correlation of infeasible scenarios with its dependent factors.

Use case	Percent of infeasible MM path (C)	Percent of infeasible MM pairs (F)	Percent of infeasible scenarios (I)	C Rank (CR)	F Rank (FR)	I Rank (IR)	$d_3 = \text{IR} - \text{CR}$	d_3^2	$d_4 = \text{IR} - \text{FR}$	d_4^2
Deliver Order	23.52	63.33	70	6	5	4	−2	4	−1	1
Manage Item	54.54	91.66	88	2.5	2.5	2.5	0	0	0	0
Process Order	80.00	94.94	97.78	1	1	1	0	0	0	0
Generate Bill	0	50.00	50	10	7.5	9.5	−0.5	0.25	2	4
Generate Statistics	50.00	0	50	4	12	9.5	5.5	30.25	−2.5	6.25
Pay Bill	0	37.50	50	10	10	9.5	−0.5	0.25	−0.5	0.25
Book Ticket	0	50.00	50	10	7.5	9.5	−0.5	0.25	2	4
Cancel Ticket	0	40.00	50	10	9	9.5	−0.5	0.25	0.5	0.25
Compute Sale Commission	0	33.33	50	10	11	9.5	−0.5	0.25	−1.5	2.25
Pay Commission	14.28	60.00	60	7	6	6	−1	1	0	0
Manage Show	54.54	91.66	88	2.5	2.5	2.5	0	0	0	0
Generate Show Statistics	30.76	75	66.67	5	4	5	0	0	1	1

Table 10
Infeasible paths of open source applications for MUX patterns.

System	Source package	Number of instances	Number of MM paths	Number of infeasible MM paths
Crimson	org.apache.crimson.parser	4	64	29
	org.apache.crimson.tree	4	172	43
	Total		236	72 (30.50%)
Soot	soot.dava.toolkits.base.finders	1	810	324
	soot.javaToJimple	10	291	222
	soot.dava	2	736	296
	soot.jimple.toolkits.thread.mhp	1	24	10
	soot.jimple.spark.builder	1	40	18
	soot.jimple.toolkits.annotation.arraycheck	1	440	160
	soot.jimple.toolkits.annotation.nullcheck	1	432	144
	Total		2773	1174 (42.33%)
OpenGTS	org.opengts.db	2	1928	482
	org.opengts.war.tools	1	60	20
	org.opengts.dbtools	1	56	38
	Total		2044	540 (26.41%)

performed manually since the required tools are neither available nor developed by us and the main task namely the detection of infeasible paths in SIG was carried out by the tool of our own rather than manually.

Since SIG of a reverse engineered sequence diagram contains the artifacts belonged to single method and is equivalent to an integration colony, we apply the algorithm for MUX pattern on individual SIGs of reverse engineered sequence diagrams and find how many of the total MM paths of an SIG are infeasible. Table 10 shows the number of MUX patterns as well as the number of MM paths in different packages. From Table 10, we can observe that the MUX patterns cause 30% infeasible MM paths (72 out of 236) in *Crimson*, 42% infeasible MM paths (1174 out of 2773) in *SOOT*, 26% infeasible MM paths (540 out of 2044) in *OpenGTS*.

For NLC pattern, we identify the pairs of SIGs, where one SIG would contain the *nullify* test of a return variable and another SIG would contain both the *null* and *non-null* return values for the return variable. Applying our algorithm for NLC pattern to such pairs of SIG models, we measure how many of their total MM pairs are infeasible. We have identified 47 instances of NLC patterns in *Crimson*, 166 in *OpenGTS*, 132 in *SOOT* as shown in package wise of Table 11. From this table, we can observe that NLC patterns cause 55% infeasible MM pairs (630 out of 1145) in *Crimson*, 49% infeasible MM pairs (3656 out of 7360) in *SOOT*, 52% infeasible MM pairs (1047 out of 1982) in *OpenGTS*.

4.4. Influence of infeasible paths on test effort estimation

We now examine how much effort we can save after excluding infeasible paths. For this, we use Almeida et al.'s approach for test effort estimation based on use case points (de Almeida et al., 2009). Considering all actors as *simple types*, we find the total *Unadjusted Actor Weight* (*UAW*) of *RAS* and *AMS* as *seven* and *six*, respectively (see Table 12). Subsequently, we identify the number of normal and exceptional scenarios of their use cases to determine *Unadjusted Use Case Weights* (UUCW) as shown in Table 13.

After summing up *UAW* and *UUCW*, we obtain *Unadjusted Use Case Points* (UUCP) value for *RAS* and *AMS* as [769, 77] (for all paths) and [41, 25] (for all feasible paths). In Table 13, $UUCP_a$ and $UUCP_f$ denote the *Unadjusted Use Case Points* for all paths and all feasible paths, respectively. Please note that the weight for exceptional scenario is assumed to be half of the weight considered for normal scenario. This is because, the cumulative effort required to do design, implementation and testing is substantially high for normal scenario compared to exceptional scenario. Finally, we obtain *Adjusted Use Case Points* (AUCP) for *RAS* and *AMS* as $[769 \times c, 77 \times c]$ (for all paths) and $[41 \times c, 25 \times c]$ (for feasible paths) where $c = 0.65 + 0.01 \times TEF$ and *TEF* represents *Technical Complexity Factor*.

If the testing effort per use case point is *K Person-Hour*, then the total test effort (E) for *AUCP* use case points would be $E = AUCP \times K$. Let us suppose that E_a and E_f be the corresponding test effort for all

Table 11
Infeasible paths of open source applications for NLC patterns.

