Feature Selection using Evolutionary Computation Techniques for Software Product Line Testing

Alfredo Ibias
Universidad Complutense de Madrid
Madrid, Spain
aibias@ucm.es

Luis Llana
Universidad Complutense de Madrid
Madrid, Spain
llana@ucm.es

Abstract—Software product lines are an excellent mechanism in the development of software. Testing software product lines is an intensive process where selecting the right features where to focus it can be a critical task. Selecting the *best* combination of features from a software product line is a complex problem addressed in the literature. In this paper, we address the problem of finding the combination of features with the highest probability of being requested from a software product line with probabilities. We use Evolutive Computation techniques to address this problem. Specifically, we use the Ant Colony Optimization algorithm to find the *best* combination of features. Our results report that our framework overcomes the limitations of the brute force algorithm.

Index Terms—Software Testing, Evolutionary Computation, Software Product Lines

I. INTRODUCTION

During the last years, software product lines (in short, SPLs) have become a widely adopted mechanism for efficient software development. They are a set of similar software-based systems produced from a set of software features that are shared between them using a common means of production. The main goal of SPLs is to increase the productivity of creating software products. They achieve this goal by selecting those software systems that are better for a specific criterion (e.g., a software system is less expensive than others; it requires less time to be executed, etc.). Currently, different approaches for representing the product line organisation can be found in the literature, such as FODA [38], RSEB [30], PLUSS [28] and SPLA [2]

The formal langage SPLA was introduced in [4]. The authors present a formal language capable to express the FODA diagrams (Figure 1 shows some examples). A recent work [14] has proposed a probabilistic extension to SPLA: SPLA$^\mathcal{P}$. This proposal includes a probability whenever there is a choice in the representation of the SPL. This probability allows us to know which features are requested more frequently and which feature combinations are the most popular ones. This knowledge is beneficial to make decisions about the SPL, the resources destined to each feature, and the SPL updates.

Software testing [1] is the main validation technique to assess the reliability of complex software systems. When
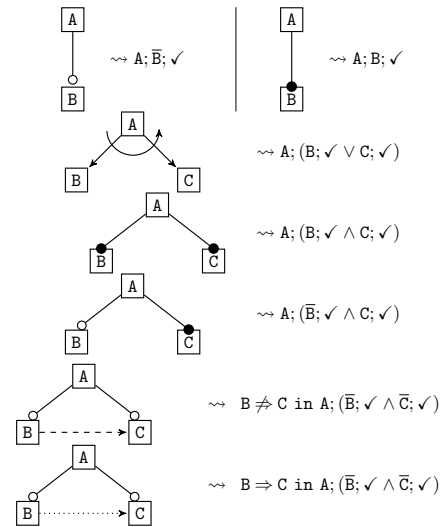
Fig. 1: Examples of translation from FODA Diagrams into SPLA.

testing SPLs [18], it is crucial to distribute the resources between the different features of the line in a smart way. We are particularly concerned about the testing resources assigned to each component of the software system. These resources are limited so, if we had information about the most requested components of the software system (what we have called features) we could assign them more testing resources. Therefore, the information about the more requested features from the SPL can be vital to distribute the resources during testing. This problem has been addressed in [14]. Besides, when producing SPLs, we want to pay attention to have a better engagement between the features that are more commonly shipped together. Therefore, more development and testing resources are ideally focused on those features and their engagement. However, it is not trivial (and sometimes not feasible) to know the probability of use of all the possible feature combinations that a product line can produce. Therefore, being able to know which feature combinations are more used can be critical when testing SPLs. In this paper, we present an approach to get those feature combinations.

Heuristic search algorithms are techniques commonly used in Mathematics and Computer Science either to optimize a

function or to find the best possible solution for a given problem. These techniques, also referred to as *metaheuristics*, can be roughly divided into three categories: *global search techniques* such as simulated annealing [40], *evolutive techniques* such as genetic algorithms [29], and *constructive techniques* such as Ant Colony Optimization [26], [27].

In this paper, we have used an Evolutionary Computation technique to select features from a SPL. We have chosen this particular family of techniques because it is well suited for parallel searching, and it also combines the knowledge obtained by each member of the population of candidate solutions. Furthermore, by starting with a random set of candidate solutions, the algorithm can quickly obtain a candidate solution that suits our goal. More specifically, we have implemented a variation of a typical Evolutionary approach: the Ant Colony Optimization (ACO) algorithm [26], [27]. In this variation we have combined the classical ACO algorithm with a SPLA$^{\mathcal{P}}$ expressions interpreter to be able to search, in an *a priory* unknown search space, the feature combination with a higher probability that the SPLA$^{\mathcal{P}}$ expression can produce.

Then, we show the results of some experiments. In them, we can clearly see how our framework can solve the feature selection problem we have raise, beating in time (saving around 67% of time) to the standard brute force algorithm. Also, we show how our framework can obtain as good results as the ones obtained by the brute force algorithm, getting in mean feature combinations with only an 18% less probability. We also analyse some extreme cases, where the randomisation factors have had a high impact.

Finally, we would like to mention that the use of metaheuristics in testing is not new [3], [5], [21], [23], [32], [33], [36], [45]. In particular, there is some work on the application of the swarm idea to testing [31], [48]. The novelty of our approach resides in the fact that we are using this metaheuristics to the feature selection problem as a previous step before properly testing an SPL.

The rest of the paper is organised as follows. In Section II, we present some theoretical concepts that we use along with our paper. In Section III, we introduce our feature selection framework. In Section IV, we present our experiments and discuss the results. In Section V, we review some of the possible threats to the validity of our results. Finally, in Section VI, we give conclusions and outline some directions for future work.

II. Preliminaries

In this section we briefly introduce the concepts that will be used along the work. First, we define the concepts of SPL, software feature, and SPLA$^{\mathcal{P}}$ process algebra.

A Software Product Line (SPL) is, as defined by The Carnegie Mellon Software Engineering Institute, "a set of software-intensive systems that share a common managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [42]. The idea of SPLs is that they allow producing, from a set of predefined software features, a piece of software that includes all those features. SPLA$^{\mathcal{P}}$ [14] extends these classic SPLs concepts with a notion of probability. This probability indicates the preferences of the users in front of a choice.

We have decided to represent SPLs using the SPLA$^{\mathcal{P}}$ process algebra [14]. In order to work with this algebra, we will consider a set of *features*, denoted by $\mathcal{F}$, and the elements A, B, C, . . . will stand for elements of $\mathcal{F}$. We have a special feature $\checkmark \notin \mathcal{F}$ to mark the end of a product. We consider non-degenerated probabilities in the syntax, that is, for all probability p we have $0 < p < 1$.

