e-Informatica Software Engineering Journal, Volume 15, Issue 1, 2021, pages: 7–27, DOI 10.37190/e-Inf210101

Fixing Design Inconsistencies of Polymorphic Methods Using Swarm Intelligence

Renu George*, Philip Samuel*
*Department of Computer Science, Cochin University of Science and Technology
renugeorge@ceconline.edu, philips@cusat.ac.in
https://orcid.org/0000-0002-3276-2961

Abstract

Background: Modern industry is heavily dependent on software. The complexity of designing and developing software is a serious engineering issue. With the growing size of software systems and increase in complexity, inconsistencies arise in software design and intelligent techniques are required to detect and fix inconsistencies.
Aim: Current industrial practice of manually detecting inconsistencies is time consuming, error prone and incomplete. Inconsistencies arising as a result of polymorphic object interactions are hard to trace. We propose an approach to detect and fix inconsistencies in polymorphic method invocations in sequence models.
Method: A novel intelligent approach based on self regulating particle swarm optimization to solve the inconsistency during software system design is presented. Inconsistency handling is modelled as an optimization problem that uses a maximizing fitness function. The proposed approach also identifies the changes required in the design diagrams to fix the inconsistencies.
Result: The method is evaluated on different software design models involving static and dynamic polymorphism and inconsistencies are detected and resolved.
Conclusion: Ensuring consistency of design is highly essential to develop quality software and solves a major design issue for practitioners. In addition, our approach helps to reduce the time and cost of developing software.

Keywords: UML models, software design inconsistency, polymorphism, particle swarm optimization

1. Introduction

Today′s biggest industry is software industry in terms of manpower, complex interactions and changing tasks with evolving designs. The way people coordinate activities and work has seen a major transformation since the use of software in industries. With the increasing relevance of software in industries, software development has become more complex. Software changes are frequent due to evolution, agility and adaptability. Customized software is used to increase productivity in industries and quality of the software is a prime concern. Design and development of quality software is a major challenge for software developers and many a times, the process is manual. Artificial intelligence (AI) techniques can replace many of these manual efforts to make the development of software easier and cost effective.

Artificial intelligence replicates human decision making techniques to make machines more intelligent. Software development involves several complex human decision makings that deal with the task of designing, implementing and deploying complex systems. Software engineering problems can be represented as optimization problems. Search based software systems use optimization techniques and computational search techniques to solve problems in software engineering [1]. Although search based systems address

Submitted: 08 April 2020; Revised: 02 November 2020; Accepted: 14 December 2020; Available online: 02 February 2021

many problems in software requirements and design, design verification is not yet addressed [2, 3]. Particle swarm optimization (PSO) is an optimization technique based on population with computational intelligence [4]. Self Regulating Particle Swarm Optimization (SRPSO) is an improved version of PSO that provides optimum solutions by incorporating the best strategies for human learning [5]. We present an intelligent approach based on SRPSO to solve the inconsistency in polymorphic methods during the software system design.

The modelling language widely used for requirements modelling and documentation of the system is Unified Modeling Language (UML). UML models handle the complexity of the system by expressing different views with different diagrams that consist of a number of interrelated model elements. The interrelated design diagrams contain redundant information in the overlapping model elements. Hence, the probability of occurrence of design inconsistencies is more. The diagrams of a single system representing the static and dynamic aspects should be consistent and not contradictory [6]. Explicit mechanisms are required to verify the consistency of redundant information present across the diagrams [7, 8]. Generally, models are constructed for a specific application and the models are eventually implemented, usually in an object oriented programming language. Validating the models for consistency in the design phase guarantee that the design inconsistencies are not carried over to the code generation phase of software development. Automated consistency checking during the design phase ensures software quality, reduced development and maintenance cost and less time to market. Inconsistent design results in incorrect code, design rework, failure to meet timelines, and increase in cost of production.

Polymorphism is one of the key concepts that determine the quality of object oriented software systems [9]. Polymorphism enables different behavior to be associated with a method name. New methods with different implementation can be created with the same interface and the amount of work required to handle and distinguish different objects is reduced [9]. The result of execution of a polymorphic method depends on the object that executes the method and produces different results when received by different objects. The advantages of designing multiple methods with the same name make polymorphism an efficient approach during software design. We define an inconsistency related to object interactions in polymorphic and non-polymorphic methods: method-invocation inconsistency. Inconsistency exists if the method invocations are bound to a wrong class in the sequence diagram, i.e., the method is not invoked on an object of the class in which the method is defined. Inconsistencies in polymorphic method invocations cannot be identified by validating the method names as all polymorphic methods have the same name. Hence, detection of method invocation inconsistency in polymorphic methods requires more effort than non-polymorphic methods. As the design complexity increases, manual verification of inconsistencies in polymorphic methods is not practical. Intelligent techniques that require expertise are required to detect and solve the inconsistencies.

Method-invocation inconsistency occurs when a polymorphic or non-polymorphic method is invoked on a wrong object in the sequence diagram. The existing approaches of detecting inconsistencies in method invocations specified in [10–15] do not mention inconsistencies in polymorphic methods. Although polymorphism has a number of advantages, serious flaws may occur due to inconsistencies. Programmers may find it a challenging task to understand all the interactions between sender and receiver objects [16]. Understanding polymorphic codes is hard and therefore, fault-prone. Usually inconsistencies related to polymorphic method invocations are difficult to identify during testing phase. Separate tests are required for each polymorphic method binding. Identifying and testing all possible bindings of certain polymorphic references is difficult thereby increasing the chances of errors [17]. Inconsistent polymorphic behaviours may cause huge financial problems when detected.

Software design is prone to errors and design imperfections have a significant effect on software quality. Software failure can be attributed to var-

ious factors starting from requirements gathering to testing and poor quality management [18]. Inconsistencies in the design lead to the generation of defective software. One of the major activities in ensuring quality involves detection and removal of defects in the design. As the errors are carried over from the software design phase to the development phase, the cost incurred in fixing the error also increases. Defect detection during the design phase significantly prevents propagation of errors to further stages of software development and reduces development cost [19, 20]. Hence, code generation is based on consistent designs. This facilitates generation of software with fewer faults and improves the quality of the software generated. Development of software with fewer faults reduces the maintenance cost of the software. Cost increases with the delay in detecting and correcting the error. The cost of detecting defects after release is 30 times more than the cost of detecting defects in the analysis and design phase [19]. Therefore, inconsistency detection in the software design phase is inevitable for the development of accurate and quality software. We propose an intelligent approach to detect and fix inconsistencies during the design phase of software development. Inconsistencies are detected and handled with a fitness function by generating fitness values for each polymorphic and non-polymorphic method in the class diagram and sequence diagram. Inconsistencies are handled by maximizing the fitness values of methods subject to the constraint that the methods are invoked on the right classes. The proposed automated intelligent approach for consistency checking during the design phase facilitates generation of software with fewer faults, improves software quality, and reduces development and maintenance cost.

The organization of the paper is as follows. The related works in the areas of consistency checking and the various applications of PSO and its variants is presented in Section 2. Section 3 deals with the inconsistencies in polymorphic methods. The architecture of the consistency checking system is described in Section 4 and the implementation of the proposed approach is presented in Section 5. Results and discussion are presented in Section 6, threats to validity is presented in Section 7 and Section 8 concludes the paper.

2. Related work

The section presents the consistency handling techniques available in the literature and the applications of PSO techniques to find optimal solutions in software development and industries.

Inconsistencies in the design may result in the failure of a software project. The problems of establishing and maintaining consistency is discussed in [21]. The authors state that it is impossible to avoid inconsistency and more flexibility can be obtained by using tools that manage and tolerate inconsistency. A tool that detects inconsistency and locates the choices for fixing inconsistency is proposed in [10]. Model profiling is used to determine the model elements affected while evaluating a rule; a set of choices for fixing the inconsistency is proposed and the designer decides the choice for fixing the inconsistency. The method proposed in [11] fixes inconsistencies in class, sequence and statechart diagrams by generating a set of concrete changes automatically. The work focuses on deciding a method to fix inconsistencies. An approach that performs real time detection and tracking of inconsistencies in class, sequence and state chart diagrams is presented in [12]. Consistency checks are initiated during a model change.

The algorithm proposed in [13] performs consistency check on class and sequence diagrams based on the syntax specified and generates a sequence of Relational Calculus of Object Systems (rCOS) class declarations. Inconsistencies in well-formed class and sequence diagrams are detected with an algorithm based on breadth first search technique. Transformation, refactoring, merging or repair of models result in changes in the model and during consistency checking it may lead to performance problems. An automated approach with tool support to re-validate parts of the design rule affected by model transformation or repair is proposed in [22]. Although the paper mentions inconsistency in sequence and class diagrams, the focus is on improving the

performance of incremental consistency checking by identifying parts of the model affected by model changes. A prototype tool developed using a UML based approach to handle impact analysis is proposed in [14]. Consistency check is performed on the UML diagrams, difference between the two versions is identified and the model elements that are directly or indirectly affected by the changes are determined. The focus of the paper is on changes and its impact, i.e. which model elements are affected by the change. Instant detection of consistency between source code and design models is performed in [23] and a live report of the consistency status of the project is provided to the developers.