System	Source package	Destination package	Number of instances	Number of MM pairs	Number of infeasible MM pairs
Crimson	org.apache.crimson.parser	org.apache.crimson.parser	20	894	496
	org.apache.crimson.tree	org.apache.crimson.tree	19	189	103
	org.apache.crimson.util	org.apache.crimson.parser	1	8	4
		org.apache.crimson.tree	3	22	11
	org.xml.sax.helpers	org.apache.crimson.parser	2	10	5
		org.xml.sax.helpers	2	22	11
		Total		1145	630 (55.02%)
Soot	soot	soot	19	245	135
		soot.javaToJimple	1	9	5
	soot.coffi	soot.coffi	3	69	37
	soot.dava.toolkits.base.AST .structuredAnalysis	soot.dava.toolkits.base.AST .structuredAnalysis	11	1426	656
		soot.dava.toolkits.base.AST .transformations	13	249	116
	soot.dava.toolkits.base.AST .transformations	soot.dava.toolkits.base.AST .transformations	67	5117	2585
	soot.jimple.toolkits.pointer	soot.jimple.toolkits.pointer	8	77	35
	soot.jimple.toolkits.thread .mhp	soot.jimple.toolkits.thread .mhp	2	44	27
	soot.jimple.toolkits.thread .synchronization	soot.jimple.toolkits.thread .synchronization	5	46	22
	soot.jimple.toolkits.typing	soot.jimple.toolkits.typing	2	54	26
		soot.jimple.toolkits.typing .fast	1	24	12
		Total		7360	3656 (49.67%)
OpenGTS	org.opengts.db	org.opengts.tools	4	44	19
		org.opengts.db.tables	4	60	26
		org.opengts.db	28	200	99
		org.opengts.geocoder	2	24	10
		org.opengts.war.*	7	75	41
		org.opengts.cellid	1	4	2
		org.opengts.util	1	8	4
		org.opengts.db.dmtp	1	9	5
		org.opengts.servers.template	1	6	3
	org.opengts.db.dmtp	org.opengts.war.report.dmtp	2	30	16
		org.opengts.servers.gtsdmtp	2	43	24
		org.opengts.db.dmtp	3	33	16
		org.opengts.war.track.page.*	2	8	4
	org.opengts.db.tables	org.opengts.db.tables	52	675	368
		org.opengts.db	13	121	66
		org.opengts.tools	1	12	7
		org.opengts.servers.*	8	138	70
		org.opengts.db.dmtp	4	30	16
		org.opengts.war.*	27	444	242
		org.opengts.geocoder	2	12	6
		org.opengts.cellid	1	6	3
		Total		1982	1047 (52.82%)

paths and all feasible paths. If S be savings in test effort, then S can be defined in the following.

$$S = \frac{E_a - E_f}{E_a} \times 100\%$$

We obtain the value of S for *RAS* and *AMS* as 94.66% and 67.53%, respectively. Let us examine the average test effort savings per use case for *RAS* and *AMS*. For this, we compute the values of $UUCP_a$ (with all paths) and $UUCP_f$ (with all feasible paths) of individual use cases and thus, obtain the average value of $\%S$ as 54.34% (see Table 13). This implies that we can save on average 54.34% test effort per use case for *RAS* and *AMS* with prior detection of infeasible paths irrespective of the test environment.

Table 12
Computation of *UAW* for *RAS* and *AMS*.

System	Actor	Number of use cases	Weight	Partial *UAW*	Total *UAW*
RAS	Manager	3	1	3	7
	Staff	4	1	4	
	Manager	4	1	4	6
AMS	Agent	2	1	2	

4.5. Influence of locations of interaction patterns on infeasibility amount

We now investigate whether location of an interaction pattern has any influence on the number of infeasible paths arising out of

Table 13
Computation of % *S* for use cases of *RAS* and *AMS*.

System	Use case	Normal scenario (N)	Exceptional scenarios (E)	Total weight P_T	$UUCP_a$	$UUCP_f$	% *S*
RAS	Deliver Order	8 [2]	22 [7]	19 [5.5]	$19+1=20$	$5.5+1=6.5$	67.5%
	Process Order	672 [12]	6 [3]	675 [13.5]	$675+1=676$	$13.5+1=14.5$	97.85%
	Generate Statistics	7 [3]	1 [1]	7.5 [3.5]	$7.5+1=8.5$	$3.5+1=4.5$	47.05%
	Generate Bill	2 [1]	6 [3]	5 [2.5]	$5+1=6$	$2.5+1=3.5$	41.66%
	Pay Bill	4 [2]	2 [1]	5 [2.5]	$5+1=6$	$2.5+1=3.5$	41.66%
	Manage Item	49 [5]	1 [1]	49.5 [5.5]	$49.5+1=50.5$	$5.5+1=6.5$	87.12%
	Make Order	1 [1]	0 [0]	1 [1]	$1+1=2$	$1+1=2$	0%
	UUCW			762 [34]			
AMS	Book Ticket	2 [1]	4 [2]	4 [2]	$4+1=5$	$2+1=3$	40%
	Cancel Ticket	2 [1]	2 [1]	3 [1.5]	$3+1=4$	$1.5+1=2.5$	37.5%
	Compute Sale Commission	4 [2]	2 [1]	5 [2.5]	$5+1=6$	$2.5+1=3.5$	41.66%
	Pay Commission	6 [2]	4 [2]	8 [3]	$8+1=9$	$3+1=4$	55.55%
	Manage Show	17 [3]	33 [3]	33.5 [4.5]	$33.5+1=34.5$	$4.5+1=5.5$	84.05%
	Generate Show Statistics	17 [5]	1 [1]	17.5 [5.5]	$17.5+1=18.5$	$5.5+1=6.5$	64.86%
	UUCW			71 [19]			
	Average % S						54.34%

Table 14
Effect of the locations of integration colonies on the number of infeasible MM pairs