Definition 1: A *probabilistic software product line* is a term generated by the following grammar:

$$P ::= \quad \checkmark \mid \mathtt{nil} \mid \mathtt{A}; P \mid \overline{\mathtt{A}};_p P \mid P \vee_p Q \mid P \wedge Q \mid$$
$$\mathtt{A} \not\Rightarrow \mathtt{B} \text{ in } P \mid \mathtt{A} \Rightarrow \mathtt{B} \text{ in } P \mid P\backslash\mathtt{A} \mid P \Rightarrow \mathtt{A}$$

where $\mathtt{A}, \mathtt{B} \in \mathcal{F}$, $\checkmark \notin \mathcal{F}$ y $p \in (0, 1)$. The set of all software product lines is denoted by SPLA$^{\mathcal{P}}$.

This probabilistic process algebra has an operational semantics to guide how to interpret the expressions of the algebra. The operational semantics rules are displayed in Figure 2. A relevant property of this algebra is that each time we find a probability, the feature from the left-hand side gets the probability p, and the feature from the right-hand side gets the probability $1 - p$, but those are not the full probabilities of each feature. In fact, the probability of a single feature in a SPL is a measure of its occurrences in the set of products.

The operational semantics of an SPLA$^{\mathcal{P}}$ expression P is a tree structure. The set of products of P is computed by traversing this tree.

Definition 2: Let $P, Q \in$ SPLA$^{\mathcal{P}}$. We write $P \overset{s}{\Longrightarrow}_p Q$ if there exists a sequence of consecutive transitions

$$P = P_0 \xrightarrow{a_1}_{p_1} P_1 \xrightarrow{a_2}_{p_2} P_2 \cdots P_{n-1} \xrightarrow{a_n}_{p_n} P_n = Q$$

where $n \geq 0$, $s = a_1 a_2 \cdots a_n$ and $p = p_1 \cdot p_2 \cdot \cdots \cdot p_n$. We say that s is a trace of P.

Let $s \in \mathcal{F}^*$ be a trace of P. We define the product $\lfloor s \rfloor \subseteq \mathcal{F}$ as the set consisting of all features belonging to s.

Let $P \in$ SPLA$^{\mathcal{P}}$. We define the set of probabilistic products of P, denoted by $\mathtt{prod}^{\mathcal{P}}(P)$, as the set

$$\mathtt{prod}^{\mathcal{P}}(P) = \big\{(pr, p) \mid p > 0 \wedge p = \sum \{q \mid P \overset{s\checkmark}{\Longrightarrow}_q Q$$
$$\wedge \lfloor s \rfloor = pr\}\big\}$$

However, computing this tree is computationally expensive, and can be infeasible in some cases. That is the reason we need to work with evolutive computation techniques. And as this tree can be seen as a directed graph, then a convenient evolutive technique to search this tree as a search space is the Ant Colony Optimization algorithm.

The Ant Colony Optimization algorithm is a well known algorithm in the Evolutionary Computation field. It is a distributed algorithm of search in a graph-like search space. It consists of a set of *ants*, that are the agents that explore the search space. Then, each ant look for the shortest path from the initial node to the target node, choosing their next move based on a random choice modified by the weigh of each

$$[\textbf{tick}] \quad \checkmark \xrightarrow{\checkmark}_1 \texttt{nil} \qquad\qquad [\textbf{feat}] \quad \texttt{A};P \xrightarrow{\texttt{A}}_1 P$$

$$[\textbf{ofeat1}] \quad \overline{\texttt{A}};_p P \xrightarrow{\texttt{A}}_p P \qquad\qquad [\textbf{ofeat2}] \quad \overline{\texttt{A}};_p P \xrightarrow{\checkmark}_{(1-p)} \texttt{nil}$$

$$[\textbf{cho1}] \quad \frac{P \xrightarrow{\texttt{A}}_p P_1}{P \vee_q Q \xrightarrow{\texttt{A}}_{p\cdot q} P_1} \qquad\qquad [\textbf{cho2}] \quad \frac{Q \xrightarrow{\texttt{A}}_q Q_1}{P \vee_p Q \xrightarrow{\texttt{A}}_{(1-p)\cdot q} Q_1}$$

$$[\textbf{con1}] \quad \frac{P \xrightarrow{\texttt{A}}_p P_1}{P \wedge Q \xrightarrow{\texttt{A}}_{\frac{p}{2}} P_1 \wedge Q} \qquad\qquad [\textbf{con2}] \quad \frac{Q \xrightarrow{\texttt{A}}_q Q_1}{P \wedge Q \xrightarrow{\texttt{A}}_{\frac{q}{2}} P \wedge Q_1}$$

$$[\textbf{con3}] \quad \frac{P \xrightarrow{\checkmark}_q \texttt{nil},\, Q \xrightarrow{\checkmark}_p \texttt{nil}}{P \wedge Q \xrightarrow{\checkmark}_{p\cdot q} \texttt{nil}}$$

$$[\textbf{con4}] \quad \frac{P \xrightarrow{\texttt{A}}_p P_1,\, Q \xrightarrow{\checkmark}_q \texttt{nil}}{P \wedge Q \xrightarrow{\texttt{A}}_{\frac{p\cdot q}{2}} P_1} \qquad\qquad [\textbf{con5}] \quad \frac{P \xrightarrow{\checkmark}_p \texttt{nil},\, Q \xrightarrow{\texttt{A}}_q Q_1}{P \wedge Q \xrightarrow{\texttt{A}}_{\frac{p\cdot q}{2}} Q_1}$$

$$[\textbf{req1}] \quad \frac{P \xrightarrow{\texttt{C}}_p P_1,\ \texttt{C} \neq \texttt{A}}{\texttt{A} \Rightarrow \texttt{B in } P \xrightarrow{\texttt{C}}_p \texttt{A} \Rightarrow \texttt{B in } P_1} \qquad\qquad [\textbf{req2}] \quad \frac{P \xrightarrow{\texttt{A}}_p P_1}{\texttt{A} \Rightarrow \texttt{B in } P \xrightarrow{\texttt{A}}_p P_1 \Rightarrow \texttt{B}}$$

$$[\textbf{req3}] \quad \frac{P \xrightarrow{\checkmark}_p \texttt{nil}}{\texttt{A} \Rightarrow \texttt{B in } P \xrightarrow{\checkmark}_p \texttt{nil}}$$