A classification of model repair techniques based on features is presented in [24]. The focus is on proposing taxonomy for model repair techniques and not on inconsistency detection and causes of inconsistency. The paper [15] proposes a method for automatic generation of executable and concrete repairs for models based on the inconsistency information, a set of generator functions and abstract repairs. An automated planning approach based on artificial intelligence is proposed in [25] to resolve inconsistencies. A regression planner is implemented in Prolog. The approach is restricted to detection of structural inconsistencies in class diagrams only.

A review of the consistency management approaches available in the literature is presented in [26]. The works described does not address inconsistencies related to polymorphic methods. Object Constraint Language (OCL) rules are specified for consistency checking of UML model in [27], the approach does not address polymorphic methods. Consistency rules to detect inconsistencies in method invocations between sequence and class diagrams are presented in [28], but no approaches are presented to detect and fix inconsistency. A method to detect inconsistencies between state diagrams and communication diagrams using the language Alloy is presented in [29].

Soft computing techniques find its application in providing solutions to problems in industries. PSO is used to minimize the cost of heating system [30], to assign applications to resources in the cloud [31], in job-shop scheduling [32], in networking [33], power systems [34, 35], signal processing [36], control system [37] and many more. PSO is also applied to find effective solutions to problems in software development. PSO is applied to UML class diagram and an algorithm for class responsibility assignment problem is presented in [38]. The PSO method reassigns the attributes and methods to the different classes indicated in the class diagram. The application of SRPSO and PSO in detecting and resolving inconsistencies in class attribute definitions is presented in [39, 40]. The fitness value determines the consistency of attributes and the PSO and SRPSO algorithm iterates to fix inconsistency by optimizing the fitness value of attributes. The papers deal with fixing inconsistencies in attribute definitions only. The performance of SRPSO algorithm is better than PSO in term of statistical evaluation parameters and convergence. An SRPSO based approach to fix state change inconsistencies in state diagrams and sequence diagrams is proposed in [41]. Inconsistencies are detected and fixed with a fitness function.

An optimization based approach using PSO and simulated annealing to find transformation fragments that best cover the source model is proposed in [42]. PSO is applied to achieve high structural code coverage in evolutionary structural testing by generating test cases automatically [43]. Parameter estimation using PSO to predict reliability of software reliability growth models (SRGM) is described in [44]. During testing, faults are detected and a mathematical model SRGM, models the properties of the process. A comparative study of metaheuristic optimization framework is proposed in [45] and the study states that a wider implementation of software engineering practices is required.

The application of PSO in diverse areas of engineering has yielded better results over existing methods, but works that describe the application of PSO in the design phase for software design consistency checking is rare. Although consistency checking of UML models is a widely discussed problem and different techniques to detect and fix inconsistencies are available in the literature, techniques that perform consistency checking of polymorphic methods are rarely re-

ported. We present an intelligent approach that detects inconsistencies with a fitness function. Inconsistencies are fixed by remodelling the sequence diagram method invocations during iterations of the SRPSO algorithm. Our approach efficiently detects and fixes the inconsistencies.

3. Inconsistencies in polymorphic methods

Polymorphism is an important feature of object oriented programming that provides simplicity and flexibility to the design and code. It enables different behaviour to be associated with a method name. Polymorphism keeps the design simple, flexible and extensible [46]. New methods with different implementation can be created with the same interface and the amount of work required to handle and distinguish different objects is reduced. Each polymorphic method has a class specific way to respond to a message. Polymorphic methods execute different subroutines depending on the type of object they are applied to. Inconsistency occurs if the method is invoked on a wrong class. Two methods of implementing polymorphism are (a) static binding: methods have the same name, different signature and different implementation (b) dynamic dispatch: methods have the same name, same signature and different implementation [47]. Static binding occurs with method overloading at compile time and the method to be invoked is determined from the signature of the method call. Dynamic dispatch is related to inheritance hierarchy. Method overriding provides a superclass/subclass inheritance hierarchy allowing different subclass implementation of inherited methods [48, 49]. The overriding methods represent different functionalities and require different algorithms [50]. The exact method to which the method call is bound is known only at run time. Method overriding is implemented with dynamic dispatch [49].

Inconsistency in UML models occurs when two or more diagrams describe different aspects of the system and they are not jointly satisfiable. Any method invoked on an object in the sequence diagram should be defined in the class instantiated by the receiving object. The rule is part of the UML well-formedness principle. There is scope for many subtle errors with polymorphism since a method name occurs in more than one class. The exact operation to be performed is determined from the data types of the arguments in static polymorphism. The same signature is used by more than one class in dynamic polymorphism and determining whether the correct method is invoked in the sequence diagram is an issue. Understanding polymorphic codes is hard and therefore fault-prone [16]. Hence, inconsistency detection during the design phase has become inevitable for the development of accurate software [16]. We propose an intelligent approach using SRPSO algorithm to detect and fix method-invocation inconsistency in polymorphic methods. Method invocation inconsistency is identified from the signatures of the class diagram and sequence diagram methods in static polymorphism. The method signatures are the same for all polymorphic methods in dynamic polymorphism and hence more difficult. Inconsistency is detected from the guard condition for message invocation in the sequence diagram and precondition for the method in the class diagram.

The inconsistencies are illustrated with the UML models 3DObject and ThreeDObject represented in Figures 1 and 2, respectively. The class diagram and sequence diagram for the UML model 3DObject is represented in Figure 1 . The model provides an example of static polymorphism. The class diagram consists of 4 classes. A generalized class ThreeDShape is defined with an attribute Area of type float. The classes Sphere, Cuboid and Cylinder are specializations of the class ThreeDShape. The methods computeArea() and perimeter() defined in the classes Sphere, Cuboid and Cylinder are polymorphic since methods with the same name and different signature are defined. The method vertices() defined in the class Cuboid is non-polymorphic. The sequence diagram represents the method invocations to compute the area of the objects. The class Cuboid has a method computeArea(l, b, h) with signature computeArea(int, int, int). Similarly, the signatures of the method computeArea() defined in the classes Sphere

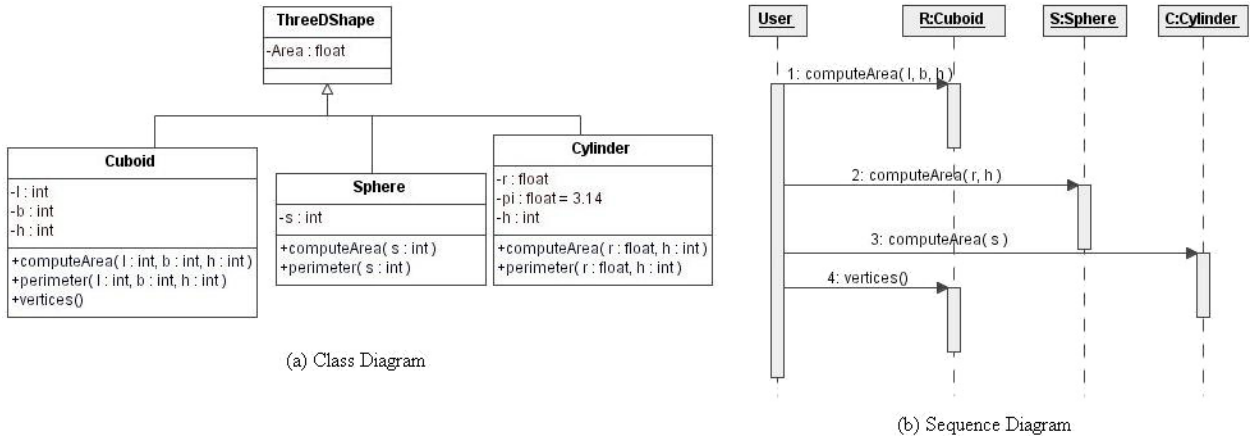


Figure 1. Class Diagram and Sequence Diagram for UML Model for 3DObject

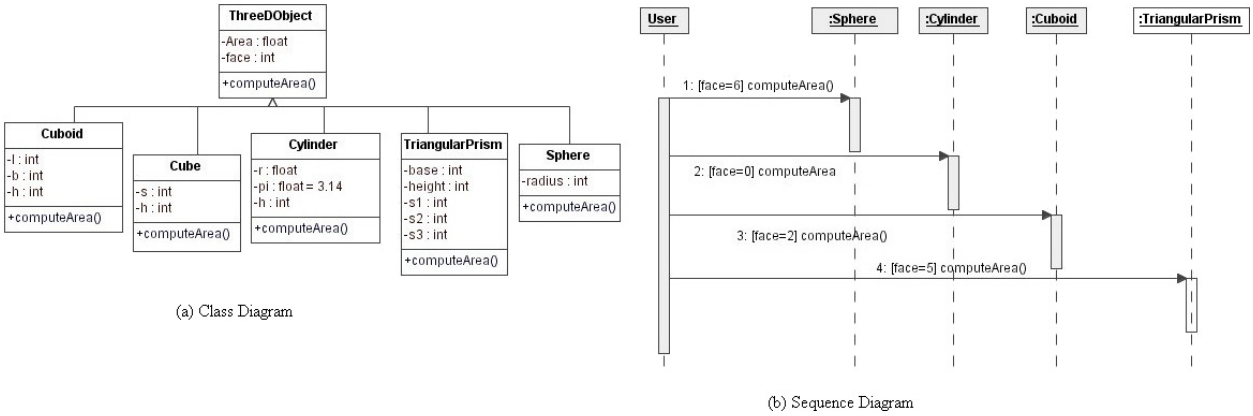


Figure 2. Class Diagram and Sequence Diagram for UML Model for 3DObject

and Cylinder are computeArea(int) and computeArea(float, fint), respectively. The signatures of the method computeArea() invoked on the objects of classes Cuboid, Sphere and Cylinder in the sequence diagram are computeArea(int, int, int), computeArea(float, int) and computeArea(int). The invocations of the polymorphic method computeArea(l, b, h) and the non-polymorphic method vertices() are consistent whereas the invocations of the polymorphic methods, computeArea(s) and computeArea(r, h) are inconsistent. The inconsistencies, if unnoticed will result in a wrong value for area. Inconsistencies are detected by computing the fitness values of methods. The fitness value computation to detect inconsistency is represented in Table 1.

