Using Ant Colony Optimisation to Select Features having Associated Costs*

Alfredo Ibias[1 [0000-0002-3122-4272]], Luis Llana[1 [0000-0003-1962-1504]], and Manuel Núñez[1 [0000-0001-9808-6401]]

Universidad Complutense de Madrid, Madrid 28040, Spain
{aibias,llana,manuelnu}@ucm.com

Abstract. Software Product Lines (SPLs) strongly facilitate the automation of software development processes. They combine features to create programs (called *products*) that fulfil certain properties. Testing SPLs is an intensive process where choosing the proper products to include in the testing process can be a critical task. In fact, selecting the *best* combination of features from an SPL is a complex problem that is frequently addressed in the literature. In this paper we use evolutionary algorithms to find a combination of features with low testing cost that include a target feature, to facilitate the integration testing of such feature. Specifically, we use an Ant Colony Optimisation algorithm to find one of the *cheapest* (in terms of testing) combination of features that contains a specific feature. Our results show that our framework overcomes the limitations of both brute force and random search algorithms.

Keywords: Software Product Lines · Integration Testing · Ant Colony Optimisation · Feature Selection.

1 Introduction

Software Product Lines (SPLs) define generic software products, enabling mass customisation. Generally speaking, SPLs provide a systematic and disciplined approach to developing software. SPLs encode a set of similar (software) systems that can be constructed from a specific set of features. These features can be combined according to some specific rules defining which products (that is, which combinations of features) are valid. In this paper we use FODA [22] to represent SPLs. In order to formally reason about FODA diagrams, it is important to have a formal framework to represent FODA diagrams. In previous work, we introduced SPLA [1], an algebra that can provide a precise semantics to these diagrams. The original framework was extended to manage an important aspect of features: their costs. This is captured in the process algebra SPLA-CRIS [4]. In this work, these costs will represent the cost of testing a specific feature of the product. Testing SPLs is fundamental to ensure the quality and

* This work has been supported by the Spanish MINECO/FEDER project FAME (RTI2018-093608-B-C31); the Region of Madrid project FORTE-CM (S2018/TCS-4314) co-funded by EIE Funds of the European Union; and the Santander - Complutense University of Madrid (grant number CT63/19-CT64/19).

reliability of the products generated by them. When testing SPLs [26], it is crucial to distribute the testing resources between the different features of the line in a smart way. One way of distributing such resources is based on the probability of each feature being requested [18]. However, if we do not have such probabilities, we can consider the costs of testing each feature. The idea is that the products with the minimum cost will be easier to test and, therefore, will consume less resources. This situation is ideal when testing the integration of a specific feature into the SPL. For example, if we add a new feature to an existing SPL and we want to test that its integration with the other features does not produce any errors, then it is useful to have a product with lower testing cost because the integration testing process will be faster and/or cheaper.

We are going to focus on the problem of *Integration Testing of Software Components* [21]. Actually, software components can be seen as the features of an SPL. In fact, integration testing within an SPL has gained attention from researchers [7,29,33]. One important aspect of integration testing is its cost: although testing each variant of an SPL may be feasible, it is impossible to independently test all possible (maybe redundant) products [24]. In our approach, we are interested in getting the product that includes a particular feature having the smallest testing cost. Note that the order of the features may be relevant in the complexity and costs of the testing process [34].

In general, testing cost can refer to multiple concepts: from actual monetary cost of testing the integration of the feature into the product, to the necessary time to test such integration, passing through the amount of resources needed to test that integration. In our framework, we only need to know that such cost exists and that it represents the same along all individual costs of the same SPL. Therefore, along this paper we will be talking about testing cost in a broad sense and we will try to minimise it. Finally, regarding the origin of such costs, we will assume that they are provided together with the SPL. Ideally, such costs would be obtained through estimation, approximation or empirical methods and added to the SPL before using the solution presented in this paper.

It is important to clarify what we mean by computing a cheap (or expensive) product of an SPL. In our context, a cheap product is a product that has a low total testing cost compared to the cost of other products. For example, if the testing cost represents the estimated time needed to test such product, then a cheap product would be one whose aggregated time to test it is low compared with other products of the SPL (e.g. hours vs. weeks). Note that cheaper to test products will not necessarily be the ones with a smaller number of features.

Finally, we want to clarify that testing cost is not a proxy for fault detection effectiveness. We are not looking for the product that will arise more faults, but for the one that will be cheaper to test. This is so because our solution looks to fill a very specific need: we have a feature to add to an SPL and we want to cheaply test its integration into the SPL. The goal is not to find all the faults in the introduced feature, but instead ensure that it can be included into products of the SPL. This is specially useful when one has an SPL with hundreds or thousands of features and there is not enough time or resources to test all the possible combinations. Therefore, it is useful to test that the feature can be included into products and that there are no errors when used in combination with other features, what can be tested using any product. One example of such

situation appears when adding a new database to a server SPL. It is necessary to test that the added database is correctly integrated with the other features of the SPL, but the tester only needs to test the integration in one product because all the productions might have the same integration faults.