Use case	IC	Path-in	Path-out	(*A*) Path-in × Path-out	(*B*) Infeasible MM pairs	A Rank (AR)	B Rank (BR)	d= BR − AR	d^2
Delivery Order	IC_1	1	15	15	17	8	8	0	0
	IC_2	2	2	4	2	13.5	17	3.5	12.25
Manage Item	IC_3	2	10	20	22	5.5	5.5	0	0
	IC_4	10	2	20	22	5.5	5.5	0	0
	IC_5	1	339	339	183	1	1	0	0
Process Order	IC_6	2	156	312	162	2.5	2.5	0	0
	IC_7	26	12	312	162	2.5	2.5	0	0
Generate Bill	IC_8	1	4	4	4	13.5	12	-1.5	2.25
Pay Bill	IC_9	1	3	3	3	16	14	-2	4
Book Ticket	IC_{10}	1	3	3	3	16	14	-2	4
Cancel Ticket	IC_{11}	1	2	2	2	18	17	-1	1
Compute Sale Commission	IC_{12}	1	3	3	2	16	17	1	1
Pay Commission	IC_{13}	1	5	5	6	12	10	-2	4
Manage Show	IC_{14}	2	10	20	22	5.5	5.5	0	0
	IC_{15}	10	2	20	22	5.5	5.5	0	0
Generate Show Statistics	IC_{16}	1	8	8	3	9	14	5	25
	IC_{17}	2	3	6	6	10.5	10	-0.5	0.25
	IC_{18}	6	1	6	6	10.5	10	-0.5	0.25

that pattern. To identify the location of an interaction pattern, we need to know the position of integration colony where the interaction pattern occurs. For this, we consider two parameters called *Path-In* and *Path-Out*. For an *SIG* and one of its integration colony (say *IC*), the parameter *Path-In* of *IC* is defined as the number of incoming paths from the start node of *SIG* to the start node of *IC*, whereas the parameter *Path-Out* of *IC* is defined as the number of outgoing paths from the end node of *IC* to the end node of *SIG*.

Presence of some interaction pattern (MUX or NLC) in an integration colony causes infeasibility of at least one path in that integration colony. Therefore, the minimum number of paths that can be infeasible due to some interaction pattern in an integration colony is the product of its *Path-In* and *Path-Out*. In this regard, we may note that positions of two integration colonies are interchangeable if they have the same product value of *Path-In* and *Path-Out*. An example of such pair of integration colonies is *IC*1 and *IC*2 of *Manage Show* (see Figs. 4 and 5).

We now determine the product of *Path-In* and *Path-Out* of the integration colonies of all sequence diagrams and the number of infeasible MM pairs arising out of some interaction pattern in those integration colonies as shown in Table 14. Note that we have not included main integration colonies of all sequence diagrams as they represent core integral parts of the graph models and hence, are not subject to restructuring. From Table 14, we can observe the following. *First*, higher product value of *Path-In* and *Path-Out* of an integration colony implies larger number of infeasible MM pairs. *Second*, integration colonies belonging to an SIG and with the equal product values have the same number of infeasible MM pairs. The examples of such integration colonies are IC_3 and IC_4 of *Manage Item*, IC_6 and IC_7 of *Process Order* etc.

We now examine whether there exists some correlation between the product of *Path-in* and *Path-Out* of an integration colony and number of affected infeasible MM pairs. For this, we apply Spearman's correlation (Spearman, 2011) and find the value of correlation coefficient as 0.95, which implies a strong positive correlation between them. From this statistical observation, we can conclude that if we can scale down the value of the product of *Path-In* and *Path-Out* of an integration colony, then the number of infeasible paths can be reduced. One possible way to achieve this is

to reconstruct corresponding sequence diagram, that is, to move its integration colony downward/upward without affecting overall behavior.

4.6. Comparison of computation overhead for detecting infeasible paths using SIG and control flow graph

Let us consider an SIG with N scenarios, M MM paths, where a scenario contains an average P number of MM paths. To detect infeasible scenarios using SIG, we need to check their constituent MM paths with respect to MUX pattern and MM pairs with respect to NLC pattern. For this, it is necessary to estimate the number of MM pairs to be checked. Regarding this, we point out two observations: (1) each scenario contains one MM path from main integration colony; (2) no other MM paths can call MM path of the main integration colony. With this, we can imply that a scenario with P MM paths can call at most $(P-1)$ number of MM paths, which means that the maximum $(P-1)$ MM pairs can be contained in that scenario. Therefore, computation overhead (T_{SIG}) for detecting infeasibility of N number of scenarios using SIG is due to checking M number of MM paths and $M \times \alpha$ number of MM pairs, where α is a factor satisfying the inequality $M \times \alpha < N \times P$. That is,

$$T_{SIG} = M + M \times \alpha$$

Let us compute the overhead for detecting infeasible paths using an equivalent control flow graph with N number of scenarios. Unlike SIG, we need to check all scenarios of control flow graph individually, where the starts and ends of constituent MM paths are implicit. In case of control flow graph, computation overhead (T_{CFG}) is equivalent to verifying P number of MM paths and $(P-1)$ number of MM pairs for each scenario. In other words,

$$T_{CFG} = [P + P - 1] \times N = [2P - 1] \times N$$

The ratio of T_{CFG} and T_{SIG} is as follows.

$$\frac{T_{CFG}}{T_{SIG}} = \frac{(2P-1) \times N}{M + M \times \alpha} = Z$$

Since $P \times N > M \times \alpha$ and $N > M$, we find

$$(P+1) \times N > M + M \times \alpha$$

Applying this, we reduce Z as

$$Z > \frac{2P-1}{P+1} \quad (2)$$

For T_{CFG} to be higher than T_{SIG} (i.e. $Z > 1$), P must have the value greater than or equal to 2. Larger be the value of Z, higher would be reduction in overhead. For this analysis, we have not considered the cost to generate the SIG/control flow graph or even the cost to generate sequence diagram from source code since they would be the same for both types of models (be it control flow graph or SIG model).

