Nature-inspired techniques for conformance testing of object-oriented software

A. Bouchachia*, R. Mittermeir, P. Sielecky, S. Stafiej, M. Zieminski

Group of Software Engineering and Soft Computing, Department of Informatics-Systems, University of Klagenfurt, 9020 Klagenfurt, Austria

ABSTRACT

Soft computing offers a plethora of techniques for dealing with hard optimization problems. In particular, nature based techniques have been shown to be very efficient in optimization applications. The present paper investigates the suitability of various nature-inspired meta-heuristics (genetic algorithms, evolutionary programming and ant-colony systems) to the problem of software testing. The present study is part of the nature-inspired techniques for object-oriented testing (NITOT) environment. It aims at addressing the problem of conformance testing of object-oriented software to its specification expressed in terms of finite state machines. Detailed description, adaptation and evaluation of the various nature-inspired meta-heuristics are discussed showing their potential in this context of conformance testing.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Nature-inspired search meta-heuristics have been the focus of many research studies in the context of software engineering and in particular in software testing [41]. The typical application of these techniques is the task of automatic generation of test data. Because such a task can be formulated as an optimization problem (that is, the test goal is transformed into an objective function or a combination of objective functions), the suitability of search techniques for software testing is an indisputable issue.

For procedural software, Ferguson and Korel [21], Marré and Bertolino [40], and Hierons [39] among others addressed the problem of automatically generating efficient and effective test suites. For object-oriented problems, the issue appears in a slightly different context. As noted in Binder [7], object-oriented software poses a host of new problems to testers. The state-encapsulation problem sticks out particularly among them. At the method level, the methodology of testing procedural programs still applies. The key difference is though, that object-oriented testing assumes that the code under test is state-based due to encapsulation, while procedural testing assumes that the code can be strictly functional, i.e., it is state-free. At the level of class (unit), the notion of state becomes even central, since each instance is identified by its state which can be changed only by a sequence of method (modifier or modifier/inspector) calls. This holds also for the cluster level and system level. Hence, as expressed by Jorgensen and Erickson [31] testing object-oriented software has, from the class level onwards, characteristics of integration testing. The challenges of developing an efficient and effective test suite become even serious.

The present research study focuses on conformance testing of object-oriented software. This assumes that a sufficiently formal specification exists to devise testing data in the form of input, output pairs necessary to execute the corresponding implementation. That is, the specification serves as a basis for an oracle and testing aims at assessing whether the actual behavior of the code conforms to the one expected by the oracle. In this context, a test case is defined as a sequence of method calls that brings the object to a certain state. Departing from a testing goal derived from the specification at hand, the testers need to be supported in their search for test sequences that satisfy that testing goal. For instance, the testing goal could be setting the object into a target state, hence testers need support to obtain test data that achieves the desired state. The testing environment NITOT we are currently constructing aims at providing this support. We will show through various experiments and using different strategies and heuristics how test cases can be generated meeting the test goal.

As various nature-inspired meta-heuristics have already been successfully used for identifying test data for procedural programs (see Section 3), it appeared reasonable to explore their suitability in the context of conformance testing of object-oriented programming. Three heuristics are studied for this purpose: genetic algorithms, evolutionary programming, and ant-colony systems. While the first two heuristics appear to be intuitively the first candidates disregarding their technical formulation and adaptation, ant-colony optimization is appealing since state
coverage can be conceived as a graph traversal problem. They have been shown to be efficient in dealing with various optimization problems.

Due to these considerations, the paper highlights the contributions of the current investigation in Section 2. Then, it describes the testing framework under development in Section 4. In Section 5, it focuses on the three strategies evaluated. The adaptations required for each of these techniques is described. After a brief introduction to the whole testing framework (Sect. 6), section 7 reports the results from the experiments conducted.

2. Contributions

While in the existing literature pertaining to the object-oriented software testing [12,13,16,24], conventional finite state machines (FSMs) have been used to model the behavior of the individual classes and even the behavior of class clusters and system, in this paper we introduce a new type of FSMs, called class finite state machines (CFSM). It is an extension of the classical FSM. It allows explicitly to deal with parameterized messages (i.e., object-oriented methods involving parameters of different types) and guarded transitions (messages are executed only if the predicates on the transitions are fulfilled).

From the operational perspective, these CFSMs themselves are refined into fine grained state machines (GFSMs) which serves as basis for effective generation of test data.

The novelty of the approach proposed is also enhanced by further aspects:

- Possibility to handle parameterized methods via augmented transitions of the CFSM.
- Application of search techniques to handle the problem of state explosion.
- Proposal of adaptation mechanisms for these search techniques to fit the context of conformance testing.
- Proposal of an integrated testing framework that is generic and can integrate any other nature-inspired search technique.

The empirical evaluation shows how these aspects are assessed by the present paper.

3. Related work

Due to their key relevance to this study, the application of finite state machines and nature-inspired meta-heuristics to the problem of software testing will be highlighted in the next subsections.

3.1. FSM as basis for testing

In the context of object-oriented programming, FSMs are used to represent the behavior of the individual classes and combination of FSMs to represent class clusters and system. Therefore, except the algorithmic level (method level), all other testing levels (i.e., class level, cluster level, and system level) can rely on the FSM formalism [12,13,16,24].

The W-Method [14] can be considered as the precursor of state-based testing methods. It was later renamed as round-trip path testing [7]. Round-trip testing guarantees covering each transition at least once. It relies on a transition tree that is obtained from the state machine. It is then traversed using traversal algorithms such as depth-first or breadth-first traversal. This tree shows all the round-trip paths which correspond to transition sequences starting and ending with the same state. Each path produces a test case. The round-trip method is quite popular in the state-based testing community [1,12]. This technique can also be applied to flattened statecharts as is originally done in [7] on unified modeling language (UML) diagrams.

There have been several attempts to apply state charts and state machines especially in the context of communication protocols. In [23], a method called "partial W-Method" an extension of the W-Method [14] has been introduced which aims at reducing the size of the testing set at the price of a complex test generation procedure. A similar method, called harmonized state identification method, targeting at the same goal has been proposed in [38]. There exist also other methods, like Unique Input-Output (UIO) or Unique Input-Output Circuit (UIOC), which are reviewed in [45,26,27,20]. Some of these methods, such as UIO, do not guarantee complete fault detection by testing [9].

In [46], a detailed study of four test sequence generation methods is presented. These include Transition Tours (T-Method), Unique Input/Output Sequence (U-Method), Distinguishing Sequence (D-Method), and Characterizing Set (W-Method -- that is already mentioned). The study has shown that the U-, D-, and W-Methods outperform the T-Method. However, it appears that the W- and U-methods seem to be established as state-based testing methodologies. The W-Method (round trip) has been applied in many recent papers [1,12,24,2] due to its acceptable cost and coverage criteria. On the other hand, the U-Method [45,26,27,20,9] has been widely used because of the fact that not all FSMs have a distinguishing sequence (DS) and most FSMs have UIOs for each state. Moreover, the length of a UIO sequence is no longer than DS.