$$[\textbf{excl1}] \quad \frac{P \xrightarrow{\texttt{C}}_p P_1,\ \texttt{C} \neq \texttt{A},\ \texttt{C} \neq \texttt{B}}{\texttt{A} \not\Rightarrow \texttt{B in } P \xrightarrow{\texttt{C}}_p \texttt{A} \not\Rightarrow \texttt{B in } P_1} \qquad\qquad [\textbf{excl2}] \quad \frac{P \xrightarrow{\texttt{A}}_p P_1}{\texttt{A} \not\Rightarrow \texttt{B in } P \xrightarrow{\texttt{A}}_p P_1 \backslash \texttt{B}}$$

$$[\textbf{excl3}] \quad \frac{P \xrightarrow{\texttt{B}}_p P_1}{\texttt{A} \not\Rightarrow \texttt{B in } P \xrightarrow{\texttt{B}}_p P_1 \backslash \texttt{A}} \qquad\qquad [\textbf{excl4}] \quad \frac{P \xrightarrow{\checkmark}_p \texttt{nil}}{\texttt{A} \not\Rightarrow \texttt{B in } P \xrightarrow{\checkmark}_p \texttt{nil}}$$

$$[\textbf{forb1}] \quad \frac{P \xrightarrow{\texttt{B}}_p P_1,\ \texttt{B} \neq \texttt{A}}{P\backslash\texttt{A} \xrightarrow{\texttt{B}}_p P_1\backslash\texttt{A}} \qquad\qquad [\textbf{forb2}] \quad \frac{P \xrightarrow{\checkmark}_p \texttt{nil}}{P\backslash\texttt{A} \xrightarrow{\checkmark}_p \texttt{nil}}$$

$$[\textbf{mand1}] \quad \frac{P \xrightarrow{\checkmark}_p \texttt{nil}}{P \Rightarrow \texttt{A} \xrightarrow{\texttt{A}}_p \checkmark} \qquad\qquad [\textbf{mand2}] \quad \frac{P \xrightarrow{\texttt{A}}_p P_1}{P \Rightarrow \texttt{A} \xrightarrow{\texttt{A}}_p P_1}$$

$$[\textbf{mand3}] \quad \frac{P \xrightarrow{\texttt{B}}_p P_1,\ \texttt{A} \neq \texttt{B}}{P \Rightarrow \texttt{A} \xrightarrow{\texttt{B}}_p P_1 \Rightarrow \texttt{A}}$$

$$\texttt{A},\texttt{B},\texttt{C} \in \mathcal{F},\ a \in \mathcal{F} \cup \{\checkmark\}$$

Fig. 2: $\text{SPLA}^{\mathcal{P}}$ operational semantics.

path and the *pheromones* released by other ants that previously performed that move.

Formally, an Ant Colony Optimization algorithm [26] needs a combinatorial optimization problem to be solved. This problem can be defined as:

Definition 3: A model $P = (\mathbf{S}, \Omega, f)$ of a combinatorial optimization problem consists of:

- a search space $\mathbf{S}$ defined over a finite set of discrete decision variables $X_i, i = 1, \ldots, n$.
- a set Ω of constraints among the variables.
- an objective function $f : \mathbf{S} \to R_0^+$ to be minimised.

The generic variable X_i takes values in $D_i = v_i^1, \ldots, v_i^{|D_i|}$. A feasible solution $s \in \mathbf{S}$ is a complete assignment of values to variables that satisfies all constraints in Ω. A solution $s^* \in \mathbf{S}$ is called a global optimum if and only if $f(s^*) \leq f(s) \ \forall s \in \mathbf{S}$.

Then, from this setup we can generate the *construction graph* $G_C(\mathbf{V}, \mathbf{E})$, where $\mathbf{V}$ is a set of vertices and $\mathbf{E}$ is a set of edges. This graph can be obtained from the set of solution components C in two ways: components may be represented either by vertices or by edges. Artificial ants move from vertex to vertex along the edges of the graph, incrementally building a partial solution.

Additionally, ants deposit a certain amount of pheromone on the components, that is, either on the vertices or on the edges that they traverse. The amount of $\Delta\tau$ pheromone

---

Set parameters;
Initialise pheromone trails;
while *termination criterion not reached* **do**
    | *Construct Ant Solutions*;
    | *Update Pheromones*;
end

Algorithm 1: Ant Colony Optimization algorithm: general scheme

deposited may depend on the quality of the solution found. Subsequent ants use the pheromone information as a guide toward promising regions of the search space.

The ACO general scheme is presented in Algorithm 1. In each iteration, each ant generates a solution. Then, the global state updates the pheromones left by the ants in their solution path. Following there is a more detailed explanation of each step:

Construct Ant Solutions: At each iteration, a set of m ants generates solutions taking elements from a finite set of available solution components $\mathbf{C} = \{c_{ij}\}, i = 1, \ldots, n, j = 1, \ldots, |D_i|$. The construction starts from an empty solution set $s^P = \emptyset$ and, at each step, the ant extends its partial solution adding a feasible solution element from the set $\mathbf{N}(s^P) \subseteq \mathbf{C}$, that is the set of elements of $\mathbf{C}$ that can be added to the partial solution s^P without violating any constraint from Ω.
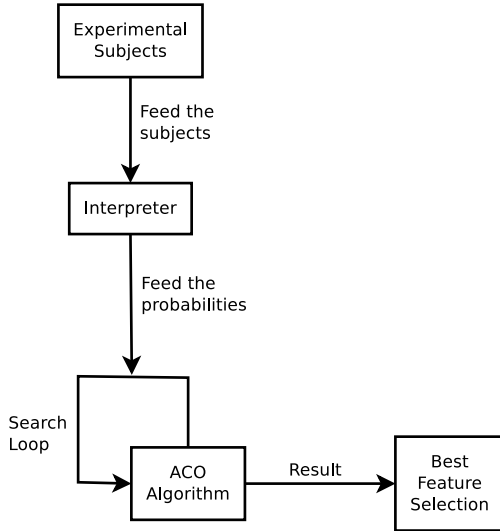
Fig. 3: Schema of the proposed feature selection framework

Fig. 4: Schema of the experiments flow

This process can be seen as a walk on the *construction graph* $G_C(\mathbf{V}, \mathbf{E})$. The choice of a solution component from $\mathbf{N}(s^P)$ is guided by a stochastic mechanism, which is biased by the pheromone associated with each of the elements of $\mathbf{N}(s^P)$. The rule for the stochastic choice of solution components vary across different ACO algorithms but, in all of them are inspired by the behaviour of real ants.