In this paper we apply Ant Colony Optimisation algorithm (ACO) [9] to select a combination of features from an SPL with the minimum testing cost that contains a given feature. This combination will be later used to test the integration of such feature into the SPL. To the best of our knowledge, this is the first attempt to develop an efficient solution to this problem if we rely on a formal approach (in our case, a process algebra). To develop this algorithm we modify, enhance and extend our recently developed framework [18] so that we select feature combinations with low testing cost from SPLs including testing costs information, and so that we have the requirement of including a given feature in the generated product.

In order to evaluate the quality of the solutions obtained by our ACO-approach, we compare our framework with a brute force algorithm (computing all the combinations of features and choosing one with the lowest cost) and a random algorithm (randomly choosing features but such that they conform a valid product). We could not compare our algorithm to other alternatives as there were no previous proposals addressing our specific scenario. Our framework takes significantly less time to compute a solution than the brute force algorithm (around a 99% saving), while obtaining total testing costs that are not much higher (around a 25% increase). It also gets solutions with lower testing cost than the ones obtained by the random algorithm (around a 15% cheaper). In order to properly compare our ACO and the random approach, we allow the random approach to run an equal amount of time as the ACO one. In conclusion, our approach represents a preferable choice to these two alternatives.

The rest of the paper is organised as follows. In Section 2 we review related work. In Section 3 we present background concepts that we use in our paper. In Section 4 we introduce our feature selection framework. In Section 5 we present our experiments and discuss the results. In Section 6 we briefly review some threats to the validity of our results. In Section 7 we discuss some considerations concerning the different choices that we took when defining our algorithm. Finally, in Section 8, we give conclusions and outline some directions for future work.

2 Related work

In this section we review previous work related to the research presented in this paper.

We have chosen FODA [22] to represent SPLs but there are other alternative approaches such as RSEB [12] and PLUSS [10]. We think that FODA represents several advantages: it is widely used and, more important, it is based on graphic models.

We are aware that we cannot compute the best, according to a given criteria, combination of features due to the combinatorial nature of the problem. In fact, we performed a small experiment to show that this is the case also in our framework. Therefore, we have to rely on an heuristic approach. Our previous work on applying heuristic approaches to testing [17, 19, 20] showed that Swarm Intelligence [36] was very suitable. Among the different approaches to implement a swarm, in this paper we have decided

to consider the Ant Colony Optimisation algorithm (ACO) [9] because it allowed us to build on top of previous work, facilitating the implementation of the approach. ACO is inspired by the behaviour of real ant colonies in nature and has been successfully used in computationally hard classical optimisation problems such as the travelling salesman problem but, to the best of our knowledge, the research presented in this paper is a novel application of ACO. Although we have used ACO, other alternative approaches in the broad field of *evolutionary algorithms* could have been selected. Evolutionary algorithms are a family of meta-heuristics that base its intelligent behaviour in the evolution of its population. Some approaches in the broad field of Artificial Intelligence consider the combination of many individuals, usually with limited intelligence, that work as a collective to either reach a goal or find a *good enough* solution to a certain problem. In particular, there are several applications of these algorithms in testing [3, 5, 30].

We have used an evolutionary computation approach to find cheap to test products but a framework supporting constraint propagation could be used. In this case, we could rely on tools like FaMa [2] and FeatureIDE [35]. However, we prefer to use the combination of a process algebra and an evolutionary computation technique because they allow us to work with a precise semantic description of each product, facilitating the task of deciding the equivalence, up to a certain criterion, of different products.

There exist evolutionary approaches for test case selection and prioritisation in SPLs [13, 25]. Despite working on testing, these solutions cannot be easily adapted to cope with our problem because we do not select/generate test cases: we select a set of features such that testing the resulting product is as cheap as possible.