Recently a heuristic approach based on UIO has been proposed in [27] to diagnose faults when testing from finite state machines. Very close to the idea we investigate in this paper, Guo et al. compare the implementation under test (IUT) against a finite state machine (for which the complete transition diagram is available) describing the specifications of the program. A IUT is faulty if the output sequences resulting from both the specification and the IUT are different. This can be caused by either an incorrect output (output fault) or an incorrect state transfer (state transfer fault). The approach proposed by the authors provides mechanisms to identify the set of transitions that are responsible for the faulty behavior of the IUT. To realize that, a set of reduction techniques are used to reduce the set of suspect transitions.

Hartmann, Imoberdorf and Meisinger suggest in [28] the application of UML statecharts to perform conformance testing of distributed component based systems. In order to generate test cases automatically, developers first define the dynamic behavior of the components via UML Statecharts, specify the interactions among them and finally annotate them with test requirements. Test cases are then derived from these annotated statecharts. A very similar approach using extended finite state machines has been investigated in [24].

Similar to control and data flow coverage criteria, state-based testing relies on coverage criteria defined in [43]. These include: all-transitions, all-transition-pairs, all-predicates and all-sequences. Other coverage criteria like all-states and all-n-transitions are very often used. However, all-states (each state must reached at least once) is subsumed by all-transitions criterion which is in turn subsumed by all-n-transitions. While these criteria are typical for state-based testing, in many research publications [6,30,24,34] state machines have been extended to deal with data-flow coverage criteria [50].

3.2. Heuristics for test case generation

It is, however, important to notice that conformance testing is still an active research area due to the complexity of systems being developed in contemporary applications [5]. This complexity is due to the existence of large number of states and transitions when such systems are modeled by state machines. This applies in
particular to control and reactive systems. Moreover, little work has been done in the context of system (or integration) and cluster testing using FSMs. Again at these testing levels, a large number of states and transitions are required to exhibit the interactions between the system components and overall behavior of the system. Due to these problems, testing such FSMs may become unmanageable and subject to the state explosion problem.

To deal with such problems, there have been recently some attempts to apply search techniques. Among other population-based search techniques that have been used for structural test data generation [11], evolutionary algorithms and ant-colony systems have been used to conduct state-based testing. Because such techniques are not popular in the context of state-based testing, there have been few studies showing their relevance for testing.

In the following we review these studies from the perspective of state-based (be it procedural, object-oriented or conformance) testing for the sake of completeness. Note that structural testing aims at finding test suites relying on the structure of the code, hence the name of white-box testing. Moreover, state-based testing can be understood from two perspectives: (1) as traditional structural testing when the searched test cases aim at achieving a statement depending on a variable whose value (called state) is set only by executing other parts of the code (e.g., method call) and (2) as testing object-based code in which objects change their state after executing messages. Test cases in the first scheme represent tuples of input/output, while in the second scheme a test cases represent sequences of message calls that lead to a state of interest.

In [42], evolutionary algorithms have been applied to search for a sequence of inputs that allow to reach a test target (statement). Such an approach does not deal explicitly with state machines, but rather with procedural programming. It handles the problem of reachability. The authors used the chaining approach [21] to identify a sequence of statements, involving internal object variables, that need to be executed prior to the test target. This may require the execution of functions. In a similar approach, Barasel et al. [4] suggested a particular encoding that allows to take function call into account.

Tonella suggests in [47] an evolutionary approach that aims at generating input sequences for the structural testing of classes. The encoding of chromosomes, which stand for test cases in the form of method sequences, incorporates information on which objects to create, which methods to invoke and which values to use as inputs (i.e., methods’ parameters). To preserve consistency when applying genetic operators, Tonella proposed a grammar that defines the syntax of chromosomes. The mutation operation relies on different operators: mutation of input value, constructor change, insertion of method invocation and removal of method invocation. A one point-crossover has been used for the combination of chromosome. The proposed algorithm is applied with the aim of maximizing the branch coverage measure.

In a similar work, but also restricted to structural testing, Wappler and Lammermann [48] propose an evolutionary approach where each chromosome represents a sequence of statements. Each statement consists of a target object, a method and its parameters. An algorithm to decode and encode the test program is proposed. Since chromosomes are of fixed length, the maximal number of statements must be predefined. An interesting idea proposed in this approach is a multi-level optimization i.e., each part of the statement (a target object, a method and parameters) can be optimized on a different level.

Departing from the fact that the computation of Unique Input/Output sequences (UIOs) is NP-hard [36], the authors in [25] proposed genetic algorithms (GAs) to deal with the problem of generating UIOs. An initial population is produced by randomly generating input sequences. Based on the state splitting tree proposed in [36], a fitness function encourages candidates to split the set of all states into more discrete units and punishes the length of the sequences is proposed. Roulette wheel selection and uniform crossover are used as genetic operators. Experimental results suggest that, in a small system, the performance of the GA based approaches is no worse than that of random search while, in a more complex system, the GA based approaches outperform random search.

Ant-colony optimization (ACO) has also been used in previous research. For instance, in [37], Li et al. suggested an ACO approach to generate test suites from the UML Statechart diagrams for state-based testing. During test case generation, only feasible and optimal (shortest) ones are retained. This approach does not deal with object oriented programming features and does not assume that transitions are equipped with methods having parameters. In [41], the authors reported on the application of ACO to generate sequences of method calls in the context of structural testing. This work does not deal explicitly with state machines, but rather with procedural programming that handles the problem of reachability.

An ACO-based approach relying on the Markov Usage Model (MUM) was proposed in [32] to derive test cases (test paths) for a software system. In this approach, the possible uses of the software system are modeled by a MUM that reflects the operational distribution of the software system and is enriched by estimates of failure probabilities, losses in case of failure and testing costs. In the proposed MUM approach the arcs represent the processing steps of the program (functions) and the nodes represent the state of the program after executing the function(s) on the incident arc. In order to generate test paths with minimum loss (due to failure) and the cost (not exceeding a given budget), the authors resort to an NP-hard combinatorial problem. They applied ACOs where the transition probabilities are computed using a combination of the estimates of failure probability, loss probability, and cost probability.

4. A framework for object-oriented conformance testing

State-bearing software, notably object-oriented software is quite often modeled by means of finite state machines (FSM) such as UML state diagrams. A finite state machine is defined as $A = (Q, \Sigma, \delta, q_0, F)$ with:

- $Q$ is a finite set of states,
- $\Sigma$ is a finite alphabet of input symbols,
- $q_0 \in Q$ is an initial start state,
- $F \subseteq Q$ is a set of final states and
- $\delta: Q \times \Sigma \rightarrow A$ is a transition function.