To validate the above theoretical reduction, we compute the values of the parameters N, M, $M \times \alpha$, P, Z for all sequence diagrams of RAS and AMS as shown in Table 15. Please note that the value of Z (reduction in overhead for infeasibility detection) is high (7) for *Process Order* sequence diagram because the number of its scenarios as well as the average number of MM paths contained in each scenario are significantly larger than that of other sequence diagrams (say *Generate Bill, Pay Bill, Book Ticket, Cancel Ticket*) whose Z value lies between 1 and 2. From Table 15, we can also see that the value of P is greater than two for all sequence diagrams except *Generate Statistics* whose scenarios contain one MM path each. This substantiates that detection of infeasible paths incurs higher overhead using control flow graph than using SIG, when scenarios contain an average two or more number of MM paths.

Table 15
Computation of Z of different sequence diagrams of RAS and AMS.

Sequence diagram	N	M	$M \times \alpha$	P	Z
Deliver Order	30	17	30	2.26	2.24
Manage Item	50	22	48	2.6	3.00
Process Order	678	105	534	3.84	7.08
Generate Bill	8	6	8	2	1.71
Generate Statistics	8	8	0	1	1
Pay Bill	6	6	8	2.66	1.85
Book Ticket	6	5	6	2	1.63
Cancel Ticket	4	5	4	2.5	1.77
Compute Sale Commission	6	6	4	2.66	2.59
Pay Commission	10	7	10	2	1.76
Manage Show	50	22	48	2.6	3
Generate Show Statistics	18	13	20	2.83	2.54

4.7. Threats to validity

We now discuss the validity threats applicable to our approach.

1. *Construct threats*. This threat poses a concern about selection of right parameters used in our experiments. First parameter that we have measured is the percentage of infeasible paths similar to the existing work (Hedley and Hennell, 1985; Ngo and Tan, 2007). In addition to this, we have used another parameter (S) to determine the savings in test effort based on use case points (de Almeida et al., 2009; Nageswaran, 2001). This measurement is justified for refinement of test effort estimation after taking infeasible paths into consideration. Further, we have considered the product of two parameters: *Path-In* and *Path-Out* while investigating influence of location of an interaction pattern on the number of infeasible paths arising out of that pattern. This investigation resembles the study of effect of the number of predicates on path infeasibility (Malevris et al., 1990). In experiments, for revere engineering of three open source applications *SOOT, OpenGTS, Crimson*, we have selected those class methods which have atleast one MUX or NLC pattern. However, filtering out of the remaining class methods in those applications is not an issue because they do not contain any MUX or NLC pattern and hence, they are irrelevant to our experiment.
2. *Internal threats*. This threat concerns the occurrence of errors in the construction of graph models, detection of infeasible MM paths (pairs). To avoid this kind of errors, we have manually checked the graph models with corresponding sequence diagrams, infeasible MM paths (pairs) detected by our proposed algorithms with the actual infeasible MM paths (pairs). To establish the relationships among different parameters, we have computed Spearman's rank correlation coefficient (Spearman, 2011), which is indeed the standard practice followed in the literature.
3. *External threats*. This threat arises from a concern about generalization of case studies used in our experiments as well as the contexts where MUX and NLC interaction patterns occur. To study the effects of these two patterns, we have used two systems (RAS & AMS) of our own. In this regard, we may note that size of our systems (smaller compared to real-life commercial software) is not an issue so far experiments with design level sequence diagrams for use cases is concerned. This is because, complex use cases of large softwares are decomposed into a number of relatively simpler use cases following the standard *decomposition* principle (Cockburn, 2000). Due to this decomposition, the design specifications of such large softwares may contain a larger number of sequence diagrams for the use cases than our systems (RAS & AMS), but abstraction levels of their individual

Table 16
Comparison of characteristics of different infeasibility patterns.

Pattern	Main features	Applicable in	Similar to	
			MUX	NLC
Mutually exclusive decision	Among a set of actions, only one action is allowed to perform. Action refers to message sending, modification/definition of variables etc.	Java code + Sequence diagram	√	×
Identical decision	Conditions are independent and identical to each other.	Java code + Sequence diagram	×	×
Complement decision	Conditions are independent and complement to each other.	Java code + Sequence diagram	√	×
Check- then-do	Successful checking of some conditions activates other action to perform. The activation is enabled through setting a value to flag variable.	Java code	×	×
Looping-by-flag	Flag variable is used to terminate loop. For this, initialization and modification of flag variable is necessary.	Java code	×	×
Type infeasible call chain	Multiple caller calls a method with a parameter of polymorphic type, which can be of different types (i.e. Object).	Java code	×	×

sequence diagrams (which is measured by the product of number of objects and messages) are similar to our sequence diagrams. Further, in our empirical study we have used three open source applications *Soot, OpenGTS, Crimson* to mitigate the subjective bias. Note that these applications have also been considered by other researchers. However, the limited number of case studies used in our experiments should not be influencing factor when our two patterns occur in common situations. In this regard, we point out that MUX interaction pattern occurs while modeling the state-dependent object interactions. On the other hand, NLC interaction pattern occurs while modeling the object selection by means of delegation from one object to another. Since state-dependent behavior, object selection, delegation are the common characteristics of the object-oriented systems (Taylor, 1998), the occurrence contexts of MUX and NLC interaction patterns are not specific to some case studies, rather they refer to a general situation.

It is also necessary to reason how extensive is the use of sequences diagrams in industry and how reliable are the sequence diagrams as a representation of the source code (the real source of infeasible paths) in practice. This can be explained as follows. In industry, testing engineers validate systems under development to prevent defect leakage from market release. In order to achieve this, availability of effective test cases is important. It is possible when testing engineers have code level understanding of functions/requirements. Mere description of SRS and class diagram in design document are not enough to understand code. In this regard, we may note that sequence diagrams can help testing engineers to understand operation sequences realizing requirements. That is why, sequence diagrams are extensively used in industry.