Update Pheromones: The pheromone update aims to increase the pheromone values associated with good or promising solutions, and to decrease those that are associated with bad ones. Usually, this is achieved by decreasing all the pheromone values through pheromone evaporation, and by increasing the pheromone levels associated with a chosen set of good solutions.

III. FEATURE SELECTION FRAMEWORK

In this section, we present our feature selection framework. Its main goal is to find a combination of features that have a high enough probability for a given SPL, that is, a SPLA$^{\mathcal{P}}$ expression. As we have already explained, we decided to rely on evolutionary computation techniques to compute these feature combinations. Specifically, we decided to use an ACO algorithm because we considered it to be the most suitable one for this problem. In future work we will address the use of other evolutionary computation techniques.

Below, we briefly describe the main components of our framework:

- A software product line, it is the system that we are working with. It is represented as a probabilistic algebra expression, specifically, as a SPLA$^{\mathcal{P}}$ expression [14].
- A SPLA$^{\mathcal{P}}$ interpreter that allows us to explore the search space generated by the SPLA$^{\mathcal{P}}$ expression without fully computing it.
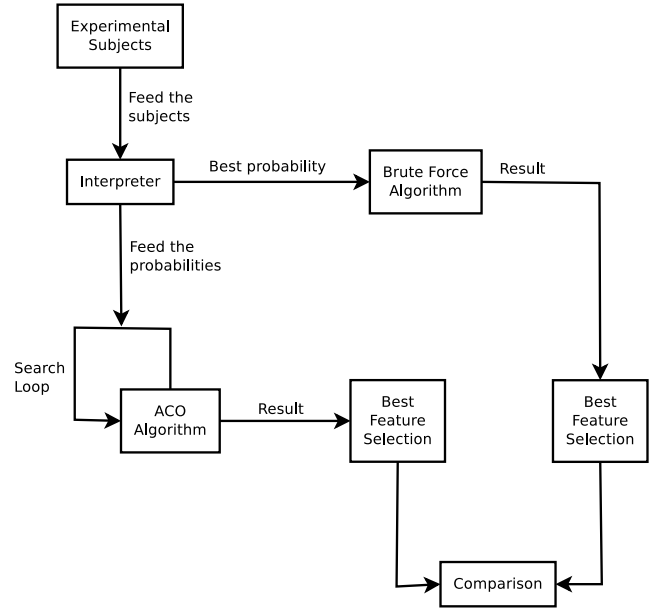- An Ant Colony Optimization algorithm. It leads the search for a feature combination with high probability.

A graphical representation of our framework can be found in Figure 3.

In our framework, we receive a SPL, expressed as a SPLA$^{\mathcal{P}}$ expression, intending to find a set of features that fulfil the requisites of the SPL and at the same time has a high probability. This probability, coming from the SPLA$^{\mathcal{P}}$ expression, usually represents the probability of each feature to be chosen, but it does not have to be limited to that purpose [14]. However, in our scenario, we assume that the probabilities represent the likelihood of each feature to be chosen, because we are looking for the feature combination that has the higher probability to be selected and therefore the one that needs more testing focus when testing the SPL.

Then, with the SPLA$^{\mathcal{P}}$ expression, we interpret it to be able to execute an ACO algorithm over it. As our target is to find a set of features with a high probability, but without having to compute all the probabilities of all the possible combinations of features of the SPLA$^{\mathcal{P}}$ expression, we need to have an interpreter. This interpreter has to, given a feature of the SPLA$^{\mathcal{P}}$ expression, return the probability of that feature, but without computing the full SPLA$^{\mathcal{P}}$ expression tree.

For our ACO algorithm to work, we need to have a combinatorial optimization problem. Then, we need to express our problem as a combinatorial optimization one, in the following way:

- Search space $\mathbf{S}$: it is the full SPLA$^{\mathcal{P}}$ tree, whose decision variables are the feature to choose next.
- Set of constraints Ω: it is composed by:
  - A constraint that states that a valid path should end in a ✓ feature.
  - A constraint that states that a valid path should fulfil the SPLA$^{\mathcal{P}}$ expression constraints.
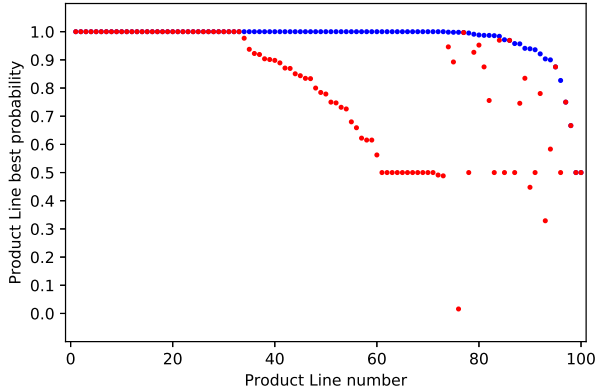- Objective Function f: it is the function assigning to each

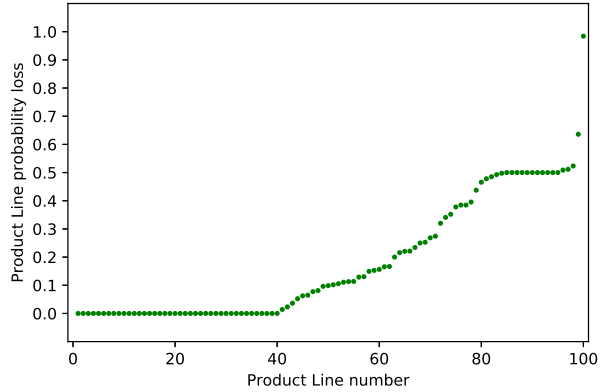Fig. 5: Sorted obtained probabilities (blue = brute force, red = ACO)