Finally, more related to our work, there are evolutionary computation approaches to select features. A study [31] showed that the *Indicator-Based Evolutionary Algorithm* (IBEA) was better than other evolutionary approaches dealing with high complexity in the decision objective spaces. We cannot use this algorithm to solve our problem because IBEA strongly depends on user preferences (we do not have them). In addition, it seems like this algorithm performs better in a multi-objective optimisation problem: we think that a simpler approach, like ours, might work better in our single-objective optimisation problem but further experiments are needed to support this claim. Finally, another important difference is that they define the set of rules from the SPL as an objective of the optimisation problem because their solution can create non-valid products. In our case, we use a process algebra as the search space to ensure the correctness of the generated products. Another related study [14] proposed the SIP method, which improved previous proposals beating even the IBEA algorithm. The approach mainly focused on enhancing the search through a novel representation that hard-codified some constraints and through optimising first the constraints related to the generation of valid products. They also used their approach over real-world SPLs. However, this approach has the same concerns than the previous one: its problem is based on user preferences, it is focused on multi-objective optimisation, and, furthermore, it can produce non-valid products. All these differences make hard to adapt this kind of algorithms to our problem, as they rely on some assumptions that we do not consider and they can generate non-valid products that our approach cannot generate.

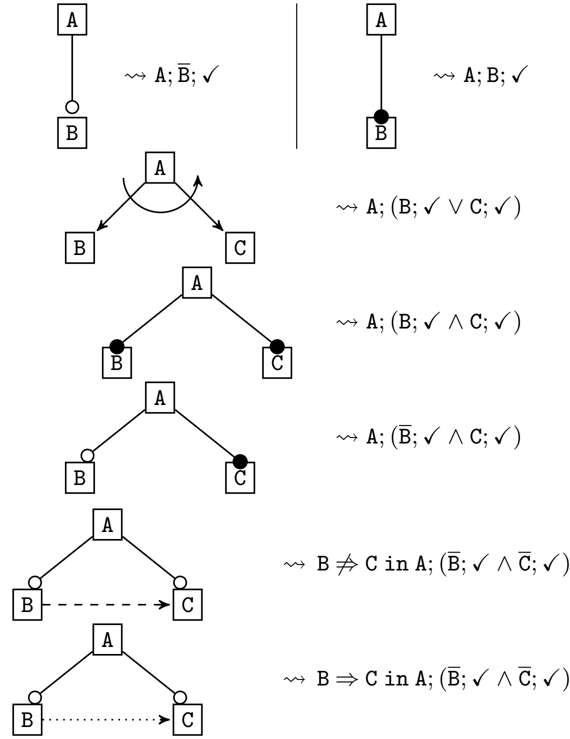

Fig. 1: Translation of FODA Diagrams into SPLA.

3 Preliminaries

In this section we present notation and introduce concepts related to the main two lines that we use in this paper: specification of Software Product Lines with costs and the Ant Colony Optimisation algorithm.

3.1 SPLA-CRIS: SPLs with costs

In this section we briefly review the formal language SPLA-CRIS. The interested reader is referred to the original work [4] for more details.

Definition 1. *We will assume that we have a finite set of features $\mathcal{F}$ and we will use* $\mathtt{A}$*,* $\mathtt{B}$*,* $\mathtt{C}\ldots$ *to denote single features. A* Software Product Line *is a term generated by the following Extended BNF-like expression:*

$$\begin{array}{rl} P ::= & \checkmark \mid \mathtt{nil} \mid \mathtt{A};P \mid \overline{\mathtt{A}};P \mid P \vee Q \mid P \wedge Q \\ & \mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ P \mid \mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ P \end{array}$$

where $\mathtt{A}, \mathtt{B} \in \mathcal{F}$*. We denote the set of terms of this algebra by* SPLA.

Next we describe the operators of the algebra. The term $\mathtt{nil}$ *represents an SPL with no products, while* $\checkmark$ *is an SPL that has only the empty product; they are the terminal elements of the syntax. Then the we have the mandatory prefix operator* $\mathtt{A};P$ *(feature* $\mathtt{A}$ *is mandatory) and the optional prefix operator* $\overline{\mathtt{A}};P$ *(*$\mathtt{A}$ *is optional). The binary operator* $P \vee Q$ *represents the* choose-one. *The binary operator* $P \wedge Q$ *represents the* conjunction *operator. These operators are associative and commutative, so they can be extended as* n*-ary operators. The operator* $\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ P$ *represents the* require *constraint. The operator* $\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ P$ *represents the* exclusion *constraint. Figure 1 shows the relation between these operators and FODA diagrams.*

We can define an operational semantics. *Given* $\mathtt{A} \in \mathcal{F} \cup \{\checkmark\}$*, we will write* $P \xrightarrow{\mathtt{A}} Q$ *if we can evolve from* P *to* Q *using the defined operational rules. It is important to remark that* $\checkmark$ *is not a feature and, as such, it is not included in the product. This semantics is given as a set of SOS rules and the interested reader can find them, as well as detailed explanations, in our previous work [1, 4].*