In initial phase of project life cycle, large gap persists between design and code, which is gradually reduced with increasing details in design documents. To meet software process compliance of an organization, it is desirable to have detailed design and such detailed designs would depict abstract level representation of source code. In this regard, one might argue that sequence diagrams prepared in industry do not correspond the way the code is implemented. Please note that sequence diagrams are prepared at different levels of abstraction in design phase. At higher level of abstraction, interactions are captured among components/modules, whereas at lower level of abstraction, interactions are modelled among classes. When sequence diagrams are prepared at component/module level, they usually do not correspond the way code is implemented. On the other hand, sequence diagrams, modelled at class level, are meant to be followed during implementation in order to maintain consistency between design and code. Please note that the sequence diagrams at class level have been considered in our work.

5. Comparison of our work with the existing approaches

We now compare our technique first with the existing static techniques (Ngo and Tan, 2007; Souter and Pollock, 2001) and then with dynamic techniques (Bueno and Jino, 2000; Gong and Yao, 2010; Ngo and Tan, 2008). Static techniques are based on infeasibility patterns such as *identical/complement-decision* (Ngo and Tan, 2007), *mutually-exclusive-decision* (Ngo and Tan, 2007), *check-then-do* (Ngo and Tan, 2007), *looping-by-flag* (Ngo and Tan, 2007), *type infeasible call chain* (Souter and Pollock, 2001). Table 16 depicts the characteristics of these patterns, whether they are similar to our patterns and are applicable in Java and/or sequence diagram.

From Table 16, we point out the following observations.

(1) *Complement-decision* and *mutually exclusive decision* are similar to our MUX pattern. Note that conditions of *complement-decision* pattern are complement to each other and hence are mutually exclusive.
(2) None of the existing patterns have similarity with our NLC pattern. In fact, NLC pattern is new of its kind.
(3) The patterns namely *check-then-do, looping-by-flag, type infeasible call chain* are not applicable in sequence diagrams. This is because, *check-then-do, looping-by-flag* consist of code-artifacts related to the assignment of flag variable, which is not captured in sequence diagrams. On the other hand, for *type infeasible call chain* pattern, multiple callers would send different types of object references via a polymorphic typed parameter to callee. To detect infeasibility for this pattern, we need the information regarding actual type of the object references, which cannot be known from sequence diagrams.

Our approach is comparable with those existing approaches which use similar patterns such as *mutually-exclusive-decision* and *complement-decision*. If we apply the existing algorithms for these patterns to our graph model (SIG, which is the input of our approach as well as control flow graph), then they may incorrectly identify two control blocks as *mutually exclusive*, even when they belong to different class methods. This can be possible for the control blocks which are influenced by the variables with the same name and whose predicates are not satisfiable. On the contrary, our approach avoids this situation by identifying integration colonies (subgraph whose model elements in the same method scope) of SIG and processing them individually. Note that SIG for sequence diagram can contain model elements belonging to multiple method scopes.

Let us now compare infeasibility detection capability of our patterns with the existing patterns. As per our empirical study, the existing patterns namely *mutually-exclusive-decisive, complement-decision, check-then-do* patterns cause considerable number of

Table 17
Number of infeasible MM paths in three open-source applications.

System	Total MM paths	Number of infeasible MM paths		
		Complement-decision	Mutually-exclusive-decisive	Check-then-do
SOOT	4059	324	949	301
OpenGTS	2227	20	520	28
Crimson	488	30	42	89

Table 18
Number of infeasible MM pairs in three open-source applications.

System	Total MM pairs	Number of infeasible MM pairs	
		NLC	Type infeasible call chain
SOOT	8300	3656	574
OpenGTS	2250	1047	148
Crimson	1145	630	0

infeasible MM paths in Java programs. To investigate their effects in three applications namely *Crimson, SOOT*, and *OpenGTS*, we have determined the number of infeasible MM paths for them as shown in Table 17.

For our MUX pattern being similar to *complement-decision* and *mutually-exclusive-decisive*, we can identify as many infeasible MM paths as detected using these two existing patterns. Next, we find how much percentage of total infeasible MM paths for three patterns can be detected using MUX pattern, that is, 80% in *SOOT* (1273 out of 1574), 95% in *OpenGTS* (540 out of 568), 44% in *Crimson* (72 out of 161) (see Table 17). However, our approach cannot detect the infeasible paths for *check-then-do* pattern. This limitation is due to the use of model information where the constituent code artifact of *check-then-do* pattern, that is, the assignment of flag variable is not captured.

Next, we compare detection capability of the existing *type infeasible call chain* pattern and our NLC pattern with regard to infeasibility of MM pairs. To investigate their effect in *SOOT, OpenGTS, Crimson*, we have determined the number of infeasible MM pairs as shown in Table 18.

We find how much percentage of total infeasible MM pairs for two patterns can be detected using NLC pattern, that is, 86% in *SOOT* (3656 out of 4230), 87% in *OpenGTS* (1047 out of 1195), 100% in *Crimson* (630 out of 630). The other MM pairs are due to type infeasible call chain and they have been identified manually. Please note that the numbers of *type infeasible call chain* and NLC pattern instances found in three applications: *Crimson, OpenGTS*, and *SOOT* are [0, 6, 33], [47, 166, 132], respectively and larger difference in number of pattern instances would contribute higher percentage of infeasible MM pairs for NLC pattern. As we have already mentioned that infeasibility due to *type infeasible call chain* cannot be determined using the sequence diagrams, consideration of this pattern in our context is out of scope.