A language $L \in \Sigma^*$ is called a regular language if there exists a finite state machine, $A$, such that $L = L(A)$ (see Fig. 1). Moreover, a regular language can be intentionally expressed in terms of a regular grammar or extensionally in terms of the set of terms/ phrases that belong to it. The three formulations: FSM, grammar and set of sentences are equivalent representations of the same regular language. This correspondence is relevant for the test data
generation problem since we can consider the specification of the class under test (CUT) as an FSM that accepts exactly the "language" made up of all messages this class can deal with. We refer to this language as the CUT's input language \( Li(CUT) \). In case one is interested also on CUT's output language, \( Lo(CUT) \), one would have to extend the FSM to a Mealy machine. As messages consist of method-id's followed by a list of parameters of the methods, \( Li(CUT) \) has to represent what will be referred to later as message call blocks (MCBs), with each MCB consisting of the method-id and possibly list of parameter values.

Departing from the assumption that each class under test is specified by means of a finite state machine, conformance testing, as shown in Fig. 2, consists of comparing the actual behavior of the code against the expected behavior of the class FSM. This is done in a systematic way by:

1. defining a testing goal (for instance, a certain state),
2. generating test cases in the form of method call sequences based on the FSM and then,
3. exercising the test cases on the code of CUT and
4. comparing the output of the execution against the testing goal.

Because at the level of implementation, states are defined by the cross-product of specification states with implementation states defined by class variables, we refer to the corresponding FSM as a fine granular finite state machine (GFSM). Transitions in the GFSM are the transitions specified in the specification augmented with the actual parameter value permitted at the implementation level. Formally, this GFSM is expressed as a finite state machine \( FM = (Q, \Sigma, q_0, \delta, F) \) with

- \( Q = K \times V = \{(k, v) | k \in \text{SpecificationStates} \land v \in \text{range(associatedImplementationVariable)}\} \)

The set of states \( Q \) is the cross-product of the names of the states an object might assume according to its specification, \( V \), with the fine grained states this object might assume in the specification, \( T \).

In case the implementation uses a user-defined scalar type that directly corresponds to the states of the specification, only the one-to-one mappings of specification state label to implementation state label will lead to reachable GFSM-states while all other mappings will lead to isolated states. In case the implementation uses a more fine granular program variable (as in the bank account example used in Sect. 7), this cross-product contains all correspondences between specification states and implementation states. Again, only combinations legitimate according to the requirements of the system will be reachable by transitions. The remaining items of the cross-product will be isolated states in the GFSM.

The alphabet \( \Sigma \) of the "language" accepted by this automaton consists of the messages the class under test can accept. This set of messages is defined by the identifiers of methods the class consists of, \( T \). Since methods might have some parameters (or none), the specification foresees the combination of message and parameter values. As this list is possibly empty, the union with the singleton set containing the empty word \( \{\} \), as elements of the alphabet is needed.

It is to be mentioned already here that in the experiments we restricted the parameter types to match the type of the implementation variable and that the parameter type was constrained to be drawn from either integer, character, or boolean.

- \( q_0 \) with \( q_0 \in Q \) is the initial state of the GFSM.
- \( \delta \) with \( \delta \subseteq Q \times \Sigma \rightarrow Q \) is the state transition function.

Due to the definition of \( Q \) and \( \Sigma \), we have \( \delta \subseteq (K \times V) \times (T \times P) \rightarrow (K \times V) \). Apparently, only those combinations of states and method invocations figure in \( \delta \), that represent legitimate transitions from (specification \times implementation)-states conforming to the specification to a new, legitimate (specification \times implementation)-state.

- \( F \) with \( F \subseteq (Q \setminus \{q_0\}) \) is the set of final states.

In coverage testing, final states of the automaton don't play a special role. All states but \( q_0 \) have to be considered as targets and hence \( F = (Q \setminus \{q_0\}) \).

The GFSM was defined to obtain a solid formal basis for the state-based testing process. Apparently, it contains many dead states (or states that should be dead according to the specification) and even states situated on a path from \( q_0 \) to an element \( f \in F \) appear in a multiplicity that calls for the definition of equivalence classes. To cope with this multiplicity in the GFSM, a class finite state machine, CFSM, has been defined. This CFSM \( CM = (Q, \Sigma, q_0, \delta, F) \) has the same granularity as the GFSM, but transitions are grouped into equivalence classes according to predicates establishing relationships between the values of the state(s) they are emanating from. Likewise states are grouped into equivalence classes by predicates which usually range over values of the implementation state variables. To facilitate the implementation of the GFSM, it groups states with common label (specification state) but different implementation values into an equivalence class, treating the implementation value like a parameter of the respective specification state.

The crucial difference is with the transition function \( \delta \). \( \delta_{\text{GFSM}} \) corresponds to \( \delta_{\text{CFSM}} \) in so far as the concrete values of the implementation state and of the parameter value are represented in the originating state and in the message by variables. In the receiving state, the implementation state is replaced by an expression using the variable representing the implementation state and the message parameter values. The resulting equivalence class of elementary transitions is guarded by a predicate over the parameter value(s) and the pre-transitional value of the implementation state (the value of the implementation state in the originating state). Thus, \( \delta_{\text{CFSM}} \) has the appearance:

\[
\delta \subseteq (K \times V) \times (T \times P) \rightarrow (K', \text{ funct}(v, p)) \mid \text{ pred}(v, p)
\]

![Fig. 2. Concept of conformance testing.](image-url)
with \( v \) representing the value of the implementation state and \( p \) representing the parameter value attached to the message. The specification-state of the receiving state, \( K' \) might be different from \( K \) or remain the same. The value of the implementation state, \( r \) will be the result of applying function \( \text{funct} \) on the parameters \( v \) and \( p \). The legitimacy of choosing this transition will be ensured by the predicate \( \text{pred}(v, p) \).

Obviously, testing object-oriented software has to cope with potentially very complex state space. This holds, due to interaction between the different granularity of specification variables and implementation variables. Moreover, The CFSM has syntactically a different appearance from the initial GFSM, but it is semantically equivalent. It can be seen as an automaton recognizing (respectively generating) the language of all message sequences generated from the fine-grained FSM. Closer to the implementation, CFSM can is used to perform conformance testing as explained earlier.

5. Nature-inspired techniques for test case generation

The crucial part of conformance testing is the generation of test cases. However, because the computation of particular sequences (e.g., unique input/output sequences), it is NP-hard [36]; hence it is natural to think about applying meta-heuristics to deal with such a combinatorial problem. It has been shown in many studies [11,35,44,49] how the test data generation problem can be conceived as an optimization problem in the context of structural testing.

Likewise, for the sake of conformance testing the goal is to find a minimal set of (shortest) message sequences needed to obtain the testing target relying on some meta-heuristics. In this paper, three different nature-inspired meta-heuristics for generating test sequences are proposed. These are genetic algorithms, evolutionary programming, and ant-colony systems. The following section describes how these heuristics are tuned to the specifics of the test data generation problem based on the concept of CFSM.