Single transitions can be sequentially executed to produce traces. *We use* ϵ *to denote an empty trace and consider the usual concatenation operator* $s_1 \cdot s_2$*. Abusing the notation, we will write* $\mathtt{A} \in s$ *is* $\mathtt{A}$ *appears in* s*. Traces ending with* $\checkmark$*, that we call* successful*, are the only ones associated with valid products. It is irrelevant the order in which the features of a trace are obtained. Given a successful trace* s*,* $[s]$ *denotes the set obtained from the elements of* s*.*

Finally, given $P \in \mathtt{SPLA}$*, we define* the products of P*, denoted by* $\mathtt{prod}\,(P)$*, as* $\mathtt{prod}\,(P) = \{[s] \mid s \in \mathtt{tr}(P)\}$. □

In order to define a cost model, we will have a cost function such that given a sequence of features (representing the part of the product that we have defined so far) and a single feature (representing the new feature that we would like to add), returns the cost of testing this new feature in the given product. This cost can represent either time and/or resources needed to perform the (integration) testing of this new feature, given the previous ones. In our framework, we assume that costs can be represented by natural numbers. Sometimes, we will not be able to compute the testing cost of integrating a new feature with the ones already chosen. For instance, if the new one is incompatible with the existing features or there are missing dependencies. Therefore, we extend the set of costs with a new symbol $\bot$ to represent *indefiniteness*.

Definition 2. *The set of costs is given by* $\boldsymbol{N}_\bot = \boldsymbol{N} \cup \{\bot\}$*. We extend arithmetic operations in the expected way: for any* $x \in \boldsymbol{N}_\bot$ *we have* $x + \bot = \bot + x = \bot$ *and* $x \leq \bot$*.*

A cost function *is a function* $\mathtt{c} : \mathcal{F}^* \times \mathcal{F} \mapsto \boldsymbol{N}_\bot$*.* □

In order to compute the cost associated with a product we need to extend the operational semantics (see our previous work [4] for a complete definition). Intuitively, let $P \in \mathtt{SPLA}$ be a process, $\mathtt{c}$ be a cost function and s be a successful trace of P. We denote by $\mathtt{tc}(P, s)$ the cost associated with the set of features included in s according to $\mathtt{c}$.

Finally, let us remind that the position of the features in the trace is not relevant to define a product although it may have an impact in its costs. Therefore, different traces

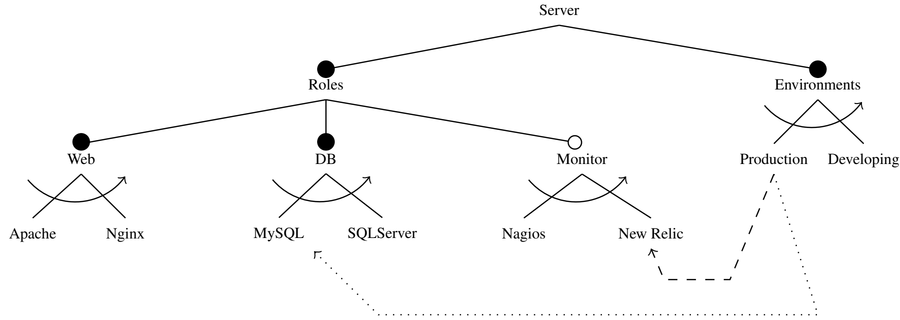

Fig. 2: FODA App Server feature diagram.

```
Server =
  P ⇒ MS in (
    P ⇏ NE in (
        S; (
          R; (
            W; (AP; ✓ ∨ NG; ✓)
            ∧
            D; (MS; ✓ ∨ SS; ✓)
            ∧
            M̄; (NA; ✓ ∨ NR; ✓)
          )
          ∧
          E; (P; ✓ ∨ Dv; ✓)
        )
      )
    )
  )
```

P	itc(P)
{S, R, E, W, D, P, AP, MS}	2.10
{S, R, E, W, D, P, NG, MS}	2.40
{S, R, E, W, D, Dv, AP, MS}	1.10
{S, R, E, W, D, Dv, NG, MS}	1.30
{S, R, E, W, D, Dv, AP, SS}	1.20
{S, R, E, W, D, Dv, NG, SS}	1.00
{S, R, E, W, D, P, M, NA, AP, MS}	2.50
{S, R, E, W, D, Dv, M, NA, AP, MS}	2.30
{S, R, E, W, D, Dv, M, NE, AP, MS}	2.20
{S, R, E, W, D, Dv, M, NA, AP, SS}	2.20
{S, R, E, W, D, Dv, M, NE, AP, SS}	2.10
{S, R, E, W, D, P, M, NA, NG, MS}	2.80
{S, R, E, W, D, Dv, M, NA, NG, MS}	2, 40
{S, R, E, W, D, Dv, M, NE, NG, MS}	2.30
{S, R, E, W, D, Dv, M, NA, NG, SS}	2.20
{S, R, E, W, D, Dv, M, NE, NG, SS}	2.10

Legend:
S : Server
R : Roles
E : Environments
P : Production
Dv : Developing
W : Web server
D : Database Server
M : Monitoring service
AP : Apache
NG : Nginx
MS : MySQL
SS : SQLServer
NA : Nagios
NE : New Relic

Fig. 3: SPLA term.

can produce the same product but with different costs. As a consequence, we need to consider a set of costs for each product, because a product will be *equivalent* to a set of sequences.