Let us compare our approach with the existing dynamic approaches (Bueno and Jino, 2000; Gong and Yao, 2010; Ngo and Tan, 2008). Bueno and Jino (2000) use dynamic information to decide lack in search progress over successive generations while finding test data for an intended path. Bueno et al.'s approach decides potential infeasibility of any arbitrary path, whereas our static approach decides infeasibility of those paths which satisfy certain infeasibility conditions. However, Bueno et al.'s approach results in large computation overhead due to use of genetic algorithm, whereas our approach incurs less overhead due to processing of model information only. Ngo and Tan (2008) decide path infeasibility with respect to few empirical properties for correlated conditional statements. Ngo et al. use program traces to check validity of the empirical properties, which cannot be done with our static analysis. On the other hand, Gong and Yao (2010) use sample values of branch outcomes to determine branch correlations related to path infeasibility. Note that these techniques (Gong and Yao, 2010; Ngo and Tan, 2008) target to detect infeasible paths for branch correlations which are identifiable at runtime. However, Ngo et al.'s and Gong et al.'s approaches are dependent on test generation process, whereas our approach is independent of such test generation process.

6. Conclusions

We have proposed two algorithms to detect infeasible paths in UML sequence diagrams. For infeasibility detection, we have constructed a novel graph model, called SIG from XMI of sequence diagram. The difference between SIG and control flow graph is that SIG subsumes control flow graph and additionally contains method scope information of interactions. Our experimental results show that the additional information in the SIG helps to reduce computation overhead for detecting infeasible paths.

Our infeasible path detection technique is based on two patterns: MUX and NLC. The NLC patterns occurs while modeling the object selection through delegation from one object to another. On the other hand, MUX pattern occurs while modeling state-dependent behavior of an object.

Applying our algorithms for infeasibility detection to the sequence diagrams of *RAS* and *AMS*, we identify 14%–80% infeasible MM paths, 33%–94% infeasible MM pairs, and 50%–97% infeasible scenarios. In case of three open-source applications: *Crimson, SOOT, OpenGTS*, we find 26%–42% infeasible MM paths, 49%–55% infeasible MM pairs.

Our empirical study shows that we can save on average 54.34% test effort per use case for *RAS* and *AMS* with prior detection of infeasible paths. This saving is realized by means of avoidance of the test data generation, test script preparation for the infeasible paths. Further, prior detection of infeasible test scenarios helps practitioners to prepare an effective test plan in the early phase of software development.

Our observation shows that if we can scale down the value of the product of *Path-In* and *Path-Out* of an integration colony, then the number of infeasible paths can be reduced. One possible way to achieve this is to reconstruct corresponding sequence diagram, that is, to move its integration colony downward/upward without affecting overall behavior. Note that we have placed integration colonies in sequence diagrams of our case studies with lower possible product value of *Path-In* and *Path-Out*. Further investigation is necessary to reconstruct an arbitrary sequence diagram such that its integration colonies can have the minimum product value of *Path-In* and *Path-Out*, which we plan to explore in our future research.

Appendix A

Algorithm 1: Detect_Infeasibility_MUX

Input: An integration colony and its MM paths;
Output: Set of infeasible MM paths;
Begin;
Initialize a stack *ST* as *null*;
Initialize *level* as *zero*; ; /*Mark hierarchy level of a control block*/
Initialize *Current_Node* as the *start node* of the integration colony;
; /*Step 1: Identify independent control blocks (*ICB*)*/
Visit the integration colony following *depth-first-search* traversal;
while *Current_Node* $\neq$ *null* **do**
  if *Current_Node represents the fragment start* (C_f^S) **then**
    Increment the value of *level* by *one*;
    Push the *Current_Node* along with its *level* into *ST*;
  end
  if *Current_Node represents the fragment end* (C_f^E) **then**
    if *a child node of* C_f^S *remains unexplored* **then**
      Backtrack to the unexplored child and resume traversal;
    end
    else
      ; /*Exploration of the control block is complete*/
      Add the control block (C_f^S, C_f^E) along with its *level* into *LIST*;
      Pop the top element from *ST*;
      Decrement the value of *level* by *one*;
    end
  end
end
; /*Step 2: Determine correlated control blocks*/
for *each pair of control blocks in LIST* **do**
  if *their level value matches* **then**
    Compute set of common predicate variables of control block pair;
    if *common variable set is* non-empty **then**
      Add the control block pair along with common variable set into LIST′;
    end
  end
end
; /*Step 3: Identify mutually exclusive control blocks*/
for *each pair of control blocks in LIST′* **do**
  Visit the subgraph of the integration colony encapsulating the pair of control blocks by using BFS traversal;
  if *a common predicate variable is modified in the subgraph* **then**
    Exclude the pair of control blocks from *LIST′* ; /*They can not be mutually exclusive control blocks*/
    continue;
  end
  Identify clause pair (Cl_1, Cl_2) which are associated with outgoing edges of start nodes of control block pair and reference the same variable;
  Conflict_Status = Call CheckConflictPredicateClausePair(Cl_1, Cl_2);
  if *Conflict_Status == false* **then**
    Exclude the pair of control blocks from *LIST′*;
  end
end
; /*Identify infeasible MM paths*/
for *each MM path of the input integration colony* **do**
  if *the MM path covers a pair of control blocks in LIST′* **then**
    Mark the MM path as *infeasible*;
  end
end
End

Algorithm 2: CheckContradictPredicateClausePair

Input: Two predicate clauses Cl_1, Cl_2
Output: Boolean value

Begin;
; /*This procedure checks contradiction between two predicate clauses, if any*/
Find pair of operators (OP_1, OP_2) for the predicate clause pair (Cl_1, Cl_2);
Find pair of right operands (RO_1, RO_2) for (Cl_1, Cl_2);
if *(OP_1 = "instanceof" && OP_2 = "=" && RO_2 = "null") || (OP_2 = "instanceof" && OP_1 = "=" && RO_1 = "null")* **then**
  return true;
end
Conflict_Status= Call CheckConflictOperators(OP_1, OP_2);
if *Conflict_Status* $= false$ & $RO_1 \neq RO_2$ **then**
  return true;
end
if *Conflict_Status* $= true$ & $RO_1 = RO_2$ **then**
  return true;
end
else
  return false;
end
End