5.1. Genetic algorithms

Genetic algorithms (GA) [29] are seen as a powerful method for solving ill-structured problems. For solving problems with GA, one usually finds a genotypical representation for solutions of the problem, randomly generates a population of \( n \) individuals, computes the fitness (adequacy with respect to the optimization criterion) for each of them, selects candidates for mating considering the candidates fitness, and generates offsprings forming a new generation of candidate solutions by crossing over the chromosomes of two members of the respective ancestor generation. The specifics of these operations vary with the different approaches. A particular specific of the genotypical approach is that chromosomes are highly abstract and of fixed length. This allows to conceive the chromosome as a bit string that admits crossover at arbitrary positions. However, there exist also variants of GA, where the basic idea of the algorithm just outlined is preserved, but the representation by genetic material is closer to the problem domain. Consequently, one has to be more selective in choosing spots for applying the crossover operation. These approaches are referred to as phenotypical.

As the issue of state-based conformance testing maps to an optimization problem of finding the smallest set of shortest message sequences that meet a given test goal, fixed length representations of the problem seem not adequate. Testing problems can be posed where any fixed length representation of a solution would be just a little too short. In this case the whole algorithm needs to be invoked again with a larger chromosome length. To avoid such inefficiencies, we decided for a phenotypical representation. This decision led to an easy encoding such that the chromosome consists of fixed length message blocks (or method call blocks, MCB), with each block consisting of the method-id and a parameter space, large enough to admit all parameters of the method admitting the longest parameter list.

Therefore, Crossover can be implemented in such a way that crossover points are either before or right after a method-id position of an MCB. That is either a partial proper message sequence is combined with the remainder of another message sequence or, in case the cut is after the method-id, the new string consists of a substring ending with a method-id that obtained new parameters from another sequence. Growth and shrinking of the representation is obtained by means of mutation. Four mutation operators are proposed, MCB-insertion, MCB-deletion Method-id-substitution, and Parameter-value update.

As both, crossover and mutation, are essentially blind operations, one risks that they lead to strings that do not conform to the language admitted by CFSM. This legitimacy problem can be treated in various ways.

- An obvious alternative would be to delete illegitimate strings right away. This has the advantage that one has only proper message sequences in the population. However, it has the disadvantage that it would constrain the diversity of the population.
- An alternative is to assign a very low (e.g., minimal) fitness value to illegitimate offsprings. With this strategy one has to take care that the population will not be dominated by illegitimate candidates. Appropriate low fitness values will ensure that this is not the case. However, this strategy has the advantage that there remains a low probability that an illegitimate candidate survives and partakes in a future crossover and mutation operations.
- Of course, one could also constrain both, mutation and crossover operations, in such a way that out of two syntactically correct parents only syntactically correct descendants would be produced. This, however, would not only lead to complex operations, but it also deviates too drastically from the principle ideas of GA. Therefore, this option was not pursued.

We decided on a mixed strategy such that illegitimacy due to an illegitimate message selector led to the immediate death of such strings, illegitimacy due to illegitimate parameter values, however, allowed such strings to survive. To diminish the effect of illegitimate sequences, a penalty parameter is added to the objective function to punish semantic violations. Therefore The objective function to be minimized consists of 4 parameters:

\[
\text{fitness} = \text{coarse grain distance } D + \text{fine grain distance } d + \text{history distance } e + \text{semantic violation penalty } P
\]

(1) The coarse grain distance \( D \) between the final state in the SMS and the target state's equivalence class in the CFSM currently aimed at.

(2) The fine grain distance \( d \) estimating the distance yet to be covered on the fine-grained level of implementation states. It consists of a component assessed from the \( d \) values in the SMS so far established \( d_{\text{historic}} \) and the distance between the final state computed at the level of the message causing divergence \( d_{\text{future}} \).

(3) The current length of the message sequence \( L \). This parameter is used to value short message sequences over longer ones.

(4) The semantic violation penalty \( P \) to punish candidates with currently illegitimate parameter values.

The fitness function is then given as:

\[
\text{fitness} = D + d_{\text{future}} + d_{\text{historic}} + L + P
\]

Two selection strategies have been implemented: proportional selection (roulette wheel) and ranking selection [8].
5.2. Evolutionary programming

Evolutionary programming (EP) [15] was originally proposed by Fogel[22]. Conceptually, there are many similarities between EP and GA. The initial considerations that gave rise to compare these two approaches was that GA usually uses fixed length genotypical representations while EP uses variable length phenotypical representations. Further GA uses sexual reproduction while EP uses only asexual reproduction, i.e. new offsprings are derived only by from mutation. Finally, only the second argument remained, since, as described above, we had to use variable length phenotypical representations also in the solution applying GA. Strategically, though, the CFSP played a slightly different role. While in the GA approach, both, crossover and mutation were performed blindly and the CFSP was used only to check the resulting offsprings for their legitimacy and to provide information for computing the overall fitness, in the EP approach, information from the CFSP was indirectly used to constrain mutation. In order to do so, a Grammar $G = (N, \Sigma, P, S)$ was derived from the CFSP such that

- The set of non-terminal symbols $N$ corresponds to the states $Q$ of the CFSP.
- The set of terminal symbols $\Sigma$ was identical for both, the grammar $(\Sigma_g)$ and the finite state machine $(\Sigma_{cfsm})$.
- The set of production rules were derived from the transition function $\delta$. However, a distinction had to be made between edges that were transient (i.e., the test sequence is not yet completed) and nodes that were to be considered as target edges (the test sequence should lead to exactly those states). Thus, one needs type-1 productions of the form $\forall n_i, n_k \in N, \sigma_j \in \Sigma(\delta(n_i, \sigma_j) = n_k \wedge n_k \notin F) \Rightarrow n_i \rightarrow \sigma_j n_k$ and a type-2 productions of the form $\forall n_i, n_k \in N, \sigma_j \in \Sigma(\delta(n_i, \sigma_j) = n_k \wedge n_k \in F) \Rightarrow n_i \rightarrow \sigma_j$.
- Each of these production rules is guarded by the same predicate that guarded the respective item of the transition function.
- The start symbol $S$ corresponds to the initial state $q_0$ of the CFSP.

The Grammar $G$ generates the test sequences admissible for the class under test, i.e., the language $Li(CUT)$. Hence, the phenotypical representation sought was exactly this test sequence.

Following a generative strategy, the EP algorithm started by creating an initial population of $n$ (trivial) elements consisting just of the start symbol. Due to application of different randomly chosen mutation operations, these initially identical elements diverge into a set of different members (message sequences with proper parameters). Thus, in EP, mutation is the crucial operation. In this case, two different versions of mutation operations are needed: insertion mutation appends a new method invocation with randomly chosen parameter settings to a given string, and parameter value mutation randomly changes a parameter value but leaves the method-ids unchanged. Therefore, parameter value mutation also leaves the length of the message string constant.