A UML model ThreeDObject representing dynamic polymorphism is depicted in Figure 2. A method computeArea() and two attributes *Area* and *face* are defined in the class ThreeDObject. The method is overridden in the child classes since the method of computing area depends on the shape of the object. The attribute *face* represents the number of faces possessed by

Table 1. Fitness values of methods in UML Models 3DObject and ThreeDObject

Method name	CD Class	SD Class	Fitness value	UML Model
computeArea(r, h)	Cylinder	Sphere	0.9375	3DObject
computeArea(l, b, h)	Cuboid	Cuboid	1	3DObject
vertice()	Cuboid	Cuboid	1	3DObject
computeArea(s)	Sphere	Cylinder	1.11	3DObject
computeArea()	Cylinder	Cylinder	1	ThreeDObject

an object. Sphere has no face, Cylinder has two faces, Cuboid and Cube have 6 faces and TriangularPrism has 5 faces. Constraints are defined for the methods and expressed as preconditions in object constraint language. The preconditions for the method computeArea() in the classes Cuboid, Cube, Sphere, TriangularPrism and Cylinder state that the value of the attribute face should be equal to 6, 6, 0, 5 and 2, respectively. As the signatures of all the methods involved in dynamic polymorphism are the same, it is impossible to detect inconsistency by comparing the method signatures. The guard conditions and the preconditions are compared and method invocation inconsistency is detected with the method computeArea() invoked on the objects of classes Cuboid, Sphere and Cylinder. The method computeArea() invoked on the object of class Sphere should satisfy the guard condition face = 6 which is not true resulting in run time errors. The invocation of the method computeArea() on the object of class TriangularPrism is consistent as the value of the attribute face in the guard condition and precondition is equal to 5.

The inconsistent method invocations in Figures 1 and 2 result in wrong value for Area. If the models are used in the cost estimation of buildings, the estimated cost will be computed with wrong values of area. The cost estimation will produce a wrong value affecting the feasibility of the project. Since the design errors are propagated to the code generation phase, the software generated will have errors. Identifying the source of errors in the code and fixing the errors is more difficult, time consuming and costly than detecting the errors in the software design. Errors detected in the testing phase may delay the software project. The errors identified during the testing phase or after delivery of the software product increases the time to market as well as development and maintenance cost of the software.

4. Architecture of the consistency handling system

PSO is an intelligent algorithm that can be used in scientific and engineering area [51]. Consistency checking is formulated as an optimization problem with a maximizing fitness function that operates on the diagram specification. An optimization problem maximizes or minimizes a fitness function subject to the condition that the constraints are satisfied. In our approach the fitness function represents the consistency and completeness of method invocations. The aim of the SRPSO algorithm is to optimize the consistency of polymorphic method invocations in sequence diagram subject to the constraint that the methods are invoked on the right classes. The SRPSO algorithm is preferred because it does not require transformation of models and can be directly applied on UML model specification. The inconsistent particles are guided by the best particles to achieve consistency and hence search speed is high [52].

The architecture of the system to perform consistency checking using SRPSO is described in Figure 3. The algorithms are implemented in Java running on a windows platform. The con-

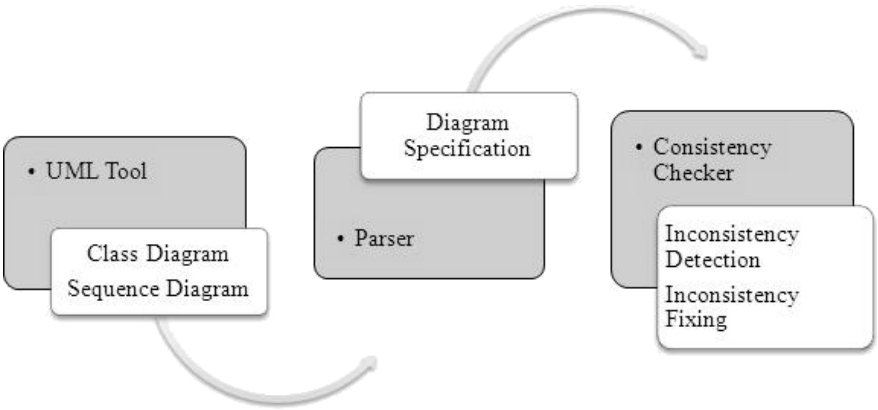


Figure 3. Architecture of Consistency Checking System

sistency checking system comprises of UML tool to model the requirements, parser to generate diagram specification and consistency checker to detect method-invocation inconsistency and fix the inconsistency using SRPSO algorithm.

4.1. UML tool

The requirements of the system to be designed are gathered and modelled into graphical representations using a UML tool. Several UML modelling tools like Magic Draw, Rational Software Architect, Agro UML, Papyrus etc. are available for modelling software. Our method can be integrated with any tool that give XMI format. The models are saved in XMI format. The static and dynamic aspects are represented using class diagram and sequence diagram. Class diagrams represent the information regarding the classes required to implement a system and the type of relationship that exists between the classes. The attributes and operations describe the properties and behaviour of the objects of a class. Preconditions associated with method invocations are also represented. The preconditions of the overridden methods in the super class and subclass are different [50]. Sequence diagram represents the dynamic aspects by portraying the interactions in the form of messages/ methods between objects and the ordering of the interactions to produce a desired outcome. Polymorphic behaviour can be represented using a sequence diagram by controlling the polymorphic invocations with guard conditions.

4.2. Parser

The parser parses the UML model and produces specifications of the diagrams. We have used the Document Object model (DOM) parser to parse the diagrams saved in XMI format. A class diagram specification comprises of the classes, the type of association between the classes, attributes and methods of each class and the preconditions for method invocations. The sequence diagram specification consists of the objects in the sequence diagram, messages, sender and receiver of each message, the guard conditions on the message invocations and the order of method invocations.

4.3. Consistency checker

The design inconsistencies in polymorphic method invocations are detected by consistency checker module. Although the focus is on detection of inconsistencies in polymorphic methods, the algorithm detects inconsistencies in polymorphic and non-polymorphic methods. The specifications of class and sequence diagrams are input to the consistency checker. Inconsistencies are detected by a fitness function. The inconsistency is resolved by reassigning methods with the SRPSO algorithm. Consistency checking is a two-step process: a) inconsistency detection and b) inconsistency fixing.

4.3.1. Inconsistency detection

The fitness function, f_s computes the fitness value of the methods to detect inconsistency. The fitness value is computed as a function of the class name, method signature and properties of the method. The sequence diagram method is defined in terms of its properties like name, id, parameters, sender, receiver, guard and a number that represents the message order. Each method has a specific value for a property (denoted as weight) and each position in the vector corresponds to one property of the method. The values for the properties are set as 5, 3, 5, 5, 5, 4 and 3, respectively. The fitness function f_s computes the fitness value of each sequence diagram method. A method invocation is classified as inconsistent if the fitness value is not equal to one. The fitness function is defined with equation 1 as

$$f_s = \frac{\left(\sum_{i=1}^{m} t_i * w_i) * (w_n + w_{cs} + \sum_{k=1}^{n} w_k * p_k\right)}{W * \left(\sum_{j=1}^{q} w_j * p_j\right)} \tag{1}$$

where t_i represents the property i of method specification, w_i represents the weight of the property t_i, w_n represents the weight value as-

sociated with the method name n, p_j and w_j represents the position and weight of parameter j of the method n in the class diagram, p_k and w_k denotes the position and weight of the parameter k of the method n in the sequence diagram, w_{cc} represents the class name of the method in the class diagram, w_{cs} represents the class name of the object on which the method is invoked in sequence diagram and W represents the weight assigned to a complete method specification. The value of W is set as 30. A complete method specification has values for all its properties. Unique values are assigned as the weights for method names, class names and data type of parameters. Distinct method names, data types and classes have distinct weight. All polymorphic methods have the same weight value for name and any numerical value can be selected to correspond to w_n. The UML model in Figure 1 has a polymorphic method with name computeArea. The value of w_n is set as 5. The value of w_n for the method names vertices() and perimeter() are set as 3 and 4, respectively. The classes are assigned weights in the range $[1 \dots n]$ where n is the number of classes. Each class has a unique weight. A class name present in both the class diagram (w_{cc}) and sequence diagram (w_{cs}) has the same weight. The UML model in Figure 1 has four classes Cuboid, Sphere, Cylinder and ThreeDShape and the weight values for the classes Cuboid, Sphere, Cylinder and ThreeDShape are 1, 2, 3 and 4, respectively. The weight value of parameter is defined as the number of bytes required for the storing the data type of the parameter and the weights of char, int and float are defined as 1, 2 and 4, respectively. The weights assign unique numerical values to the method name, class name and data types of the parameters. The fitness value computation for the methods in the sequence diagram of Figure 1 and 2 is illustrated in Table 1.