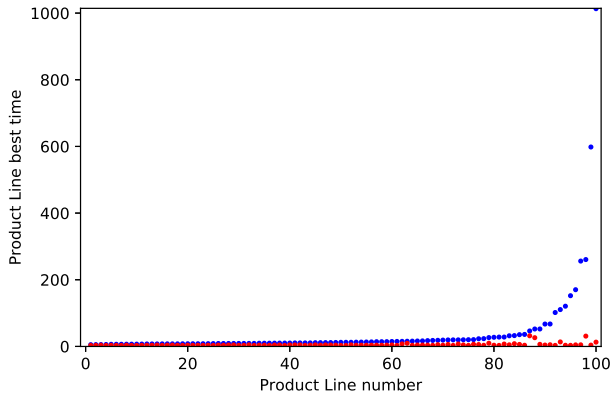Fig. 7: Sorted probability *loss*

Fig. 6: Sorted obtained times (blue = brute force, red = ACO)
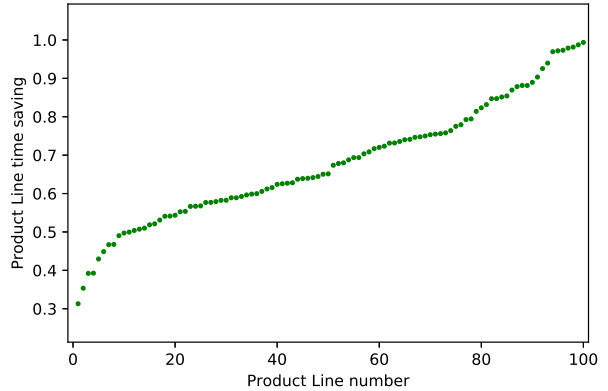

Fig. 8: Sorted time saving

set of features their probability in the SPLA$^{\mathcal{P}}$ expression. In this case, we look to maximize it.

Then, with this well defined combinatorial optimization problem, the ACO algorithm follows the general scheme presented in Algorithm 1. The only modification is the fact that, when exploring the search space, the ants are generating it instead of having a memory variable storing all the information. Thus, the ACO algorithm has to work with the SPLA$^{\mathcal{P}}$ interpreter in order to obtain the distances (in this case, the probabilities).

Another important fact about the ACO algorithm is that the distance between nodes is the probability of choosing the feature associated with the target node. This implies that the total distance travelled by an ant is the product of the probabilities of each step, instead of the sum of the weights as in the classical ACO algorithm.

IV. EXPERIMENTS

In this section, we present an experiment we performed intending to evaluate the suitability of our proposed framework. The schema of the experiment (graphically presented in Figure 4) is very similar to our general framework (see Figure 3). There is, however, a slight difference. To be able to compare the performance of our ACO algorithm with respect to computing the full tree from the SPLA$^{\mathcal{P}}$ expression, we decided to also compute the feature combination with higher probability by a brute force algorithm so that we can compare the performance in time versus the difference in the probabilities obtained. However, although we have computed all the probabilities from the SPLA$^{\mathcal{P}}$ expression, we should not use those probabilities for the ACO algorithm, but instead, it should compute their own probabilities based on which paths the ants take.

For our experiments, we needed four elements:

- A set of SPLA$^{\mathcal{P}}$ expressions as experimental subjects.
- An interpreter of SPLA$^{\mathcal{P}}$ expressions.
- A brute force algorithm to find the set of features with the highest probability.
- An Ant Colony Optimization algorithm implementation[1], modified to be able to work with the SPLA$^{\mathcal{P}}$ probabilistic

process algebra.

The experimental subjects set consists of 100 SPLA$^\mathcal{P}$ expressions automatically generated using the BeTTy tool [47] and stored in an fodaA format in .xml files. *The interpreter* allows us to compute the probabilities of each feature in an ad-hoc way, so we do not fully compute the SPLA$^\mathcal{P}$ expression tree. *The brute force algorithm* is an algorithm that asks the interpreter, for each feature of the SPLA$^\mathcal{P}$ expression, which is its probability. It returns the feature with the highest probability, but at the cost of a longer computation. Therefore, we need to work with small SPLA$^\mathcal{P}$ expressions to end in a reasonable time. *The ACO algorithm* is an ACO algorithm that searches the set of features with maximum probability from the SPLA$^\mathcal{P}$ expression, using the interpreter to compute the probabilities. For our ACO algorithm, we decided to use 5 ants, and perform a maximum of 10 iterations over the main loop. The pheromone evaporation coefficient was 0.4, and the pheromone constant was 1000. Finally, the α and β values used to calculate the attractiveness of each path where 0.5 and 1.2 respectively. The ants and iterations values are so low because we are working with small SPLA$^\mathcal{P}$ expressions (as explained before), and therefore higher values are unnecessary and a waste of resources.

With the interpreter and the ACO algorithm, we were able to implement our framework and test it against the SPLs represented as SPLA$^\mathcal{P}$ expressions coming from the experimental subjects set. To perform interesting experiments, we took the 100 SPLA$^\mathcal{P}$ expressions and computed their *best* selection of features, that is, the one with the highest probability. We computed those selections both with the brute force algorithm and the ACO algorithm and compared the probabilities obtained and the computation times. The code and results of the experiments can be found at https://github.com/Colosu/FSACO.

In Table I we can find the probabilities obtained for each SPLA$^\mathcal{P}$ expression, as well as the computation times. In Figure 5, we can compare the probabilities obtained graphically. The same happens for times in Figure 6. In both figures, the results from the brute force algorithm are in blue and the results from the ACO algorithm are in red. They are sorted to facilitate the display of the results. Finally, in Figure 7 we can observe the sorted probability *losses* obtained, that is, the difference (in percentage) between the probabilities of the best feature combination and the one obtained by our framework. And in Figure 8 we can see the sorted time savings achieved. In mean, we have *loss* an 18.2318% of probability saving at the same time a 67.33% of time. This shows that our method is not only useful to save time. We also obtain good feature combinations: those whose probability is close to the best combination (in terms of probability).

From the results, we can make some interesting remarks. First, it is interesting to see that there are some cases where the ACO algorithm saves more than 97% of the time while losing no probability (that is, giving the feature combination with the highest probability). This is the case of trials number 8 or 34. Another interesting remark is the fact that there are around 40% of the cases where there is no *loss* of probability,

while there is always at least a 30% of time-saving. Moreover, there are only 3 cases (trials 22, 32 and 70) where the *loss* of probability is proportionally higher than the time saving, what can be a result of the randomization factors. However, also due to the randomization factors, there are cases where the time saving comes with a high probability *loss*, being the case of the trial number 32 the clearest case (although the saving in time is still high). In fact, there are only 2 cases (trials 17 and 32) where the *loss* of probability surpass the 60%.

V. THREATS TO VALIDITY

In this section, we briefly discuss some of the possible threats to the results of our experiments validity. Concerning threats to *internal validity*, which consider uncontrolled factors that might be responsible for the obtained results, the main threat is associated with the possible faults in the developed experiment because they could lead to misleading results. To reduce the impact of this threat, we tested our code with carefully constructed examples for which we could manually check the results. Besides, we repeated the experiment with many subjects to diminish the effect of randomization factors.