For insertion mutation, the productions of the grammar play a role in so far, as insertions can only take place at the end of the string (location of the non-terminal) and the algorithm randomly chooses only among productions eligible on the basis of the transformed grammar. Parameter values are generated randomly by a parameter value generator. A length increasing mutation takes place only if the generated parameter is legitimate according to the predicate guarding the production rule used. Thus, with insertion mutation, only legitimate strings can be generated.

Parameter value mutation scans over the string of parameterized messages generated so far and changes the respective parameters according to a random regime considering the parameters’ type. Thus, in a single reproductive step, various parameters might be changed and these changes happen in blindness w.r.t. the grammar and the CFSP. Hence, as with GA, illegitimate strings might result. However, different from GA, there is no reason to keep them for simulating parallel mutations. Any parameter has a chance to mutate anyway due to multiple parameter mutation in a single step. Thus, with EP a check and repair mechanism is invoked. It scans the string up to the first conflicting parameter making this string illegitimate. Here, an attempt to solve the problem is made by replacing the conflicting value by the parameter value from the parent string and the check-and-repair mechanism continues. If this strategy is insufficient due to domino effects of earlier, locally admissible parameter changes, backtracking occurs that might eventually lead to retaining the original individual. Thus, none of the mutation operations causes persisting syntactic violations in the population subjected to EP.

The parameters of the fitness function for EP are similar to the fitness function of GA (Eq. (1)). Parameter $D$ and $L$ are used as in GA, $P$ is not necessary, since all elements of the population are syntactically correct, and $d$ is redefined to take advantage of the information in the predicates attached to production (resp. transition) rules. $d$ is defined by the divergence of the actual parameter value from the value(s) in the predicate that would allow this string to progress with minimal $D$ to the target state. This distance can be computed for each node, as the CFSP contains often a moderate number of equivalence classes.

After mutation the population of $n$ individuals is duplicated. Before entering the next evolutionary cycle, the population has to be reduced again to $n$ members. This selection is done by a $q$-tournament selection strategy [3] such that the parameter $q$ determines how many randomly selected members take part in a tournament. The rank obtained in the tournament on the basis of the fitness value of the individual with respect to other tournament members determines the member’s survival.

Two stopping criteria were defined. The EP algorithm terminates either after a predefined maximal number of iterations has been reached or if all of the population members reached the target state. Stopping only after all members reached the target state and not already if one member does serves to avoid premature termination with suboptimal (i.e. too long) message sequences.

5.3. Ant-colony system

This approach was designed in full contrast to the GA- and EP-solution. It had its justification in the successful use of ant systems and ant-colony algorithms (AS and ACS) for graph traversal problems such as the traveling salesman problem. As the CFSP and the related CFSP are graphs and both, the state-reachability as well as the state-coverage problem can be seen as graph traversal problems, it seemed appropriate to compare the effectiveness of ACS with the strategies mentioned so far.

Ant systems fall into the category of swarm intelligence systems [10,33]. The basic idea behind ant systems (AS) is that a number of relatively simple individuals (ants) autonomously explore a search space. These individuals cooperate indirectly by leaving some information in their environment which helps others for heuristic orientation in the search space. In nature, this information transfer is provided by ants marking the trail they passed by some chemical substance, called pheromone. Foraging ants have a higher probability to follow a trail with a high concentration of pheromone than to follow an unmarked trail or a trail with low pheromone concentration. As ants returning from a food source reemphasize the pheromone concentration, others will
recognize a short path. As pheromone evaporates over time, paths to depleted food sources will eventually be given up. The classical paper of using ant systems for optimization problems reported the application of AS to the traveling salesman problem by Dorigo et al. [18]. Ant-colony system (ACS) is an extension of the AS-algorithm as proposed by Dorigo and Gambardella [17,19].

Formally, ACS rest on the idea that a combinatorial optimization problem \( P = (\Omega, f) \) is defined by a search space \( \Omega \) over a finite set of discrete decision variables \( X_i \), a set of constraints \( \Omega \), and an objective function \( f: \Omega \rightarrow R \) which is to be minimized. The generic variables \( X_i \) take values from a domain set \( D_i = \{ v_1, \ldots, v_{N(i)} \} \). A feasible solution \( s \in \Omega \) is a complete assignment of values to variables that satisfies all constraints in \( \Omega \). \( s \in \Omega \) is a global optimum if \( \forall s': \Omega : f(s') \leq f(s) \).

The solution is sought by having a population of artificial ants traverse the problem space (graph), evaluate the solution, and, depending on the quality of the solution obtained by an individual ant, update the pheromone concentration on the path it has followed. This algorithm runs till a termination condition is met. It can be further improved by some local search heuristics.

With the test data generation problem, the state-coverage problem amounts to finding shortest paths from the origin to all states. The reachability problem involves finding the shortest path to a given target state. Thus, the ant arrives at the next state with a certain probability given, depending on the quality of the solution obtained by an individual ant, update the pheromone concentration on the path it has followed. This algorithm runs till a termination condition is met. It can be further improved by some local search heuristics.

5.4. Summary of algorithms

Before entering the evaluation section, one might mention already that the three meta-heuristics used fall into two diverse categories, variations of genetic algorithms and swarm intelligence. In all three cases, the interaction between parameter values to be chosen and the fine granular state information that has to be preserved to adequately account for the curricular history of the CUT caused specific problems for the algorithms. In case of GA and EP it may cause offsprings to be illegitimate according to the “grammar” of the input language admitted by the CUT, in case of ACS it might cause ants to get stuck at a certain node. It will be interesting how the solutions foreseen cope with this problem during the evaluation.

A further aspect to be mentioned is that due to the phenotypical representation used with GA, the basic ideas underlying the GA and the EP-solution differed less than foreseen when designing this experiment. With different strategies of handling the case of illegitimate offsprings, there is still some merit in exploring both solutions.

6. NITOT environment

The development of the environment for nature-inspired techniques for object-oriented testing (NITOT environment) intends to provide a general framework for software testing, in particular for conformance testing. As illustrated in Fig. 3, NITOT consists of the following components:

- The Program loader is responsible for loading to memory the class under test (CUT) of the system under test (SUT).
- The Specification loader is responsible for loading the specification of the CUT or SUT. Depending upon the meta-heuristic (triggered by the case generator), a particular form of the FSM (the FSM corresponding to the CUT’s specification) is used. This automaton provides the basic information for the nature-inspired optimizer.
- The Monitoring unit is responsible for monitoring the testing process by specifying progressively the testing adequacy criteria (test coverage) that the optimization should meet and activating the simulator for launching the process searching for test cases and the execution driver that executes the selected class with the automatically generated test cases. Thus, the monitoring unit includes:
  - Class collector that specifies the class or the set of classes to be checked for conformance and their specifications.
- Data manager that tracks the optimization goals and manages the test cases during the optimization process.
- Simulator that coordinates the process of test case execution of the CUT. It activates and controls the execution driver and the case generator.
- Comparator compares the results obtained from executing the test driver with the content of the actual state of the specification that served as target state of the respective message sequence.