Definition 3. *Let* $\mathtt{c}$ *be a cost function. We consider the function* $\mathtt{c}_{\mathtt{SPLA}} : \mathtt{SPLA} \times \mathcal{P}(\mathcal{F}^*) \mapsto \mathcal{P}(\boldsymbol{N}_\perp)$ *defined as follows:*

$$\mathtt{c}_{\mathtt{SPLA}}(P, p) = \{\mathtt{tc}(P, s) \in \boldsymbol{N}_\perp | \exists s \text{ trace of } P : [s] = p\}$$

□

Example Let us illustrate the previous definitions with an example. Let us consider a *Server* consisting of a Web Server and a Database. There are two possible environments for the running server: the *production* environment and the *developing* environment. There are two possibilities for the database: MySQL or SQLServer. For the Web server we can use either Apache Web Server or Nginx. There are also two restrictions in the

case of the Production environment: First, the use of the New Relic monitor system is forbidden. Second, the use of MySQL is mandatory. Figure 2 show the FODA diagram corresponding to this description. This FODA diagram is translated to the SPLA term in Figure 3 (left) to handle the system formally. The Integration Test costs appears in the centre of Figure 3. Formally, the cost function is defined as follows: for $s \in \mathcal{F}^*$ and $\mathtt{A} \in \mathcal{F}$, $\mathtt{c}(s, \mathtt{A}) = \mathsf{itc}([s\mathtt{A}])$ if the product $[s\mathtt{A}]$ is listed in table and $\mathtt{c}(s, \mathtt{A}) = 0$ otherwise.

3.2 Ant Colony Optimisation

The Ant Colony Optimisation algorithm (ACO) [9] is a well-known algorithm in the evolutionary algorithms field. It is a distributed algorithm to explore a graph-like search space associated with a combinatorial optimisation problem. It consists of a set of *ants*, which are the agents that explore the search space. Each ant looks 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 path and the *pheromones* released by other ants that previously performed that move.

Definition 4. *A* model *P of a combinatorial optimisation problem is a tuple $(\mathbf{S}, \Omega, f)$, where $\mathbf{S}$ is a search space defined over a finite set $X_1, \ldots, X_n$ of discrete decision variables, Ω is a set of constraints over the variables, and $f : \mathbf{S} \to R_0^+$ is the objective function to be minimised.*

Each 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 such that all the constraints in Ω are satisfied. A feasible solution $s^* \in \mathbf{S}$ is called a* global optimum *if and only if for all $s \in \mathbf{S}$ we have $f(s^*) \leq f(s)$.* □

Once we have a model of the problem that we would like to solve, we can generate a *construction graph*. Artificial ants move from vertex to vertex along the edges of this graph, incrementally building a partial solution. During this traversal of the graph, the ants deposit a certain amount of pheromone on the edges that they traverse. The amount of pheromone deposited by each artificial ant usually depends on the *quality* of the solution reached after that specific traversal. The idea underlying ACO and the simulation of pheromone is that other ants will use the information concerning the concentration of pheromone as a hint to further explore promising regions of the search space.

The *ACO general scheme* proceeds as follows. After a preliminary step, where the main parameters and the pheromone trails are initialised, we have a main loop that iterates until we reach the termination criterion. This criterion may be based on the numbers of iterations of the loop or on the quality of the obtained solution. In each iteration of the loop, each ant generates a solution. Then, the global state updates the pheromones left by the ants in their solution path. This task consists of two main consecutive steps.

First step of the loop: Construct ant solutions. In each iteration, m ants generate solutions from a finite set of available solution components C. The construction starts from an empty solution set $s^P = \emptyset$ and, in each step, the ant extends its partial solution by adding a feasible solution element from the set of elements of $\mathbf{C}$ that can be added

to the partial solution s^P without violating any constraint in Ω. The choice of a solution component from this set is guided by a stochastic mechanism, which is biased by the pheromone associated with each of the elements in it. The rule for the stochastic choice of solution components varies across different ACO algorithms but they are always inspired by the behaviour of real ants. This process can be seen as a traversal of the *construction graph*.