Method invocation inconsistency is detected with the methods computeArea(r, h) and computeArea(s) since the fitness values of the methods are not equal to one. The methods computeArea(l, b, h) and vertices() are consistent since the fitness values are equal to one. Although inconsistency is detected from the fitness value in static polymorphism, fitness value alone does not reveal inconsistency in dynamic polymorphism. Irrespective of the object on which the method is invoked, the fitness value of the method computeArea() in the UML model ThreeDObject is one since the method is overridden in the child classes. Hence, validation of the guard condition and method precondition is necessary. The guard condition and precondition are represented as tuples consisting of attribute, operator-value pairs. Depending on the precondition, there can be more than one operator-value pair. The tuples are compared to identify inconsistency. The tuple corresponding to the guard condition for the method computeArea() in Figure 2 in the sequence diagram invoked on the object of class Cylinder is (*face*, (=, 0)). The tuple representation for the precondition of the method computeArea() in the class Cylinder is (*face*, (=, 2)). There is a mismatch in the value of the attribute *face* and method invocation inconsistency is detected.

4.3.2. Inconsistency fixing with SRPSO

The inconsistency is resolved by identifying the right classes and remodelling the sequence diagram by replacing the inconsistent method invocations with consistent method invocations using SRPSO. To identify the right class, we compute the cohesion of the attributes of the inconsistent method to all the classes in the class diagram. The inconsistent method is reassigned to the class with the highest cohesion value. The cohesion value between the method attributes and the class attributes is computed for each method-class pair. The method attributes MA(m) of method m are derived from the parameters of the method. The class attributes of class C, CA(C) are obtained from the class specification. The cohesion value of a method m to class C is computed using equation 2 as

$$\text{cohesion}(m, C) = \frac{n(\text{CA}(C) \cap \text{MA}(m))}{n(\text{MA}(m))} \quad (2)$$

where CA(C) represents the attributes defined in class C, MA(m) represents the attributes of method m and n represents the number of at-

tributes. The SRPSO algorithm iterates until all the method definitions are complete and consistent. The sequence diagram is remodelled during iterations of the SRPSO algorithm. With static binding, the cohesion value determines the class to which an inconsistent method is to be reassigned whereas in dynamic binding, the cohesion value and guard condition together determine the class to which the method belongs.

5. Consistency handling with SRPSO algorithm

SRPSO is a bio-inspired metaheuristic technique that can provide better results than exact techniques even with increased size of search space. Metaheuristic techniques are more effective in finding software errors utilizing less number of resources when compared with exact techniques [53]. SRPSO is an intelligent, optimization procedure in which the solution space contains a swarm of particles and the optimum value is attained by an iterative process of updating generations. The particles occupy a position in the solution space. They have a velocity, a fitness value and the particles update their velocity and position based on the direction of a) the previous velocity, b) the personal best position and c) position of the global best [54]. The fitness function determines how close a particle is to the optimum solution by computing the fitness value. The velocity directs the movement of particles and during each iteration of the SRPSO algorithm the particles compute their new velocity. The position is updated using the new velocity and with each position update the particle moves to a better position. The process is iterated until an optimum solution is reached.

5.1. Fitness function

The fitness function is an integral part of the SRPSO algorithm and it determines how close a particle is to the optimum solution. We have defined a maximizing fitness function, f_s to detect and fix method invocation inconsistency. The fitness function is defined with equation 1. The consistency and completeness of a sequence diagram method is computed using the fitness function. The invocations of inconsistent methods are removed from the sequence diagram and the inconsistent methods are added to the set of inconsistent methods (IM).

5.2. Particle creation

The search space of the SRPSO algorithm is initialized with particles. The proposed approach focuses on inconsistency in polymorphic and non-polymorphic method invocations and hence, the methods invoked in the sequence diagram are treated as particles. A sequence diagram method is specified using a set of properties and is represented as a vector. The representation of the sequence diagram method (SeqM) is SeqM = [name id param sender receiver guard number]

The representation of SeqM consists of a method name, a unique xmi id, the parameters, sender class of the method, receiver class of the method, guard condition for method invocation and number representing the message order in the sequence diagram. Each method has a specific value for a property and each position in the vector corresponds to one property of the method. The values for the properties are fixed as 5, 3, 5, 5, 5, 4 and 3, respectively. Any numerical value can be used to represent a property. The restriction is that the value of W should be equal to the sum of the numerical values assigned to the properties. The inconsistent methods in the set IM are represented as particles.

5.3. Velocity and position update

The particles in the search space are characterized by a position and velocity. A particle is defined in terms of its properties and in our approach; the position of a particle represents the number of properties defined for the particle. The specification of the inconsistent particle initially has only one property, *name* and hence, the value of position is one. As the iteration progresses, depending on the value of velocity the particle specification will be updated with its properties like id, sender, receiver etc. The number of properties of the par-

ticle to be updated in one iteration is determined by the value of velocity. If the value of velocity is one, one property will be added to the particle specification and position will be incremented by one. Velocity of the best particle is computed with equation 3, velocity of the rest of the particles with equation 4, position is updated using the equation 5 and inertia weight with equation 6.

$$V_k(t+1) = \omega_k + V_k(t) \tag{3}$$

$$V_k(t+1) = \omega_k + V_k(t) + a_1 * r_1 * (\text{pBest}_k - X_k(t)) + a_2 * r_2 * p_{so} * (\text{gBest} - X_k(t)) \tag{4}$$

$$X_k(t+1) = X_k(t) + V_k(t+1) \tag{5}$$

$$\omega_k(t) = \begin{cases} \omega_k + \eta\Delta\omega & \text{for best particle} \\ \omega_k - \Delta\omega & \text{otherwise} \end{cases} \tag{6}$$

where $V_k(t)$ represents the velocity of particle k at time t, a_1 and a_2 are the acceleration coefficients, r_1 and r_2 are the random numbers, $X_k(t)$ represents the position of particle k at time t, pBest_k represents the personal best of particle k and gBest the global best of all the particles in the swarm, p_{so} is the perception for the social cognition, ω_k is the inertia weight of the k^{th} particle, $\Delta\omega = (\Delta\omega_I - \Delta\omega_F)/Itr$, $\Delta\omega_I = 1.05$ and $\Delta\omega_F = 0.5$, Itr is the number of iterations, and $\eta = 1$ is the constant to control the rate of acceleration.

5.4. Stopping criteria

The SRPSO algorithm resolves method invocation inconsistency. The algorithm iterates until method invocation inconsistency is resolved or the number of iterations reaches a maximum limit. We have defined a variable method consistency count (MCC) that keeps track of the number of methods with consistent and complete invocations. MCC is incremented if fitness value of a method is equal to one. If MCC is equal to the number of inconsistent methods in the set IM, method invocation inconsistency is resolved.

5.5. Algorithm

The consistency checking algorithm for polymorphic methods is outlined in algorithm 1.

Algorithm 1: Consistency Checking

Begin
Initialize SRPSO parameters, IM = ϕ
for each method, m in sequence diagram **do**
identify sender class, SC(m) and receiver class, RC(m)
compute $f_s(m)$ with equation 1
Case I: $f_s(m) == 1$
  if guard conditions do not match
    IM = IM $\cup$ $\{m\}$
  endif
Case II: $f_s(m) \neq 1$
  IM = IM $\cup$ $\{m\}$
endfor
identify the receiving class
for each method, m in set IM **do**
  identify method attributes, MA(m)
  for each class, C_i in class diagram **do**
    determine class attributes CA(C_i)
  endfor
  compute cohesion(m, C_i)
  RCnew = C_j where C_j = max(cohesion(m, C_i), i = 1 to number of classes
  delete sequence diagram invocation for the method m
endfor
Initialize the search space with particles in the set IM
repeat
  for each particle k in IM **do**
    compute fitness of particle k
    if ($f_s(X_k) > f_s(\text{pBest}_k)$)
      $\text{pBest}_k = X_k(t)$
    endif
    if ($f_s(X_k) > f_s(\text{gBest})$)
      gBest = $X_k(t)$
    endif
    Compute inertia weight using equation 6
    Update velocity of gBest particle using equation 3
    for each particle except gBest particle **do**
      Generate random number, r between 0 and 1
      if ($r > 0.5$)
        $p_{so} = 1$ **else** $p_{so} = 0$
      endif
      Compute velocity using equation 4
    endfor
    if($V_k(t+1) > 1$

$V_k(t+1) = 1$ **else** $V_k(t+1) = 0$
endif
Update position using equation 5
if $(f_s(X_k(t+1)) == 1)$
Increment MCC
endif
endfor
Increment iteration count, *Itr*
until *Itr* = maxCount or MCC = number of inconsistent methods
End

The algorithm initializes the search space with particles and SRPSO parameters. The acceleration coefficients are set as 1.49445 [5], *Itr* is initialized as zero, W is set as 30 and maxCount is set as 35. The set IM is initialized to null. The algorithm computes the fitness values of methods. The guard conditions and preconditions of methods are also validated. The inconsistent method names are added to the set IM and the inconsistent method invocations are removed from the sequence diagram.