Algorithm 3: CheckConflictOperators

Input: Two operators OP_1, OP_2
Output: Boolean value

Begin;
if *(OP_1 = ">" && (OP_2 = "≤" || OP_2 = "<"))* **then**
  return true;
end
if *(OP_1 = "<" && (OP_2 = "≥" || OP_2 = ">"))* **then**
  return true;
end
if *((OP_1 = "≥" || OP_1 = ">") && OP_2 = "<")* **then**
  return true;
end
if *((OP_1 = "≤" || OP_1 = "<") && OP_2 = ">")* **then**
  return true;
end
if *(OP_1 = "≠" && OP_2 = "=") || (OP_1 = "=" && OP_2 = "≠")* **then**
  return true;
end
return false;
End

Algorithm 4: Detect_Infeasibility_NLC

Input: *SIG*, set of integration colonies and their MM paths;
Output: Infeasible MM path pairs;

Begin;
; /*Step 1: Identify correlated integration colony pairs*/
for *each integration colony* IC_1 **do**
  for *each integration colony* IC_2 **do**
    Identify the pair of incoming and outgoing scope edges $(\hat{l}, \hat{m})$ which enclose IC_2 in the SIG;
    Find the origin node x of the scope edge $\hat{l}$;
    Find the target node y of the scope edge $\hat{m}$;
    if IC_1 *has the edge (x,y)* **then**
      Add the pair of integration colonies (IC_1, IC_2) along with the edge (x,y) into *LIST* ;
      ; /*IC_1 and IC_2 are correlated*/
    end
  end
end
; /*Step 2: Check infeasibility for concatenation of MM paths of correlated integration colonies*/
for *each pair of integration colonies in LIST* **do**
  Find the product set $\{M_1\} \times \{M_2\}$ where M_1 is an MM path of IC_1 and references a return variable *RVar*; M_2 is an MM path of IC_2 and is associated with a return value *RVal*;
  for *each pair of MM paths* (M_1, M_2) *in the product set* **do**
    Find the predicate clause *Cl* used for nullify check of *RVar* along M_1;
    if *Cl exists and RVar is not modified between x and Cl* **then**
      ; /*x is the place of return*/
      ConflictStatus = CheckConflict(Cl,RVal);
      if *ConflictStatus == true* **then**
        Mark the pair of MM paths (M_1, M_2) as infeasible;
      end
    end
  end
end
End

Algorithm 5: CheckConflict

Input: Predicate clause *Cl*, return value *RVal*
Output: Boolean value

Begin;
if *Cl is (RVar* $\neq null$*) and RVal* $= null$ **then**
  return true;
end
if *Cl is (RVar* $== null$*) and RVal* $\neq null$ **then**
  return true;
end
return false;
End

References

Ali, S., Briand, L.C., Jaffar-ur Rehman, M., Asghar, H., Iqbal, M.Z.Z., Nadeem, A., 2007. A state-based approach to integration testing based on UML models. Inform. Softw. Technol. 49 (11-12), 1087–1106.

Bandyopadhyay, A., Ghosh, S., 2009. Test input generation using UML sequence and state machines models. In: International Conference on Software Testing Verification and Validation. IEEE Computer Society, pp. 121–130.

Binder, R.V., 1999. Testing Object-Oriented Systems Models, Patterns, and Tools. Addison Wesley, Reading, MA.

Bodík, R., Gupta, R., Soffa, M.L., 1997a. Interprocedural conditional branch elimination. In: Proceedings of the ACM SIGPLAN 1997 conference on Programming Language Design and Implementation. ACM, New York, NY, USA, pp. 146–158. http://doi.acm.org/10.1145/258915.258929.

Bodík, R., Gupta, R., Soffa, M.L., 1997b. Refining data flow information using infeasible paths. In: Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Springer-Verlag New York, Inc., New York, NY, USA, pp. 361–377.

Booch, G., Rumbaugh, J., Jacobson, I., 2005. The Unified Modeling Language User Guide (Second edition). Addison Wesley.

Briand, L.C., Labiche, Y., 2002. A UML-based approach to system testing. Softw. Syst. Model. (Springer) 1 (1), 10–42.

Bueno, P.M.S., Jino, M., 2000. Identification of potentially infeasible program paths by monitoring the search for test data. In: Proceedings of the 15th IEEE International Conference on Automated Software Engineering. IEEE Computer Society, Washington, DC, USA, pp. 209–218.

Bullseye, 2015. http://www.bullseye.com/.

Clarke, L., 1976. A system to generate test data and symbolically execute programs. IEEE Trans. Softw. Eng. SE-2 (3), 215–222. doi:10.1109/TSE.1976.233817.

Cockburn, A., 2000. Writing Effective Use Cases. Addison Wesley.

de Almeida, E.R.C., de Abreu, B.T., Moraes, R., 2009. An alternative approach to test effort estimation based on use cases. In: International Conference on Software Testing Verification and Validation. IEEE CS, pp. 279–288.

Forgács, I., Bertolino, A., 1997. Feasible test path selection by principal slicing. In: Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Springer-Verlag New York, Inc., New York, NY, USA, pp. 378–394.

Fraikin, F., Leonhardt, T., 2002. SeDiTeC—Testing based on sequence diagrams. In: IEEE International Conference on Automated Software Engineering (ASE'02). IEEE Xplore, pp. 261–266.

Gallagher, L., Offutt, J., Cincotta, A., 2006. Integration testing of object-oriented components using finite state machines. Softw. Test. Verif. Reliabil. 16 (4), 215–266.

Gong, D., Yao, X., 2010. Automatic detection of infeasible paths in software testing. IET Softw. 4 (5), 361–370.

Hedley, D., Hennell, M.A., 1985. The causes and effects of infeasible paths in computer programs. In: Proceedings of the 8th International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 259–266.

IBMRationalUML. UML basics the sequence diagram 2014. http://www.ibm.com/developerworks/rational/library/3101.html.