Second step of the loop: Update pheromones. The pheromone update aims to increase the pheromone values associated with good or promising solutions and, in turn, decrease those 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.

4 ACO for feature selection taking into account testing costs

Our feature selection framework finds, for a given `SPL` and a selected feature, a combination of features that contains said feature and such that the cost (in time and/or resources) of testing the generated product is as low as possible. We will consider that the `SPL` is formally defined as an SPLA-CRIS term. We use an ACO algorithm because it is the most suitable one for this problem. A comprehensive discussion about this choice can be found in Section 7. Next, we briefly describe the main components of our framework:

- An `SPL` represented as an SPLA-CRIS expression.
- An SPLA-CRIS interpreter that allows us to explore the search space generated by the SPLA-CRIS expression without fully computing it.
- An ACO to lead the search for a feature combination with low cost.

We combine these three components as follows. We consider an SPLA-CRIS expression and derive the structure needed to execute our ACO over it with the goal of finding a *cheap* to test product. However, we cannot compute the testing cost of all the possible combinations of features of the SPLA-CRIS expression. We will rely on an interpreter to compute the added testing cost after adding a new feature to the current selection, but without constructing the full SPLA-CRIS expression tree.

As usual, our ACO needs to have a representation of our setting as a combinatorial optimisation problem. We will define this problem as follows:

- Search space $\mathbf{S}$. This is the full SPLA-CRIS tree. In addition, the associated decision variables are associated to the feature that we have to choose next.
- Set of constraints Ω. We have three constraints.
  - A constraint stating that the last symbol of a valid path must be $\checkmark$. Remind that this is the special symbol that we use to denote successful termination, that is, the last symbol of a successful trace.
  - A constraint stating that a valid feature combination should contain the previously selected feature.
  - A constraint stating that a valid path can be generated by the definition of the SPLA-CRIS expression that we are considering.

- Objective function f. This function assigns its cost to each set of features that can be produced from the SPLA-CRIS expression. The goal of our ACO is to minimise the value of this function.

Once we have our problem redefined as a combinatorial optimisation problem, our ACO follows the general scheme presented in Section 3.2. The only adaption with respect to this general scheme is that our ants generate *on the fly* the search space while exploring it, instead of having all the information stored beforehand. Thus, our ACO has to work together with our SPLA-CRIS interpreter in order to obtain the associated costs.

It is important to note that our algorithm does not use any additional heuristic optimisation. In the literature there are some common heuristics, like removing mandatory features (i.e. computing atomic sets), that are usually used to simplify the problem at hand. In our case, as the goal is to have a lower testing cost, we cannot consider such heuristic optimisation as they would modify the obtained testing costs. For example, in the case of removing the mandatory features, that heuristic would produce testing costs that do not consider the additional testing costs that each mandatory feature would add with each added feature, costs that are not constant neither uniform between different features.

5 Experimental Results

In order to evaluate the usefulness of our ACO to find *cheap* (in terms of testing) combinations of features, according to a certain set of constraints defined by the corresponding SPL, we decided to initially compare it with a *brute force* algorithm. The brute force algorithm will effectively compute a feature combination with the lowest testing cost at the expense of a long execution time. In contrast, we will show that our framework can give feature combinations with slightly higher testing costs but having (much) shorter execution times.

We set our ACO algorithm with the following parameters:

- Number of ants: 10.
- Number of maximum iterations: 100.
- Pheromone constant: 1000.
- Pheromone evaporation coefficient: 0.4.
- α coefficient: 0.5.
- β coefficient: 1.2.

These parameters are typical parameters in the literature and they worked very well in our previous work [18]. Moreover, we did small experiments to tune the parameters and none of them show better performance than these ones.

For our experiments, we used 75 SPLA-CRIS expressions with between 10 and 85 features. These SPLA-CRIS expressions were generated using previous work with SPLA-CRIS [4], automatically generating them using the BeTTy tool [32] and storing them in an fodaA format in .xml files. The costs in these expressions are also automatically generated, and thus we consider that they represent the additional testing costs that a feature will add to the product if included in it.

Trial Number	Brute Force Cost	ACO Cost	Cost Increase	Brute Force Time	ACO Time	Time Saving
1	27	36	33.33%	1.1713	4.5798	-291.01%
2	18	27	53.33%	6.5209	8.9389	-37.08%
3	36	45	25.00%	20.5985	9.8473	52.19%
4	63	72	14.29%	4,434.4519	14.8687	99.66%
Average	**36**	**45**	**25.42%**	**1,115.6856**	**9.5587**	**99.14%**

Table 1: Comparing our approach and brute force (time is measured in seconds).