The main threat to *external validity*, which concerns conditions that allow us to generalize our findings to other situations, is the different possible SPLs to which we could apply our framework. Such a threat cannot be entirely addressed since the population of possible SPLs is unknown, and it is not possible to sample from this (unknown) population. To diminish this risk, we considered different SPLs in the experiments.

Finally, we considered threats to *construct validity*, which are related to the *reality* of our experiments, that is, whether our experiments reflect real-world situations or not. In our work, the main construct threat is what would happen if we use our framework with much more complex SPLs, which is a matter of future work.

VI. CONCLUSIONS

We have proposed a new framework for feature selection in software product lines with probabilities. Our framework strongly relays on Evolutionary Computation techniques to perform feature selection. Specifically, we have used a novel variant of ACO to deal with an *a priory* unknown search space. With this new framework, we can obtain new feature combinations for a given SPL without computing all the possible feature combinations, which is a time-consuming task. Besides, to present the new framework, in this paper, we have reported some of our experiments. Their goal was to show that the *loss* in the feature combination probabilities produced by our framework pays back with the saving of time when computing those combinations.

For future work we have already identified several research directions concerning applicability, scalability, suitability and adaptability of our framework. First, we plan to adapt our framework to perform feature selection in SPLs where instead of probabilities we have costs. Since there are similarities, but also differences, between probabilities and costs we will try to

Trial Number	Brute Force Probability	ACO Probability	Probability Loss	Brute Force Time	ACO Time	Time Saving
1	0.9568	0.7458	0.2204	14.1544	3.6634	0.7411
2	0.9868	0.7558	0.2341	14.7537	4.2998	0.7085
3	1.0	1.0	0.0	7.5902	3.0380	0.5997
4	0.9578	0.5	0.4780	67.1252	4.0392	0.9398
5	1.0	0.8333	0.1666	35.2642	5.9373	0.8316
6	1.0	0.5	0.5	17.3872	3.9101	0.7751
7	1.0	0.7317	0.2682	12.4005	4.4771	0.6389
8	1.0	1.0	0.0	152.1496	3.223	0.9788
9	0.9971	0.9968	0.0003	19.8590	4.3838	0.7792
10	1.0	1.0	0.0	8.7878	3.5475	0.5963
11	0.9952	0.5	0.4976	17.9624	3.1754	0.8232
12	1.0	0.7843	0.2156	52.3167	6.2011	0.8814
13	1.0	0.9226	0.0773	15.1992	7.4489	0.5099
14	1.0	1.0	0.0	6.8014	2.9485	0.5664
15	0.9	0.5833	0.3518	7.3125	3.4301	0.5309
16	1.0	0.75	0.25	9.5622	3.5962	0.6239
17	0.9034	0.3289	0.6359	22.9624	5.5561	0.7580
18	1.0	1.0	0.0	11.3242	3.6242	0.6799
19	1.0	0.9037	0.0962	11.4791	4.2828	0.6269
20	0.9978	0.8925	0.1055	13.5067	3.6242	0.7316
21	1.0	0.8	0.1999	5.7608	3.0709	0.4669
22	1.0	0.6153	0.3846	5.5168	3.5669	0.3534
23	1.0	0.8344	0.1655	26.4979	9.5484	0.6396
24	1.0	1.0	0.0	8.0393	3.3579	0.5823
25	1.0	0.5	0.5	8.0977	3.1421	0.6119
26	1.0	0.6222	0.3777	7.4093	3.9446	0.4676
27	1.0	1.0	0.0	9.6065	3.5982	0.6254
28	1.0	1.0	0.0	12.3441	2.9144	0.7638
29	0.9910	0.9268	0.0647	260.5862	30.8953	0.8814
30	1.0	1.0	0.0	31.9025	4.7426	0.8513
31	1.0	0.9190	0.0809	52.2687	26.1559	0.4995
32	0.9978	0.0158	0.9840	36.01	3.4776	0.9034
33	0.9836	0.9696	0.0141	8.9961	3.3452	0.6281
34	1.0	1.0	0.0	120.6549	3.4089	0.9717
35	1.0	1.0	0.0	15.8293	3.2840	0.7925
36	1.0	0.8695	0.1304	7.9836	3.5726	0.5525
37	1.0	1.0	0.0	5.4251	3.2981	0.3920
38	1.0	1.0	0.0	10.4942	4.4392	0.5769
39	0.9409	0.8346	0.1129	11.1994	4.4170	0.6056
40	1.0	1.0	0.0	67.2651	5.0123	0.9254
41	1.0	0.8982	0.1017	8.8722	4.0693	0.5413
42	0.875	0.875	0.0	8.7718	3.7987	0.5669
43	1.0	1.0	0.0	6.9869	3.1898	0.5434
44	1.0	0.5	0.5	9.7102	3.1239	0.6782
45	1.0	0.4883	0.5116	6.4319	2.8708	0.5536
46	0.9983	0.9461	0.0523	110.5130	13.4314	0.8784
47	0.6666	0.6666	0.0	7.6843	3.3206	0.5678
48	1.0	1.0	0.0	101.6822	3.1101	0.9694
49	1.0	0.5625	0.4375	16.2177	4.0925	0.7476
50	1.0	1.0	0.0	8.5078	3.0497	0.6415

(a) First part.