To fix the inconsistency, the inconsistent methods in the set IM are treated as new particles and the position of the particles are initialized. The cohesion of each method in the set IM to the different classes of the class diagram is computed to identify the new receiving class, RCnew. The class with the maximum cohesion value is identified as RCnew. The receiving class is identified from the precondition and cohesion value in dynamic polymorphism.

The newly created particles are inconsistent since its properties are not completely specified. Initially, all the inconsistent particles have only one property, its name. The fitness values of the particles in their current position are computed using the fitness function, f_s. If the current position is better than the personal best (pBest) position of the particle, the personal best position of the particle is updated. If the current position is better than the global best (gBest) position of all the particles in the swarm, the global best position is updated. New velocity and position of the particles are computed. Depending on the velocity value, properties such as id, sender, receiver etc. are added to the particle specification. The velocity component determines the number of properties to be updated in one iteration. If the fitness value is equal to one, the method consistency count is incremented. The velocity, position, fitness value, pBest and gBest values of all the particles in the set IM are updated during an iteration of the algorithm. The iteration count is also incremented. The SRPSO algorithm iterates until the method consistency count is equal to number of particles in the set IM or maximum number of iterations is reached. The updation of the properties of the inconsistent particles ensures that inconsistencies are resolved and the method specification is complete. The SRPSO algorithm efficiently detects and resolves inconsistency.

6. Results and discussions

The consistency checking algorithm is applied to the UML models to detect method invocation inconsistency. The UML model in Figure 1 contains the polymorphic method computeArea. The method-invocation inconsistency detection module detects two inconsistent methods: computeArea(r, h) and computeArea(s) by computing the fitness values of the methods. The inconsistent methods are added to the set IM and the sequence diagram invocations of the inconsistent methods are removed. The attributes required for the implementation of the method are derived from the parameters of the method. The cohesion of the method attributes to the different classes in the class diagram is computed. The cohesion values of the inconsistent methods to different classes are represented in Table 2.

Table 2. Cohesion Value for UML Model 3DObject

Method	Class Name		
	Cube	Cuboid	Cylinder
computeArea(r, h)	0.0	f 0.5	1.0
computeArea(s)	1.0	0.0	0.0

The class Cylinder has the highest cohesion value for the method computeArea(float, int) and the class Cube has the highest cohesion value for the method computeArea(int). The receiving class of the inconsistent method computeArea(r, h) is identified as class Cylinder

and the new receiving class of the method computeArea(s) is identified as class Cube. The sequence diagram methods are specified with a set of properties. On detecting inconsistency, the properties related to the method invocation of the inconsistent methods are also deleted. The inconsistency is fixed during iterations of the SRPSO algorithm. During each iteration of the SRPSO algorithm, the specification of the sequence diagram method in the set IM is updated by adding the properties of the methods. The approach ensures that method invocation inconsistency is resolved and the method specification is complete. The algorithm terminates when MCC becomes equal to the number of inconsistent methods or when *Itr* reaches the maxCount.

A graph representing the fitness value of the inconsistent methods computeArea(r, h) and computeArea(s) during different iterations of the SRPSO algorithm with acceleration coefficient values equal to 1.49445 is represented in Figure 4. The method computeArea(s) has a fitness value 0.0925 during the first iteration of the algorithm. As the iteration count increases, the fitness value of the particle increases. In iteration 8, the fitness values of the two inconsistent particles become one and the UML model 3DObject is consistent in terms of polymorphic method invocation and specification. The fitness value of the inconsistent method in the UML model ThreeDObject is represented in Figure 5. The algorithm is implemented with acceleration coefficient values equal to 1.49445 and converges in 8 iterations.

The result of implementation of the algorithm is represented in Figure 6. The XMI parser identifies the methods present in each class of the class diagram. The method computeArea() is overridden in all child classes. The signatures of the class diagram method and sequence diagram methods are compared and no inconsistency is detected. A further validation of guard conditions and preconditions identifies three inconsistent methods due to wrong guard conditions. The SRPSO algorithm resolves the inconsistencies in 8 iterations and the sequence diagram specification has consistent method invocations with guard conditions matching the preconditions. The execution time of the algorithm is 875 ms.

The UML model Deposit and Payroll System used for evaluating the algorithm are represented in Figures 7 and 8, respectively. The UML model exhibits dynamic polymorphism, whereas the UML model Payroll system exhibits static polymorphism. The UML model Deposit has three inconsistent method invocations. The method invocations are prefixed with the guard condition. The UML model Payroll System has 9 method invocations out of which 5 invocations are inconsistent. The UML model Deposit in Figure 7 forms a part of the banking system to compute the interest of term deposits. The method Interest() is overridden in the derived classes. The interest rate depends on the period of the term deposit. The three method invocations are inconsistent. Inconsistent design results in wrong values for the interest calculated and maturity value. This creates a set of unsatisfied customers and affects the credibility of the banking system. Inconsistent design results in the creation of software with faults. This affects the