In our first experiment we evaluated these expressions through our SPLA-CRIS interpreter. Using this interpreter, we executed a brute force algorithm to compute all the possible feature combinations as well as their costs. We also executed our ACO algorithm using the SPLA-CRIS interpreter to obtain a feature combination with low cost. Due to the randomisation involved in the ACO algorithm, we executed both algorithms 15 times for each SPLA-CRIS expression and measured the mean of the results of all the computations. Unfortunately, after running during 20 hours the brute force algorithm was able to compute the solution only for four expressions (note that the longest time used by our ACO was less than 15 seconds). In Table 1 we compare the cost and computation time for these expressions.

As expected, the brute force algorithm was unable to compute, in a reasonable time, the best feature selection for most of the experiments (in fact, it was only able to compute it for the smaller expressions, the ones with less than 13 features) due to the combinatorial explosion underlying feature selection, aggravated with minimising the cost. This leaves us with only four values to compare our ACO with the brute force algorithm. In this comparison we can see that our algorithm obtains, on average, a solution that it is 25.42% more expensive than the best features combination (computed by the brute force algorithm). In contrast, it needs on average 99.14% less time to produce this solution.

Here, it is important to note that for the simplest cases, the brute force algorithm needs less time than our ACO algorithm. The reason is that the expressions are so simple that our ACO algorithm is overpowered for this task. That means that, as the expression is so small, brute force computes all the combinations quickly (because there are so few) while the ACO algorithm not only has to explore the expression, but it also needs to achieve convergence (what will take a while due to the required iterations). However, as we increase the complexity of the expressions, the brute force algorithm quickly raises its execution time a lot (due to its exponential nature), while our ACO algorithm keeps its execution time in a reasonable value.

The comparison with the brute force algorithm leaves us with so few results that we decided to perform a second experiment and compare our framework with a random algorithm. This random algorithm will give us the feature combination with lowest costs of a set of randomly generated feature combinations that represent valid products. The number of feature combinations on this set of randomly generated feature combinations

Trial Number	Random Cost	ACO Cost	Cost Saving (%)
1	52.2	52.2	0.00
2	36.0	34.8	3.33
3	50.4	50.4	0.00
4	73.8	73.8	0.00
5	63.6	63.6	0.00
6	70.2	69.0	1.71
7	63.6	61.2	3.77
8	73.2	70.8	3.28
9	67.8	67.2	0.88
10	75.0	70.8	5.60
11	63.6	63.6	0.00
12	62.4	57.0	8.65
13	91.8	87.0	5.23
14	81.6	77.4	5.15
15	70.8	66.0	6.78
16	58.2	54.0	7.22
17	83.4	79.2	5.04
18	99.6	79.8	19.88
19	78.0	76.8	1.54
20	99.0	81.0	18.18
21	80.4	79.2	1.49
22	111.0	96.6	12.97
23	70.8	67.8	4.24
24	96.0	80.4	16.25
25	76.8	69.0	10.16
26	131.4	102.6	21.92
27	120.6	98.4	18.41
28	147.6	115.8	21.54
29	109.2	94.8	13.19
30	115.8	102.0	11.92
31	161.4	128.4	20.45
32	114.6	82.8	27.75
33	130.2	117.0	10.14
34	157.2	148.8	5.34
35	78.0	70.2	10.00
36	93.0	78.0	16.13
37	100.8	97.2	3.57
38	149.4	133.2	10.84
39	122.4	101.4	17.16
40	166.8	142.2	14.75
41	147.0	138.0	6.12
42	159.0	146.4	7.92
43	115.2	89.4	22.40
44	148.8	135.6	8.87
45	168.0	132.0	21.43
46	156.0	134.4	13.85
47	118.2	98.4	16.75
48	189.6	144.6	23.73
49	175.8	168.0	4.44
50	201.6	175.8	12.8
51	221.4	159.6	27.91
52	136.8	120.6	11.84
53	176.4	142.8	19.05
54	151.2	123.0	18.65
55	142.2	96.0	32.49
56	208.8	176.4	15.52
57	166.8	139.8	16.19
58	145.2	105.0	27.69
59	166.8	135.0	19.06
60	175.2	135.0	22.95
61	178.2	139.2	21.89
62	185.4	167.4	9.71
63	199.2	171.0	14.16
64	254.4	171.6	32.55
65	168.6	121.2	28.11
66	173.4	146.4	15.57
67	238.8	187.8	21.36
68	210.0	191.4	8.86
69	226.2	129.0	42.97
70	201.0	133.2	33.73
71	209.4	163.8	21.78
72	309.6	196.8	36.43
73	340.8	207.0	39.26
74	285.6	150.6	47.27
75	197.4	143.4	27.36

Table 2: Results of the experiment comparing with respect to random.

will depend on how much time the algorithm is running. In our experiment, we first run the ACO algorithm and then we run the random algorithm until it overcomes the execution time the ACO algorithm needed. This way, the random algorithm always has the same (or more) time to execute as our ACO and we compare the algorithms performance, that is, the feature combination costs obtained.