- The Case generator gathers test cases from the nature-inspired algorithms. It also provides some utilities used by these algorithms. Nature-inspired algorithms are search strategies that are applied on the specifications of the CUT to generate adequate test cases. At this stage, only three algorithms (evolutionary programming, ant-colony optimization and genetic algorithm) are available, but alternative strategies can be plugged in in the future.
- The Execution driver serves to execute the CUT using the test cases obtained by the test case generator.

7. Experimental evaluation

For the sake of evaluation, we use several benchmarks. Since the objective of the present study is to deal with isolated classes, we have chosen benchmarks that represent individual classes but modeling an entire application. In Section 7.1, each of these benchmarks is briefly described.

In this evaluation study, we applied the simple all-states coverage which is achieved when the test reaches every state in the model at least once. The coverage results and analysis of each of the proposed algorithms are discussed in Sections 7.2–7.4.

7.1. Benchmarks

The following five benchmark specifications have been used for assessing the relative merits of the evaluated algorithms:

1. **Account**: This benchmark has been used at several occasions in testing literature [30]. The class implements a bank account with possible operations on it (see Fig. 4). It has 5 states and 17 transitions.
2. **Locks**: This class has been created by the authors. It models the consumer-producer problem and consists of 7 states and 12 transitions (see Fig. 5). This benchmark is useful because the messages have Boolean parameters, which are known to cause difficulties in white-box testing.
3. **Stack**: This benchmark models the behavior of a stack. The Stack consists of 4 states and 7 transitions (see Fig. 6). It also contains messages with Boolean parameters and predicates with Boolean variables. Stack is further of interest as it has, on the basis of a very simple FSM-model for specifying the stacks behavior, a tricky interaction between the usual FSM on the specification level and the related state model on the implementation level. State full will be reached only after a sufficient number of push operations without intermittent pop operations: having been performed (strictly, push(\_), pop(\_)) > (stackCapacity - 1)).
4. **Coffee Vendor Machine (CVM)**: This benchmark models the behavior of a coffee machine after inserting a coin. CVM
consists of 5 and 10 transitions (see Fig. 7). It also contains messages with Boolean parameters and predicates with Boolean variables.

(5) **Exam**: This is also proposed by the authors. The particularity of this benchmark is that attributes and method parameters are of character type. It consists of 7 states and 6 transitions (see Fig. 8). Beyond this, Exam has the interesting feature that it models certain aspects of the curriculum of the object represented, but these aspects are dependent on some internal variable (Ans) that matters for the legitimacy of transitions from state **After Second** but are not explicitly captured in the state information **After Init** and **After First**. Thus, in an exam of a modeling class, a student producing such a state diagram to represent the problem would probably flunk the exam. However, in practical situations, one does meet state diagrams with such hidden interactions. Therefore, the example has been introduced into the evaluation benchmark to see how the algorithm would cope with such implicit interactions between specification states and implementation states.

The diagrams in Figs 4–8 graphically represent CFSM’s. The circles indicate the equivalence classes of states, labeled with the label of the specification FSM. The arrows show the equivalence classes of transitions. They are labeled by the method-id characterizing this transition equivalence class with its associated parameter. The angular brackets enclose the predicates on the transition. The curly brackets include the actions to be executed once the transition is traversed.

---

7.2. **Evaluation of GA**

Following the description of the methodology described in Section 5.1, the genetic algorithm produces chromosomes whose quality (fitness) is controlled by the closeness (or inversely the distance) to the target state in the FSM. The GA starts with a population of n chromosomes of length 1 and iterates by applying the genetic operators. Depending on the evaluation goals, the algorithm stops when either a solution is found, the average fitness exceeds a threshold, the execution time exceeds a certain threshold, or the number of iterations exceeds a certain maximum.

GA is run on the five benchmarks Account, CVM, Stack, Lock, and Exam using the parameter setting shown in Table 1. Note that for the stopping criterion is **avgFitness < 30**. This is because the value of **D** (in Eq. (1)) is multiplied by 100 (for scaling purposes).

### Table 1

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value/strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>40</td>
</tr>
<tr>
<td>Penalty, K</td>
<td>50</td>
</tr>
<tr>
<td>Insert probability</td>
<td>0.05</td>
</tr>
<tr>
<td>Delete probability</td>
<td>0.05</td>
</tr>
<tr>
<td>Substitute probability</td>
<td>0.05</td>
</tr>
<tr>
<td>Parameter update probability</td>
<td>0.05</td>
</tr>
<tr>
<td>Generation</td>
<td>Simple</td>
</tr>
<tr>
<td>Selection</td>
<td>Ranking</td>
</tr>
<tr>
<td>Stopping criterion</td>
<td>avgFitness &lt; 30</td>
</tr>
</tbody>
</table>
Any sequence with a fitness value less than 100 indicates that the final state is the target state; hence the criterion allows to find a group of solutions not only one. We set stopping criterion to 30, but it could be set to any value below 100 (smaller values are preferred to obtain multiple solutions). The other parameters are set based on either default values (mutation, crossover) or initial empirical experiments (typical parameters: penalization, average fitness, etc.). As these parameter values are crucial with all algorithms, we run the experiments 30 times each.

The values of the parameters are obtained by varying each parameter (or strategy) in its allowed range (resp. set). This analysis was guided by the legitimacy ratio of the population (meaning the proportion of legitimate runnable sequences). GA is then executed on the five benchmarks to fulfill the all-states coverage. This has has been achieved after different durations as shown in Table 2. The most demanding benchmark is Exam, while the less demanding one is Account (Table 2).

GA was able to achieve full coverage for each of the benchmarks. For the Account case Table 3, each state is reached in less than a second. The shortest sequences of length 2 (e.g., account(), withdraw(100)) are found as shown in Fig. 9. For some states the average legitimacy ratio is very low (15% of the population is legitimate). This is because during the algorithms initialization the population is generated randomly, usually with a low legitimacy ratio. However, the most important issue to find at least one test case has been achieved.

In the CVM benchmark case, reaching the state busy took twice as long than the longest sequence in the Account’s FSM (Table 4).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Cov. time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td>1.589</td>
</tr>
<tr>
<td>CVM</td>
<td>2.021</td>
</tr>
<tr>
<td>Stack</td>
<td>2.264</td>
</tr>
<tr>
<td>Locks</td>
<td>6.187</td>
</tr>
<tr>
<td>Exam</td>
<td>7.253</td>
</tr>
</tbody>
</table>
GA was able to deal with this benchmark containing non-parameterized methods in the specification in an efficient way. Likewise, the state full in the Stack benchmark requires a long message sequence. However, in this example it can be seen from the average solution length that the algorithm is continuously adding messages to the sequence (Fig. 10).