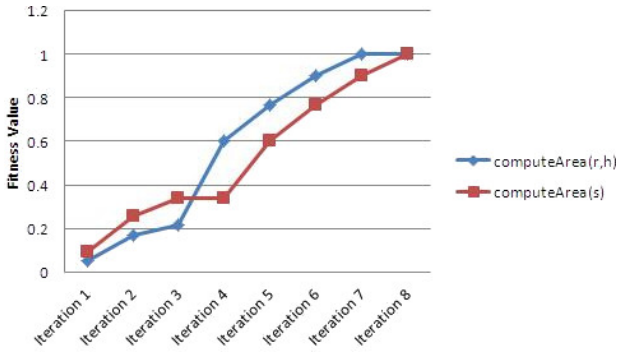


Figure 4. Fitness Values of Inconsistent Methods for UML model 3DObject

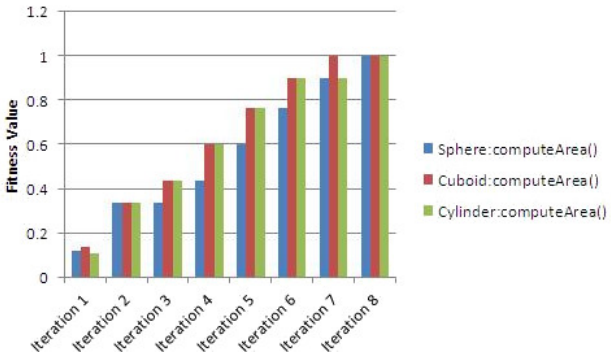


Figure 5. Fitness Values of Inconsistent Methods for UML model ThreeDObject

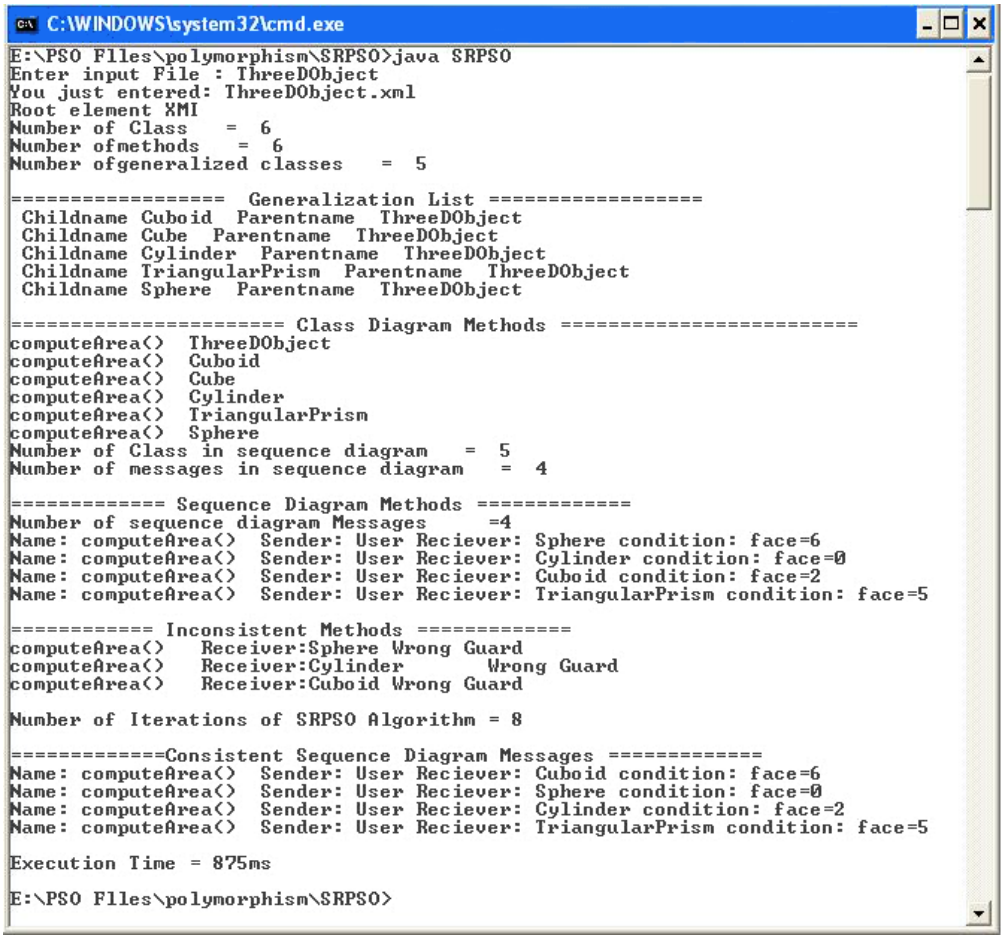


Figure 6. Handling Inconsistencies for the UML model ThreeDObject

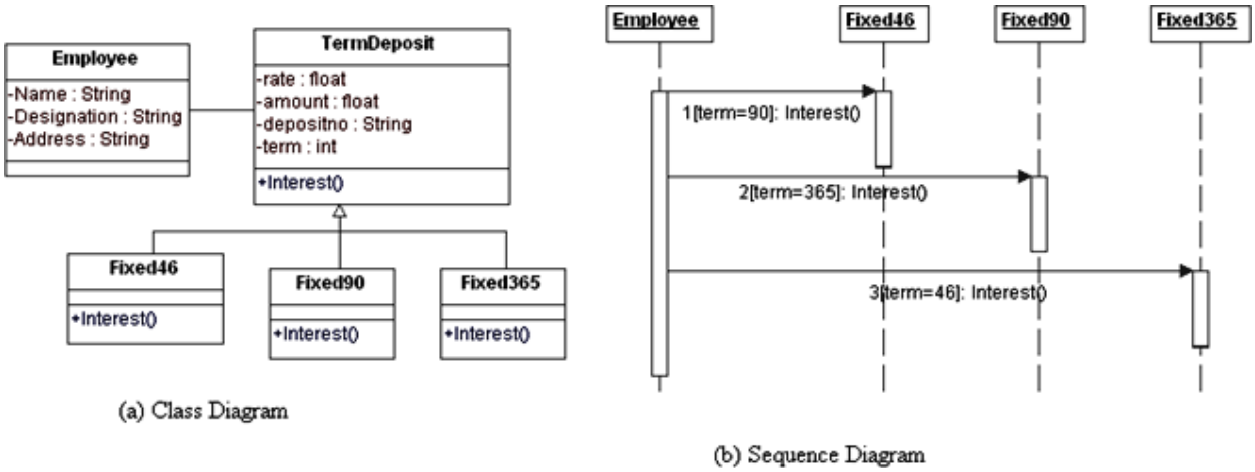
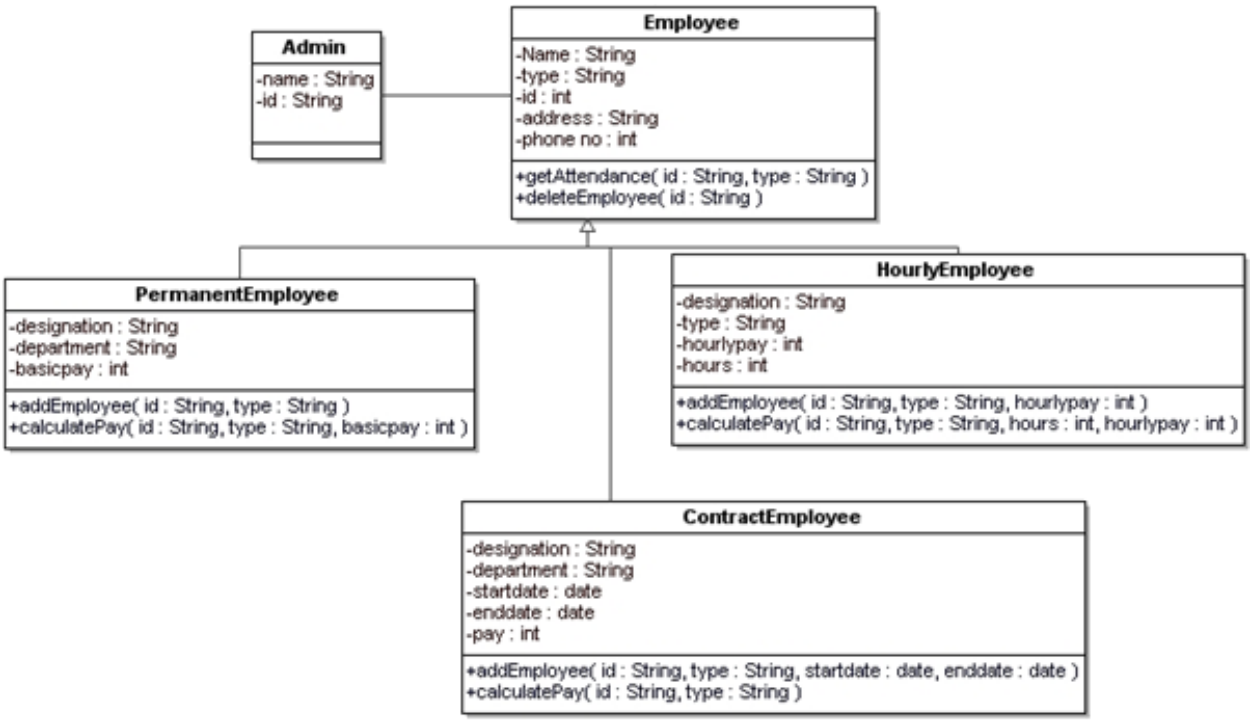


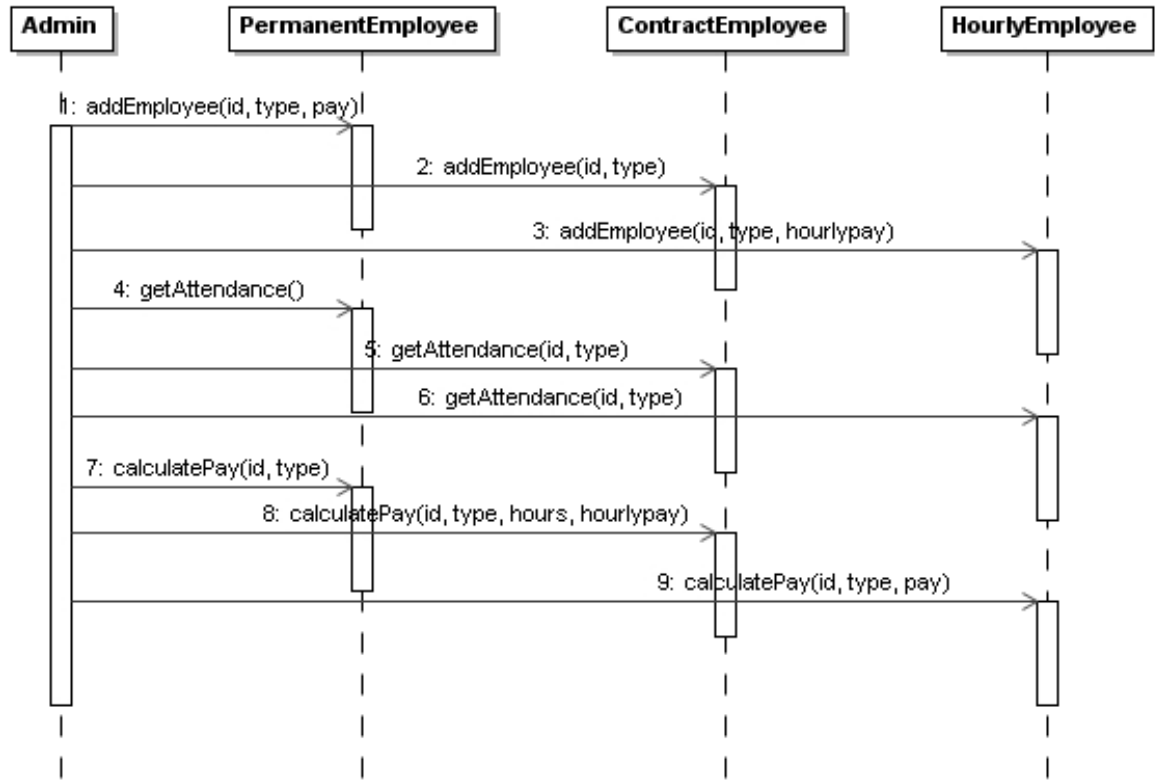
Figure 7. UML Model Deposit

software quality. The errors may be identified either during the testing phase or after delivery of the product, which increases the software devlopment cost, maintenance cost, and time to market the software.