Trial Number	Brute Force Probability	ACO Probability	Probability Loss	Brute Force Time	ACO Time	Time Saving
51	1.0	0.7258	0.2741	6.6376	3.6591	0.4487
52	1.0	0.8709	0.1290	9.8265	4.0051	0.5924
53	0.9359	0.5	0.4657	12.2820	3.0111	0.7548
54	1.0	0.9768	0.0231	32.4471	8.1153	0.7498
55	1.0	1.0	0.0	6.4097	3.2232	0.4971
56	1.0	1.0	0.0	10.5378	3.1247	0.7034
57	0.5	0.5	0.0	18.9448	3.5220	0.8140
58	1.0	1.0	0.0	27.3563	3.5678	0.8695
59	0.9882	0.9523	0.0362	256.0567	4.7459	0.9814
60	1.0	0.9375	0.0625	19.6477	7.1258	0.6373
61	1.0	0.7472	0.2527	1013.2517	12.8199	0.9873
62	1.0	0.5	0.5	14.2741	3.7750	0.7355
63	1.0	0.6590	0.3409	11.5821	4.0514	0.6502
64	1.0	0.5	0.5	10.0417	2.8096	0.7201
65	1.0	0.5	0.5	6.7181	2.8282	0.5790
66	1.0	1.0	0.0	9.9793	4.1015	0.5889
67	0.8269	0.5	0.3953	8.0312	3.2244	0.5985
68	1.0	0.5	0.5	12.9420	3.5798	0.7233
69	1.0	1.0	0.0	598.2494	3.8589	0.9935
70	1.0	0.4912	0.5087	6.3343	3.2288	0.4902
71	1.0	0.5	0.5	8.2636	3.4499	0.5825
72	1.0	1.0	0.0	27.9893	3.0902	0.8895
73	0.9714	0.5	0.4852	9.0948	4.4819	0.5071
74	1.0	0.5	0.5	23.3201	3.3990	0.8542
75	0.9212	0.7804	0.1528	8.8912	4.2803	0.5185
76	1.0	0.6153	0.3846	8.9465	3.1816	0.6443
77	1.0	0.5	0.5	14.1738	3.4967	0.7532
78	1.0	0.7787	0.2212	10.4757	4.3059	0.5889
79	1.0	1.0	0.0	7.6632	3.6695	0.5211
80	1.0	0.68	0.3199	10.5954	3.6948	0.6512
81	1.0	1.0	0.0	9.1105	2.9731	0.6736
82	1.0	1.0	0.0	7.3558	3.3774	0.5408
83	1.0	1.0	0.0	7.8904	3.0344	0.6154
84	0.9858	0.5	0.4928	170.1517	4.5620	0.9731
85	1.0	1.0	0.0	12.7440	3.4231	0.7313
86	0.75	0.75	0.0	5.2166	2.9751	0.4296
87	1.0	1.0	0.0	11.8614	3.6342	0.6936
88	1.0	1.0	0.0	14.9899	3.0819	0.7943
89	1.0	1.0	0.0	13.2409	3.4413	0.74
90	0.9393	0.4477	0.5233	10.1325	4.2891	0.5766
91	1.0	0.8505	0.1494	18.3256	5.1875	0.7169
92	1.0	0.9012	0.0987	15.5291	9.4323	0.3926
93	0.9690	0.9690	0.0	46.3790	31.8633	0.3129
94	1.0	1.0	0.0	20.1493	3.0774	0.8472
95	1.0	0.8892	0.1107	20.3595	6.3576	0.6877
96	0.5	0.5	0.0	7.6664	3.8037	0.5038
97	0.9870	0.875	0.1134	19.4409	5.9538	0.6937
98	1.0	0.8437	0.1562	16.5696	4.0402	0.7561
99	1.0	1.0	0.0	28.0511	7.1120	0.7464
100	1.0	0.5	0.5	19.6242	3.0028	0.8469

(b) Second part.

TABLE I: Results of the experiments.

incorporate into our framework recent work on formal testing of fuzzy systems [12], [13], where probabilities are *replaced* by confidences, and on testing using Information Theory concepts [37]. Second, concerning scalability, we would like to consider more complex SPLs and check whether our technique scales well. In addition, we would like to use current approaches to mutation testing [15], [20], [22] to efficiently generate and process big amount of mutants representing either non-optimal or faulty selections of features. Concerning suitability, we have two orthogonal lines of work. First, we would like to compare our ACO approach with other metaheuristics such as Bee Swarm [39] and Water Based [46] metaheuristics and Collective Intelligence [24], [25], [43], [44]. Second, we would like to consider SPLs with existing feature selections, produced by an expert, and compare the quality of the existing feature selections and the ones produced by our framework. Concerning adaptability, we would like to assess the useful-

ness of our methodology in other frameworks. In particular, we consider more complicated feature selection frameworks where we have to work with deadlock avoidance/analysis [9]–[11], [19], so that we can scale the feature selection from single systems to entire software families. A second line of work consists in applying our framework to formal models of cloud [6], [7], [16] and distributed [34], [35] systems because they are highly configurable and, therefore, will induce SPLs with many features. Finally, it is interesting the possibility of integration of our feature selection framework to existing tools like ProFeat [17], to represent product lines, PRISM [41], to analyse probabilistic systems, and MEdit4CEP-CPN [8], to represent complex events.

REFERENCES

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2nd edition, 2017.

[2] C. Andrés, C. Camacho, and L. Llana. A formal framework for software product lines. *Information & Software Technology*, 55(11):1925–1947, 2013.

[3] C. Andrés, M. G. Merayo, and M. Núñez. Multi-objective genetic algorithms: Construction and recombination of passive testing properties. In *22nd Int. Conf. on Software Engineering & Knowledge Engineering, SEKE'10*, pages 405–410. Knowledge Systems Institute, 2010.

[4] César Andrés, Carlos Camacho, and Luis Llana. A formal framework for software product lines. *Inf. Softw. Technol.*, 55(11):1925–1947, 2013.

[5] M. Benito-Parejo, I. Medina-Bulo, M. G. Merayo, and M. Núñez. Using genetic algorithms to generate test suites for FSMs. In *15th Int. Work-Conf. on Artificial Neural Networks, IWANN'19, LNCS 11506*, pages 741–752. Springer, 2019.

[6] A. Bernal, M. E. Cambronero, A. Núñez, P. C. Cañizares, and V. Valero. Improving cloud architectures using UML profiles and M2T transformation techniques. *The Journal of Supercomputing*, 75(12):8012–8058, 2019.

[7] A. Bernal, M. E. Cambronero, V. Valero, A. Núñez, and P. C. Cañizares. A framework for modeling cloud infrastructures and user interactions. *IEEE Access*, 7:43269–43285, 2019.

[8] J. Boubeta-Puig, G. Díaz, H. Macià, V. Valero, and G. Ortiz. MEdit4CEP-CPN: An approach for complex event processing modeling by prioritized colored Petri nets. *Information Systems*, 81:267–289, 2019.

[9] M. Bravetti, M. Carbone, and G. Zavattaro. Undecidability of asynchronous session subtyping. *Inf. Comput.*, 256:300–320, 2017.

[10] M. Bravetti, M. Carbone, and G. Zavattaro. On the boundary between decidability and undecidability of asynchronous session subtyping. *Theor. Comput. Sci.*, 722:19–51, 2018.

[11] M. Bravetti and G. Zavattaro. On the expressive power of process interruption and compensation. *Mathematical Structures in Computer Science*, 19(3):565–599, 2009.

[12] I. Calvo, M. G. Merayo, and M. Núñez. A methodology to analyze heart data using fuzzy automata. *Journal of Intelligent & Fuzzy Systems*, 37(6):7389–7399, 2019.

[13] I. Calvo, M. G. Merayo, M. Núñez, and F. Palomo-Lozano. Conformance relations for fuzzy automata. In *15th Int. Work-Conf. on Artificial Neural Networks, IWANN'19, LNCS 11506*, pages 753–765. Springer, 2019.