The Locks benchmark is solved quickly by GA. 100% coverage is reached (Table 6). The Boolean parameters in the specification are

<table>
<thead>
<tr>
<th>State</th>
<th>Avg #Iter</th>
<th>Avg time</th>
<th>Avg coverage</th>
<th>Avg #Iter for the 1st solution</th>
<th>Avg legitimacy ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overdrawn</td>
<td>16.7</td>
<td>0.812</td>
<td>100%</td>
<td>12.40</td>
<td>82%</td>
</tr>
<tr>
<td>inCredit</td>
<td>1.0</td>
<td>0.015</td>
<td>100%</td>
<td>1.00</td>
<td>15%</td>
</tr>
<tr>
<td>Blocked</td>
<td>7.6</td>
<td>0.381</td>
<td>100%</td>
<td>1.00</td>
<td>15%</td>
</tr>
<tr>
<td>Final</td>
<td>7.7</td>
<td>0.381</td>
<td>100%</td>
<td>4.10</td>
<td>75%</td>
</tr>
</tbody>
</table>
efficiently handled. Fig. 11 pertaining to the state emptyCLock shows the typical evolution of the message length together with the objective minimization values as the number of iterations increases. Moreover, the Stack benchmark is also processed conveniently by GA achieving 100% coverage (Table 5). However, reaching the state Full requires a long message sequence as shown in Fig. 12.

The Exam benchmark is the most difficult one among all. On the path leading to the state good, it contains an equality condition, which refers to a transition encountered in previous steps. Despite that, 100% coverage in each run was reached (Table 7). With the state good, the convergence requires more iterations because of the predicate on the transition leading to it. Usually equality conditions are hard to fulfill and might lead even to a plateau as shown in Fig. 13. On average the state good was reached after 71 iterations.

These experiments show that GA, as proposed in this paper, is an interesting approach to deal with conformance testing.

![Fig. 9. Overview of the state overdrawn (Account).](image)

![Fig. 10. Overview of the state buy (CVM).](image)

![Fig. 11. Overview of the state empty (Locks).](image)

---

### Table 4

Results of the benchmark CVM.

<table>
<thead>
<tr>
<th>State</th>
<th>Avg #iter</th>
<th>Avg time</th>
<th>Avg coverage</th>
<th>Avg #iter for the 1st solution</th>
<th>Avg legitimacy ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Busy</td>
<td>21.6</td>
<td>0.990</td>
<td>100%</td>
<td>18.30</td>
<td>75%</td>
</tr>
<tr>
<td>Idle</td>
<td>10.3</td>
<td>0.309</td>
<td>100%</td>
<td>6.30</td>
<td>69%</td>
</tr>
<tr>
<td>notEmpty</td>
<td>18.6</td>
<td>0.715</td>
<td>100%</td>
<td>6.30</td>
<td>69%</td>
</tr>
<tr>
<td>Off</td>
<td>1.0</td>
<td>0.007</td>
<td>100%</td>
<td>1.00</td>
<td>20%</td>
</tr>
</tbody>
</table>

### Table 5

Results of the benchmark Stack.

<table>
<thead>
<tr>
<th>State</th>
<th>Avg #iter</th>
<th>Avg time</th>
<th>Avg coverage</th>
<th>Avg #iter for the 1st solution</th>
<th>Avg legitimacy ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>1.0</td>
<td>0.022</td>
<td>100%</td>
<td>1.00</td>
<td>35%</td>
</tr>
<tr>
<td>notEmpty</td>
<td>5.7</td>
<td>0.167</td>
<td>100%</td>
<td>2.40</td>
<td>70%</td>
</tr>
<tr>
<td>full</td>
<td>19.5</td>
<td>2.075</td>
<td>100%</td>
<td>15.70</td>
<td>82%</td>
</tr>
</tbody>
</table>

### Table 6

Results of the benchmark Locks.

<table>
<thead>
<tr>
<th>State</th>
<th>Avg #iter</th>
<th>Avg time</th>
<th>Avg coverage</th>
<th>Avg #iter for the 1st solution</th>
<th>Avg legitimacy ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>emptyUnlock</td>
<td>1.0</td>
<td>0.017</td>
<td>100%</td>
<td>1.00</td>
<td>23%</td>
</tr>
<tr>
<td>emptyLocked</td>
<td>40.2</td>
<td>2.601</td>
<td>100%</td>
<td>36.50</td>
<td>75%</td>
</tr>
<tr>
<td>notEmptyPLocked</td>
<td>12.7</td>
<td>0.399</td>
<td>100%</td>
<td>8.90</td>
<td>75%</td>
</tr>
<tr>
<td>notEmptyUnlock</td>
<td>21.9</td>
<td>1.065</td>
<td>100%</td>
<td>17.70</td>
<td>75%</td>
</tr>
<tr>
<td>emptyPLocked</td>
<td>10.0</td>
<td>0.301</td>
<td>100%</td>
<td>4.30</td>
<td>71%</td>
</tr>
<tr>
<td>notEmptyLocked</td>
<td>32.9</td>
<td>1.804</td>
<td>100%</td>
<td>29.10</td>
<td>76%</td>
</tr>
</tbody>
</table>
However, equality predicates pose a challenge to the algorithm. More discussion on this will follow in Section 7.5.

7.3. Evaluation of EP

EP relies on many parameters that need to be set. These include the population size ($n$), the insertion mutation probability ($p_{IM}$), the parameter value mutation probability ($p_{PVM}$), the tournament selection parameter ($q$), the maximum integer number ($b_{max}$), the maximum mutating value ($b_{mutation}$), and the number of iterations ($NumOfIterations$). The assigned values are shown in Table 8.

Table 8
Setting of EP parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value (Exam)</th>
<th>Account, CVM, Stack, Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>$p_{IM}$</td>
<td>0.4</td>
<td>0.8</td>
</tr>
<tr>
<td>$p_{PVM}$</td>
<td>0.2</td>
<td>0.05</td>
</tr>
<tr>
<td>$q$</td>
<td>$2n$</td>
<td>$n$</td>
</tr>
<tr>
<td>$b_{max}$</td>
<td>10,000</td>
<td>10,000</td>
</tr>
<tr>
<td>$b_{mutation}$</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>NumOfIterations</td>
<td>200</td>
<td>200</td>
</tr>
</tbody>
</table>

However, equality predicates pose a challenge to the algorithm. More discussion on this will follow in Section 7.5.

Table 7
Results of the benchmark Exam.