The algorithm is evaluated based on two criteria: convergence and execution time. The convergence of the algorithm is evaluated based on the number of iterations required to resolve inconsistency. Inconsistency is resolved when the



(a) Class Diagram



(b) Sequence Diagram

Figure 8. UML Model Payroll System

Table 3. Execution Time and Convergence

UML Model	Polymorphism Type	Methods	Number of Inconsistent Invocations	Iteration Count	Avg. Running Time(ms)
Deposit	Dynamic	3	3	8	764
3DObject	Static	4	2	8	954
ThreeDObject	Dynamic	4	3	8	984
Course Registration System	Dynamic	12	6	7	850
Payroll System	Static	9	5	7	998
Demonstrative Sample	Static	12	7	8	1052

fitness values of all particles in the swarm are equal to one. We have modelled different case studies and the algorithm is experimented on different inconsistent models exhibiting static and dynamic polymorphism. Table 3 represents the execution time and convergence of the algorithm on different UML models. The table represents the UML models, the type and number of inconsistencies present in the models, the number of iterations required to converge, and the average running time of the algorithm. The execution time of the algorithm is computed on an Intel Core i7 CPU running at 2.80 GHz with 4 GB primary memory. The UML model Deposit requires an average running time of 764 ms to achieve consistency; the average running time of UML model 3DObject and ThreeDObject are 954 ms and 984 ms, respectively. The UML model 3DObject exhibits static polymorphism and has 4 method invocations out of which two invocations are inconsistent. Models Deposit and ThreeDObject exhibit dynamic polymorphism. The UML model Demonstrative Sample has polymorphic and non-polymorphic methods. Inconsistencies in non-polymorphic methods are detected from the fitness value computation. The results show that the average time taken by the algorithm to detect and fix inconsistencies in polymorphic methods is of the order of milliseconds and the algorithm converges in all cases.

Table 4 represents the statistical evaluation results of the algorithm. The values of mean, standard deviation and variance are computed for different values of acceleration coefficients. The algorithm is statistically evaluated on the UML model and better results are obtained with acceleration coefficient values equal to 1.49445. The evaluation results have shown that the algorithm detects and fixes all inconsistent method invocations. As a result, no false positives or false negatives are detected. Hence the precision and recall values are high and equal to the one in our approach.

Inconsistency handling has a prime role in the development of quality software. Polymorphism makes the design extensible. It simplifies the design and enables the addition of new functions without creating additional overheads. Inconsistencies arising due to method invocation inconsistency of polymorphic methods are hard to detect. We have presented an AI based approach that detects and fixes inconsistency in polymorphic and non-polymorphic methods. Our approach provides significant role in ensuring software design consistency. The proposed approach of inconsistency detection has a number of advantages. The method operates on a specification of the diagram and uses a direct approach of detecting and fixing inconsistencies without transforming the model to an intermediate representation. The approach detects and fixes method invocation inconsistency in polymorphic and non-polymorphic methods. The fitness function uses simple calculations. Addition of new rules requires only a redefinition of the fitness function. Inconsistencies are fixed by identifying the receiver class from the cohesion values and guard conditions and redefining the method invocations in the sequence diagram. The algorithm is fast and computationally inexpensive. As the inconsistencies are detected and fixed in the design phase, the errors are not propagated to the code generation phase. Hence, the development and maintenance costs are reduced and quality of the code can be improved.

Table 4. Statistical Evaluation of the Algorithm

UML Model	Parameter	$a1 = a2 = 1.49445$		$a1 = a2 = 1$	
		4 Runs	7 Runs	4 Runs	9 Runs
3DObject	Mean	0.4685	0.95	0.2768	1
	SD	0	–5.551E–17	0	0
	Variance	0.017292	0.0025	0.003624	0
ThreeDObject	Mean	0.544444	0.9333333	0.488889	1
	SD	–3.70E–17	–1.110E–16	0	0
	Variance	0.00617	0.0022222	0.000202	0
Deposit	Mean	0.65556	0.95556	0.4888809	1
	SD	0	3.701E–17	–1.850E–17	0
	Variance	0.006173	0.003951	0.00617284	0
Course Registration System	Mean	0.711111	1	0.416667	0.911111
	SD	0	0	9.2519E–18	–3.701E–17
	Variance	0.006173	0	0.001389	0.003951
Payroll System	Mean	0.7	1	0.413333	1
	SD	0	0	–1.110E–17	0
	Variance	0.006667	0	0.0016	0

7. Threats to validity

The section deals with threats to validity.

External Validity concerns with how the result of the experiments can be generalized to other environments. As part of the evaluation, we have evaluated the algorithms on UML models involving polymorphic method invocations. The proposed approach detects and fixes inconsistencies involving static and dynamic polymorphic method invocations. The algorithm can be generalized to detect inconsistencies in non-polymorphic method invocations and handle other inconsistencies involving sequence diagrams. The generalization can be performed by modifying the fitness function. This argument is substantiated by describing how another inconsistency related to the class and sequence diagram is handled. The consistency rule states that two objects in the sequence diagram interact only if there is an association in the class diagram between the interacting objects. The fitness function can be modified to include another term comprising of the sender and receiver classes in the class and sequence diagram. The proposed approach models inconsistency handling as an optimization problem and detecting inconsistencies with fitness function. The algorithm can be expanded to detect and fix intra-model inconsistencies among different diagrams. We have defined the fitness function in terms of the properties of the inconsistent model elements. Inconsistency detection among different diagrams requires definition of the fitness function in terms of the properties of the inconsistent model element and a particle representation has to be formulated for the inconsistent model element in terms of its properties.

Construct Validity refers to the extent to which the experiment setting reflects the theory. We are able to successfully implement the algorithm on a set of UML models involving static and dynamic polymorphic method invocations. The fitness functions are defined with the aim of detecting method invocation inconsistencies and inconsistencies are identified and resolved accurately. The UML models are a representative of the models on which a consistency check can be performed. The number of inconsistencies in the UML models varies from 3 to 7 and the number of method invocations varies from 3 to 12.

Internal Validity represents the extent to which the casual relationship established cannot be explained by other factors. The casual relationships between class diagram method signature and sequence diagram method signature are analyzed to detect inconsistency. Method

invocation inconsistency arises due to the invocation of a method on an object of a class in which the method is not defined. Fitness function is defined in terms of the method signature and class names. Hence, the method signature is the major component in inconsistency detection and the casual relationship between method signatures is exploited to detect inconsistencies. In the case of dynamic polymorphism, since the method signatures of the polymorphic methods are the same, a further comparison of guard conditions and constraints is performed.

Conclusion Validity: We have performed a statistical evaluation of the algorithm and the results are summarized in Table 4. The models used for evaluation are a representative of the UML models used in the design of software systems. The statistical evaluation results show that the algorithm converges in less number of iterations with acceleration coefficient values equal to 1.49445. The convergence of the algorithm and execution time are also computed. The average execution time is of the order of milliseconds and the number of iterations required for the algorithm to converge is independent of the number of method invocations or the number of inconsistencies.

8. Conclusion

With the increasing relevance of software in industries and manufacturing, the complexity and size of the software and the complexity of the design has increased. Developing quality software is one of the major challenges faced by software developers. One of the definitions of quality software is fitness for purpose and quality software should be able to function as per the user′s requirements. One of the key aspects to ensuring software quality is good design. Inconsistent design leads to the generation of software with faults. A periodic review of the software design is one the factors that can enhance the software quality and reduce software failures thereby improving manufacturing and productivity. The review helps to detect inconsistencies and fix the inconsistencies. Polymorphism is an important feature that makes the software design compact and extensible. It is hard to trace the polymorphism as it is often detected at run time. We introduce an intelligent automated approach that uses the SRPSO algorithm to detect and fix inconsistency in polymorphic methods. The algorithm is evaluated on different case study involving static and dynamic polymorphism. The method detects and fixes inconsistencies in all cases. Analysis of the results shows that the inconsistency detection and fixing in our approach is quick, easy, and effective. The proposed approach has a number of advantages. The algorithm can be invoked after the application is modelled or during and after refinements to the models. The method operates directly on the diagram specification and does not require transformation to another representation. Addition of new rules requires only a redefinition of the fitness function. The fitness function uses simple calculations. The time required to detect and fix inconsistencies is of the order of milliseconds. The inconsistencies developed in the design are detected and corrected in the same phase. Maintenance cost of software is a huge burden for manufacturing industries. Automatic detection of inconsistencies in polymorphic methods during the design phase ensures quality of the code produced and reduces development and maintenance cost of the software.

References

[1] M. Harman, “The role of artificial intelligence in software engineering,” in *First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*. IEEE, 2012, pp. 1–6.

[2] M. Harman and B.F. Jones, “Search-based software engineering,” *Information and Software Technology*, Vol. 43, No. 14, 2001, pp. 833–839.

[3] O. Raiha, “A survey on search-based software design,” *Computer Science Review*, Vol. 4, No. 4, 2010, pp. 203–249.

[4] J. Kennedy and R. Eberhart, “Particle swarm optimization,” in *Proceedings of IEEE International Conference on Neural Networks*. IEEE, 1995, pp. 1942–1948.

[5] M.R. Tanweer, S. Suresh, and N. Sundararajan, “Self regulating particle swarm optimization al-

gorithm," *Information Sciences*, Vol. 294, 2015, pp. 182–202.

[6] P. Stevens and R.J. Pooley, *Using UML: Software engineering with objects and components*. Pearson Educationr, 2006.

[7] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *Future of Software Engineering*. IEEE Computer Society, 2007, pp. 37–54.

[8] B. Selic, "The pragmatics of model-driven development," *IEEE Software*, Vol. 20, No. 5, 2003, pp. 19–25.

[9] C. Pons, L. Olsina, and M. Prieto, "A formal mechanism for assessing polymorphism in object-oriented systems," in *Proceedings of First Asia-Pacific Conference on Quality Software*. IEEE, 2000, pp. 53–62.