Jorgensen, P.C., Erickson, C., 1994. Object-oriented integration testing. Commun. ACM 37 (9), 30–38.

Kundu, D., Samanta, D., Mall, R., 2012. An approach to convert XMI representation of UML 2.x interaction diagram into control flow graph. ISRN Softw. Eng. 2012, 1–22.

Kundu, D., Samanta, D., Mall, R., 2013. Automatic code generation from unified modelling language sequence diagrams. IET Softw. 7 (1), 12–28.

Kundu, D., Sarma, M., Samanta, D., Mall, R., 2009. System testing for object-oriented systems with test case prioritization. Softw. Test. Verif. Reliabil. 19 (4), 297–333.

Larman, C., 2002. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd, Prentice Hall Professional.

LCOV, 2015. http://ltp.sourceforge.net/coverage/lcov.php.

Malevris, N., 1995. A path generation method for testing LCSAJs that restrains infeasible paths. Inform. Softw. Technol. 37 (8), 435–441.

Malevris, N., Yates, D., Veevers, A., 1990. Predictive metric for likely feasibility of program paths. Inform. Softw. Technol. 32 (2), 115–118.

Mall, R., 2004. Fundamentals of Software Engineering. Prentice-Hall of India Pvt.Ltd.

Nageswaran, S., 2001. Test effort estimation using use case points. Qual. Week 1–6.

Nayak, A., Samanta, D., 2009. Model-based test cases synthesis using UML interaction diagrams. ACM SIGSOFT Softw. Eng. Notes 34 (2), 1–10.

Ngo, M.N., Tan, H.B.K., 2007. Detecting large number of infeasible paths through recognizing their patterns. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 215–224.

Ngo, M.N., Tan, H.B.K., 2008. Heuristics-based infeasible path detection for dynamic test data generation. Inform. Softw. Technol. 50 (7-8), 641–655.

NID IIT KGP Web, 2010. http://www.nid.iitkgp.ernet.in/isd/.

OCD Principle, 2011. The open-closed principle. http://www.objectmentor.com/resources/articles/ocp.pdf.

OMG UML Specification, 2011. OMG Unified Modeling LanguageTM, superstructure version 2.2. http://www.omg.org/spec/UML/2.2/Superstructure.

OpenGTS, 2013. http://opengts.sourceforge.net/.

Pilone, D., Pitman, N., 2005. UML 2.0 in a Nutshell. O'Reilly.

Pilskalns, O., Andrews, A., Ghosh, S., France, R., 2003. Rigorous testing by merging structural and behavioral UML representations. In: International Conference on the Unified Modeling Language. Springer LNCS, pp. 234–248.

Sarma, M., Kundu, D., Mall, R., 2007. Automatic test case generation from UML sequence diagram. In: Proceedings of the 15th International Conference on Advanced Computing and Communications (ADCOM'07). IEEE Computer Society, Washington, DC, pp. 60–67.

Soot, 2013. http://www.sable.mcgill.ca/soot/.

Souter, A.L., Pollock, L.L., 2001. Type infeasible call chains. In: IEEE International Workshop on Source Code Analysis and Manipulation. IEEE Xplore, p. 0196.

Spearman, 2011. Spearman's rank correlation coefficient. http://en.wikipedia.org/wiki/Spearman.

Taylor, D.A., 1998. Object Technology: A Manager's Guide. Addison-Wesley Professional.

The NAG Fortran Library, 2011. http://www.nag.co.uk/numeric/fl/fldescription.asp.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V., 1999. Soot—A java bytecode optimization framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative research (CASCON). IBM Press, p. 13.

Yan, J., Zhang, J., 2008. An efficient method to generate feasible paths for basis path testing. Inform. Process. Lett. 107 (3-4), 87–92.

Yates, D., Malevris, N., 1989. Reducing the effects of infeasible paths in branch testing. ACM SIGSOFT Softw. Eng. Notes 14 (8), 48–54.

Zar, J.H., 1972. Significance testing of the spearman rank correlation coefficient. J. Amer. Stat. Assoc. 67 (339), 578–580.

Zhang, J., Wang, X., 2001. A constraint solver and its application to path feasibility analysis. Int. J. Softw. Eng. Knowl. Eng. 11 (2), 139–156.

Zhao, R., Lin, L., 2007. An UML statechart diagram-based MM-path generation approach for object-oriented integration testing. Int. J. Appl. Math. Comput. Sci. 3 (1), 22–27.

Debasish Kundu received B.Tech. degree in computer science and technology from Kalyany University, India. He holds M.S. degree in information technology from Indian Institute of Technology Kharagpur, India. Presently, he is pursuing his Ph.D degree on the topic of model-based software testing in School of Information Technology, Indian Institute of Technology Kharagpur, India. His research interest includes Software Testing and Intelligent Systems.

Monalisa Sarma received her Ph.D. degree in computer science and engineering from Indian Institute of Technology Kharagpur, India. She holds M.S. (by research) and B.Tech. degrees both in computer science and engineering from Indian Institute of Technology Kharagpur, India, and North Hill University, India respectively. Presently, she is an assistant professor, Reliability Engineering Centre, India Institute of Technology Kharagpur. Prior to joining Indian Institute of Technology Kharagpur, she was working in the Department of Computer science and Engineering, Indian Institute of Technology Indore and Siemens Research and Devolvement, Bangalore, India. Her current research includes Software reliability and Big data analytics.

Debasis Samanta received his Ph.D. degree in computer science and engineering from Indian Institute of Technology Kharagpur, India. He holds M.Tech. and B.Tech. degrees both in computer science and engineering from Jadavpur University, Kolkata, India and Calcutta University, India, respectively. Presently, he is an associate professor, School of Information Technology, India Institute of Technology Kharagpur. His current research includes Human Computer Interaction, Model-Based Software Testing, Biometric-based System Security, and Big Data.