<table>
<thead>
<tr>
<th>State</th>
<th>Avg #iter</th>
<th>Avg time</th>
<th>Avg coverage</th>
<th>Avg #iter for the 1st solution</th>
<th>Avg legitimacy ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>AfterSecond</td>
<td>8.9</td>
<td>0.287</td>
<td>100%</td>
<td>8.9</td>
<td>83%</td>
</tr>
<tr>
<td>Bad</td>
<td>11.7</td>
<td>0.539</td>
<td>100%</td>
<td>7.9</td>
<td>78%</td>
</tr>
<tr>
<td>Medium</td>
<td>53.6</td>
<td>2.782</td>
<td>100%</td>
<td>50.2</td>
<td>78%</td>
</tr>
<tr>
<td>Good</td>
<td>71.1</td>
<td>3.645</td>
<td>100%</td>
<td>67.7</td>
<td>84%</td>
</tr>
</tbody>
</table>

Fig. 12. Overview of the state full (Stack).

Fig. 13. Overview of the state good (Exam).

Table 9
Coverage results of the benchmarks by EP.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#Iter.</th>
<th>Time (ms)</th>
<th>Coverage</th>
<th>Optimal sol.</th>
<th>Difficult state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td>18.00</td>
<td>689</td>
<td>100%</td>
<td>100%</td>
<td>Blocked</td>
</tr>
<tr>
<td>CVM</td>
<td>24.30</td>
<td>460</td>
<td>100%</td>
<td>100%</td>
<td>Busy</td>
</tr>
<tr>
<td>Stack</td>
<td>26.40</td>
<td>976</td>
<td>100%</td>
<td>100%</td>
<td>Full</td>
</tr>
<tr>
<td>Locks</td>
<td>43.00</td>
<td>684</td>
<td>100%</td>
<td>100%</td>
<td>EmptyLocked</td>
</tr>
<tr>
<td>Exam</td>
<td>206.78</td>
<td>434</td>
<td>100%</td>
<td>100%</td>
<td>Good</td>
</tr>
</tbody>
</table>

Fig. 14. Minimization of the objective function.
most demanding states portrayed in column 6 of Table 9 for the Account, CVM, Locks, and Stacks benchmarks. Details related to the state good of the benchmark Exam are shown in Fig. 16.

It is important to note that EP is very effective in handling the set of benchmarks. Compared to GA, EP looks even more efficient as suggested by Tables 2 and 9.

7.4. Evaluation of ACS

In order to set the parameters of the the ACS algorithm, a reasonably complex graph has been devised. It contains 27 states and 81 transitions, but without conditions on the transitions and without appended actions. The set of parameters influencing the cooperation between ants are $\tau_0 \in (0, \infty)$ (the initial pheromone on every transition), $p_0 \in [0, 1]$ (the transition probability threshold, higher values of $p_0$ promote exploitation of the pheromone and low values of $p_0$ encourage ants to randomly explore the search space), $\varphi \in [0, 1]$ (pheromone evaporation (actually $1 - \varphi$) higher value leads to faster evaporation), $\rho \in [0, 1]$ is a parameter used in the local update, a high value leads to high evaporation).

The best value of $\tau_0$ corresponding to the shortest path between a given starting node and a given target node was found to be 0.2, $\rho = 0.4$, $\varphi = 0.09$, and $p_0 = 0.6$. Moreover, the number of ants is set to 4, the number of max iterations is set to 10, and the number of max steps (standing for the number of times an ant is allowed to restart a tour) is set to 60. Intuitively, the number of steps has to be at least as large as the length of the shortest path to the target state. In fact, on average the optimal path is 15 transitions length. However, increasing this number excessively implies much computational time.

Relying on these parameters, the ACS algorithm has easily achieved the all-states coverage on three benchmarks Account, CVM and Locks as shown in Figs. 17–19. The maximum number of iterations required by ACS to reach any state in these benchmarks is 3, which is small. This illustrates the efficiency of the ACS algorithm.

More interesting is the case of the states full and good in the Stack and Exam problems (Figs. 20, 22). They have not been covered by the algorithm. However, after a close look into the reason, it turned out that the state full requires a larger number of iterations to get covered. It is easy to obtain at least one sequence that reaches the state full as shown in Fig. 21. As to the state good in Exam (see Fig. 22), the condition on the transition leading to it, that is $\text{Ans}1 = \text{"B" } \& \& \text{ Ans}2 = \text{"x"}$, has to be fulfilled. It seems that the characters ‘B’ and ‘x’ could not be generated so that this condition gets satisfied, meaning that the objective function has reached a plateau such that successive iterations are not producing better results anymore.
7.5. Discussion

The three algorithms look very efficient in dealing with the problem of conformance testing. Although the benchmarks are not too complex (for the sake of illustration), some of them pose challenges as described in Sections 7.1–7.4. The most difficult issue is the equality predicates. In order to get them fulfilled, an exact value is required. Without additional execution time, equality conditions are hardly satisfied. However, for an efficient treatment, additional investigations are required to devise mechanisms to guide the search towards particular values of interest.

An another problem that requires more attention is when the fulfillment of a predicate is indirectly dependent on previous actions. For instance in Fig. 23, the CFSM contains a single Boolean variable \( V_1 \), initially set to false. The condition on the transition from \( S_1 \) to \( S_2 \) cannot be satisfied unless the transition from \( S_1 \) to \( S_1 \) is traversed and \( M_1() \) is called with a value 7777. In such a scenario one needs some chaining mechanisms [21] that allow tracking back the definition of the variable so that an updated sequence is generated including \( M_1() \). This allows to set \( V_1 = true \) and then \( M_1() \) can be executed to reach the goal state \( S_2 \).

The proposed techniques are efficient when the CFSM reflects the correct specifications of the problem. They offer the opportunity to search the test suite via an pseudo-parallel exploration methodology. Moreover, EA and GA are similar in terms of coverage, but they differ in terms of efficiency. EP appears to be the most efficient approach. The reason for this stems from the fact that EP constructs feasible solutions, while GA seems to work blindly due mainly to the crossover operation. ACS, on the other hand, was trapped by examples where one specification state is mapped to a single point in a large number of implementation states.

As a last observation, the coverage criterion may not be strong enough in generating conformance test suites. Other criteria such all transitions, all \( n \)-transitions, etc. are planned to be studied in future investigations.

8. Conclusion

The present paper deals with the problem of conformance testing. It suggests the CFSM formalism that allows to check a class against its specification. Three nature-inspired meta-heuristics are proposed, each with a particular respective on the CFSM (set of words, regular grammar, graph). These meta-heuristics have been evaluated on various benchmarks. Evolutionary programming has been found to be slightly better than the other meta-heuristics due to its knowledge-based approach. However, the first aim is to show that such meta-heuristics can efficiently deal with the problem of test case generation for the purpose of conformance testing knowing that each of these heuristics needs a particular representation.

As a primary future work, we intend to follow the same line of research but for the cluster level, and system level relying on class CFSMs combination. Moreover, mechanisms for solving equality predicates and string based predicates need to be devised.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments which helped to improve the manuscript.

References