[10] A. Egyed, "Fixing inconsistencies in UML design models," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 292–301.

[11] A. Egyed, E. Letier, and A. Finkelstein, "Generating and evaluating choices for fixing inconsistencies in UML design models," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2008, pp. 99–108.

[12] A. Egyed, "Automatically detecting and tracking inconsistencies in software design models," *IEEE Transactions on Software Engineering*, Vol. 37, No. 2, 2011, pp. 188–204.

[13] Q. Long, Z. Liu, X. Li, and H. Jifeng, "Consistent code generation from UML models," in *Proceedings of the Australian Software Engineering Conference*. IEEE, 2005, pp. 23–30.

[14] L.C. Briand, Y. Labiche, and L. O'Sullivan, "Impact analysis and change management of UML models," in *Proceedings of the International Conference on Software Maintenance, ICSM 2003*. IEEE, 2003, pp. 256–265.

[15] R. Kretschmer, D.E. Khelladi, A. Demuth, R.E. Lopez-Herrejon, and A. Egyed, "From abstract to concrete repairs of model inconsistencies: An automated approach," in *24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2017, pp. 456–465.

[16] A. Rountev, A. Milanova, and B.G. Ryder, "Fragment class analysis for testing of polymorphism in Java software," *IEEE Transactions on Software Engineering*, Vol. 30, No. 6, 2004, pp. 372–387.

[17] D.K. Saini, "Testing polymorphism in object oriented systems for improving software quality," *ACM SIGSOFT Software Engineering Notes*, Vol. 34, No. 4, 2009, pp. 1–5.

[18] R. Kaur and J. Sengupta, "Software process models and analysis on failure of software development projects," *arXiv preprint arXiv:1306.1068*, 2013.

[19] K.A. Briski, P. Chitale, V. Hamilton, A. Pratt, B. Starr, J. Veroulis, and B. Villard, "Minimizing code defects to improve software quality and lower development costs," *Development Solutions White Paper, IBM*, 2008.

[20] P. Jalote, "An integrated approach to software engineering," *Springer Science and Business Media*, 2012.

[21] B. Nuseibeh, S. Easterbrook, and A. Russo, "Making inconsistency respectable in software development," *Journal of Systems and Software*, Vol. 58, No. 2, 2001, pp. 171–180.

[22] A. Reder and A. Egyed, "Incremental consistency checking for complex design rules and larger model changes," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, Berlin, Heidelberg, 2012, pp. 202–218.

[23] M.R. Ehrenleitner, A. Demut, and A. Egyed, "Towards model-and-code consistency checking," in *38th Annual Computer Software and Applications Conference*. IEEE, 2014, pp. 85–90.

[24] N. Macedo, T. Jorge, and A. Cunha, "A feature-based classification of model repair approaches," *IEEE Transactions on Software Engineering*, Vol. 43, No. 7, 2017, pp. 615–640.

[25] J.P. Puissant, R.V.D. Straeten, and T. Mens, "A regression planner to resolve design model inconsistencies," in *European Conference on Modelling Foundations and Applications*. Springer, Berlin, Heidelberg, 2012, pp. 146–161.

[26] F.J. Lucas, F. Molina, and A. Toval, "A systematic review of UML model consistency management," *Information and Software technology*, Vol. 51, No. 12, 2009, pp. 1631–1645.

[27] D. Kalibatiene, O. Vasilecas, and R. Dubauskaite, "Rule based approach for ensuring consistency in different UML models," in *EuroSymposium on Systems Analysis and Design*. Springer, Berlin, Heidelberg, 2013, pp. 1–16.

[28] C.F. Borbaand and A.E.A. Da Silva, "Knowledge-based system for the maintenance registration and consistency among UML diagrams," in *Brazilian Symposium on Artificial Intelligence*. Springer, Berlin, Heidelberg, 2010, pp. 51–61.

[29] D. Torre, Y. Labiche, and M. Genero, "ML consistency rules: A systematic mapping study," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. Springer, Berlin, Heidelberg, 2014, pp. 1–10.

[30] R.J. Ma, N.Y. Yu, and J.Y. Hu, "Application of particle swarm optimization algorithm in the heating system planning problem," *The Scientific World Journal*, 2013, pp. 1–11.

[31] S. Pandey, L. Wu, S.M. Guru, and R. Buyya, "A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments," in *24th IEEE International Conference on Advanced Information Networking and Applications*. IEEE, 2010, pp. 400–407.

[32] D.Y. Sha and H.H. Lin, "A multi-objective PSO for job-shop scheduling problems," *Expert Systems with Applications*, Vol. 37, No. 2, 2010, pp. 1065–1070.

[33] M. Gong, Q. Cai, X. Chen, and L. Ma, "Complex network clustering by multiobjective discrete particle swarm optimization based on decomposition," *IEEE Transactions on Evolutionary Computation*, Vol. 18, No. 1, 2014, pp. 82–97.

[34] N.K. Sharma, D.S. Babu, and S.C. Choube, "Application of particle swarm optimization technique for reactive power optimization," in *IEEE-International Conference on Advances in Engineering, Science and Management (ICAESM-2012)*. IEEE, 2012, pp. 88–93.

[35] P. Sivakumar, S.S. Grace, and R.A. Azeezur, "Investigations on the impacts of uncertain wind power dispersion on power system stability and enhancement through PSO technique," in *International Conference on Energy Efficient Technologies for Sustainability*. IEEE, 2013, pp. 1370–1375.

[36] F. Li, D. Li, C. Wang, and Z. Wang, "Network signal processing and intrusion detection by a hybrid model of LSSVM and PSO," in *15th IEEE International Conference on Communication Technology*. IEEE, 2013, pp. 11–14.

[37] Z. Jun and Z. Kanyu, "A particle swarm optimization approach for optimal design of PID controller for temperature control in HVAC," in *Third International Conference on Measuring Technology and Mechatronics Automation*. IEEE, 2011, pp. 230–233.

[38] D.K. Saini and Y. Sharma, "Soft computing particle swarm optimization based approach for class responsibility assignment problem," *International Journal of Computer Applications*, Vol. 40, No. 12, 2012, pp. 19–24.

[39] R. George and P. samuel, "Fixing class design inconsistencies using self regulating particle swarm optimization," *Information and Software Technology*, Vol. 99, 2018, pp. 81–92.

[40] R. George and P. samuel, "Particle swarm optimization method based consistency checking in UML class and activity diagrams," in *Innovations in Bio-Inspired Computing and Applications*. Springer, Cham, 2016, pp. 117–127.

[41] R. George and P. Samuel, "Fixing state change inconsistency with self regulating particle swarm optimization," *Soft Computing*, Vol. 24, No. 24, 2020, pp. 18 937–18 952.

[42] M. Kessentini, H. Sahraoui, and M. Boukadoua, "Search-based model transformation by example," *Software and Systems Modeling*, Vol. 11, No. 2, 2012, pp. 209–226.

[43] A. Windisch, S. Wappler, and J. Wegener, "Applying particle swarm optimization to software testing," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM, 2007, pp. 1121–1128.

[44] R. Malhotra and A. Negi, "Reliability modeling using particle swarm optimization," *International Journal of System Assurance Engineering and Management*, Vol. 4, No. 3, 2013, pp. 275–283.

[45] J.A. Parejo, A. Ruiz-Cortes, S. Lozano, and P. Fernandezi, "Metaheuristic optimization frameworks: A survey and benchmarking," *Soft Computing*, Vol. 16, No. 3, 2012, pp. 527–561.

[46] S. Milton and H. Schmidt, "Dynamic dispatch in object-oriented languages," The Australian National University, Canberra, Technical Report TR-CS-94-02, January 1994.

[47] E. Ernst and D.H. Lorenz, "Aspects and polymorphism in AspectJ," in *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*. ACM, 2003, pp. 150–157.

[48] D. Ancona, S. Drossopoulou, and E. Zucc, "Overloading and inheritance," in *FOOL' 01 – International. Workshop on Foundations of Object Oriented Languages*, 2001.

[49] D.P. Friedman, M. Wand, and C.T. Haynes, *Essentials of Programming Languages*, 2nd ed., Prentice-Hall of India, 2001.

[50] R.V. Binder, "Testing object oriented software: A survey," *Software Testing, Verification and Reliability*, Vol. 6, No. 3–4, 1996, pp. 125–252.

[51] D.P. Rini, S.M. Shamsuddin, and S.S. Yuhaniz, "Particle swarm optimization: technique, system and challenges," *International journal of computer applications*, Vol. 14, No. 1, 2011, pp. 19–26.

[52] Q. Bai, "Analysis of particle swarm optimization algorithm," *Computer and information science*, Vol. 3, No. 1, 2010, pp. 180–184.

[53] M. Ferreira, F. Chicano, E. Alba, and J.A. Gomez-Pulido, "Detecting protocol errors using particle swarm optimization with Java pathfinder," in *Proceedings of the High Performance Computing and Simulation Conference (HPCS 08)*, 2008, pp. 319–325.

[54] D. Floreano and C. Mattiuss, "Bio-inspired artificial intelligence: theories, methods, and technologies," MIT press, Aug 2008.