[14] C. Camacho, L. Llana, A. Núñez, and M. Bravetti. Probabilistic software product lines. *Journal of Logical and Algebraic Methods in Programming*, 107:54 – 78, 2019.

[15] P. C. Cañizares, A. Núñez, and M. G. Merayo. Mutomvo: Mutation testing framework for simulated cloud and HPC environments. *Journal of Systems and Software*, 143:187–207, 2018.

[16] P. C. Cañizares, A. Núñez, J. de Lara, and L. Llana. MT-EA4Cloud: A methodology for testing and optimising energy-aware cloud systems. *Journal of Systems and Software*, 163:110522:1–110522:25, 2020.

[17] P. Chrszon, C. Dubslaff, S. Klüppelholz, and C. Baier. Profeat: feature-oriented engineering for family-based probabilistic model checking. *Formal Aspects of Computing*, 30(1):45–75, 2018.

[18] M. Cordy, P. Heymans, P. Schobbens, A. M. Sharifloo, C. Ghezzi, and A. Legay. Verification for reliable product lines. *CoRR*, abs/1311.1343, 2013.

[19] F. S. de Boer, M. Bravetti, M. D. Lee, and G. Zavattaro. A petri net based modeling of active objects and futures. *Fundam. Inform.*, 159(3):197–256, 2018.

[20] P. Delgado-Pérez and I. Medina-Bulo. Search-based mutant selection for efficient test suite improvement: Evaluation and results. *Information and Software Technology*, 104:130–143, 2018.

[21] P. Delgado-Pérez, I. Medina-Bulo, and M. Núñez. Using evolutionary mutation testing to improve the quality of test suites. In *19th IEEE Congress on Evolutionary Computation, CEC'17*, pages 596–603. IEEE Computer Society, 2017.

[22] P. Delgado-Pérez, Louis M. Rose, and I. Medina-Bulo. Coverage-based quality metric of mutation operators for test suite improvement. *Software Quality Journal*, 27(2):823–859, 2019.

[23] K. Derderian, M. G. Merayo, R. M. Hierons, and M. Núñez. A case study on the use of genetic algorithms to generate test cases for temporal systems. In *11th Int. Conf. on Artificial Neural Networks, IWANN'11, LNCS 6692*, pages 396–403. Springer, 2011.

[24] G. Díaz, H. Macià, V. Valero, J. Boubeta-Puig, and F. Cuartero. An intelligent transportation system to control air pollution and road traffic in cities integrating CEP and colored petri nets. *Neural Computing and Applications*, 32(2):405–426, 2020.

[25] G. Díaz, H. Macià, V. Valero, J. Boubeta-Puig, and G. Ortiz. Facilitating the quantitative analysis of complex events through a computational intelligence model-driven tool. *Scientific Programming*, 2019:2604148:1–2604148:17, 2019.

[26] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006.

[27] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, 2004.

[28] M. Eriksson, J. Börstler, and K. Borg. The PLUSS approach - domain modeling with features, use cases and use case realizations. In *Int. Conf. on Software Product Lines, SPLC'05*, pages 33–44, 2005.

[29] D. E. Goldberg. *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison-Wesley, 1989.

[30] M. L. Griss, J. M. Favaro, and M. D'Alessandro. Integrating feature modeling with the RSEB. In *Int. Conf. on Software Reuse, ICSR'98*, pages 76–85, 1998.

[31] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *22nd ACM SIGSOFT Int. Symposium on Software Testing and Analysis, ISSTA'12*, pages 78–88. ACM Press, 2012.

[32] L. Gutiérrez-Madroñal, A. García-Domínguez, and I. Medina-Bulo. Evolutionary mutation testing for IoT with recorded and generated events. *Software - Practice & Experience*, 49(4):640–672, 2019.

[33] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[34] R. M. Hierons, M. G. Merayo, and M. Núñez. Bounded reordering in the distributed test architecture. *IEEE Transactions on Reliability*, 67(2):522–537, 2018.

[35] R. M. Hierons and M. Núñez. Implementation relations and probabilistic schedulers in the distributed test architecture. *Journal of Systems and Software*, 132:319–335, 2017.

[36] A. Ibias, D. Griñán, and M. Núñez. GPTSG: a Genetic Programming Test Suite Generator using Information Theory measures. In *15th Int. Work-Conf. on Artificial Neural Networks, IWANN'19, LNCS 11506*, pages 716–728. Springer, 2019.

[37] A. Ibias, R. M. Hierons, and M. Núñez. Using Squeeziness to test component-based systems defined as Finite State Machines. *Information & Software Technology*, 112:132–147, 2019.

[38] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-Oriented Domain Analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, 1990.

[39] D. Karaboga and B. Akay. A survey: algorithms simulating bee swarm intelligence. *Artificial Intelligence Review*, 31(1):61, Oct 2009.

[40] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[41] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Int. Conf. on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[42] J. D. McGregor. Testing a software product line. In *Testing Techniques in Software Engineering, Pernambuco Summer School on Software Engineering, PSSE'07*, pages 104–140, 2007.

[43] V. D. Nguyen, H. B. Truong, M. G. Merayo, and N. T. Nguyen. An overview on consensus-based approaches to processing collective inconsistency and knowledge integration. *WIREs Data Mining and Knowledge Discovery*, 9(4):1311:1–1311:9, 2019.

[44] V. D. Nguyen, H. B. Truong, M. G. Merayo, and N. T. Nguyen. Toward evaluating the level of crowd wisdom using interval estimates. *Journal of Intelligent & Fuzzy Systems*, 37(6):7279–7289, 2019.

[45] A. Núñez, M. G. Merayo, R. M. Hierons, and M. Núñez. Using genetic algorithms to generate test sequences for complex timed systems. *Soft Computing*, 17(2):301–315, 2013.

[46] P. Rabanal, I. Rodríguez, and F. Rubio. Applications of river formation dynamics. *Journal of Computational Science*, 22:26–35, 2017.

[47] S. Segura, J. A. Galindo, D. Benavides, J. A. Parejo, and A. Ruiz-Cortés. Betty: Benchmarking and testing on the automated analysis of feature models. In *6th Int. Workshop on Variability Modeling of Software-Intensive Systems, VaMoS'12*, pages 63–71, 2012.

[48] A. Windisch, S. Wappler, and J. Wegener. Applying particle swarm optimization to software testing. In *9th Genetic and Evolutionary Computation Conference, GECCO'07*, pages 1121–1128. ACM Press, 2007.