We started with the same set of 75 SPLA-CRIS expressions and evaluated them using our SPLA-CRIS interpreter. For each SPLA-CRIS expression, we also used this interpreter to execute 15 times both our ACO algorithm and the random algorithm. We computed mean costs and compared them (see Table 2).

In order to present an easy visualisation of all the results, we sorted the obtained costs for the ACO approach, from lowest to highest, and produced the graphic shown in Figure 4. We also obtained the sorted percentage cost saving of the ACO algorithm with respect to the random algorithm (see Figure 5). In order to compute the cost saving of our approach with respect to the random algorithm, we proceeded as follows. For each SPLA-CRIS expression, we computed the cost using both our ACO and the random algorithm and computed the percentage difference of the ACO with respect to the random algorithm. For example, if the cost associated with the selected product by the ACO is equal to 135.0 and the cost obtained by the random algorithm, most likely for a different product but also fulfilling the constraints associated to the `SPL`, is 175.2, then the cost saving is equal to $100 \cdot (1 - \frac{135.0}{175.2}) \approx 22.95$.

The analysis of the results shows that our ACO algorithm always finds feature combinations with lower costs than the random algorithm (or equal cost in the worst cases). Therefore, our algorithm performs better than the random algorithm. On average, our ACO computes solutions that are 14.87% cheaper.

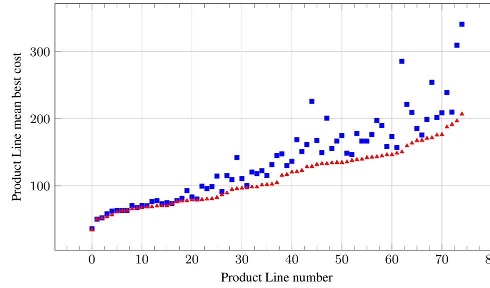

Fig. 4: Sorted obtained costs (blue = random, red = ACO).

We performed a statistical hypothesis test over the results, whose null hypothesis was that the random algorithm and our framework give similar results, that is, both obtain similar costs. We applied a one-way ANOVA test where we tested whether the results of both algorithms are similar in average. Then, we computed the p-value for the experiment, obtaining a p-value of 0.0037. This represents that there is a 00.37% of probability that the null hypothesis is fulfilled. Therefore, we can reject the null hypothesis for the experiment with a confidence higher than 99%, as its p-value is lower than 0.01. In order to double-check our results, we also performed a t-test and obtained the same p-value. Thus, the conclusion is that the performance of our ACO algorithm is better than the random algorithm.

6 Threats to Validity

Threats to *internal validity* refer to uncontrolled factors that can affect the output of the experiments, either in favour or against our hypothesis. The main threat in this category is the possibility of having faults in the code of the experiments. We diminished this threat by carefully testing the code, even using small examples for which we knew the expected results. Additionally, in order to reduce the impact of the randomness associated with our methodology, we repeated the experiments several times.

Threats to *external validity* refer to the generality of our findings to other situations. The main threat in this category is given by the different possible `SPLs` to which we could apply our framework. As the population of `SPLs` is unknown, this threat is not fully addressable. In order to diminish this risk, we considered different `SPLs` in the experiments.

Finally, threats to *construct validity* refer to the relevance of the properties we are measuring for the extrapolation of the results to real-world examples. The main threat in this category is what would happen if we use our framework with real-world `SPLs` and/or with much more complex `SPLs`, which is a matter of future work.

7 Discussion about the suitability of ACO

We have shown that our ACO achieves good solutions for this task. However, it is possible that other heuristics could work better than ACO in this specific framework. Al-

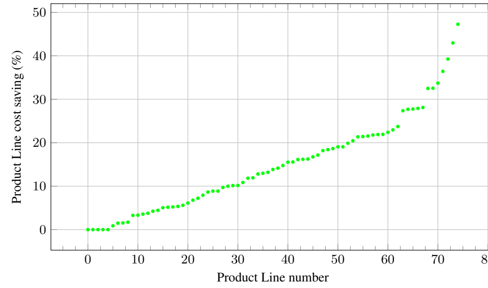

Fig. 5: Sorted cost *saving*.

though this comparison should be further investigated, and it will be indeed a matter of future work, we would like to briefly justify why we decided to use an ACO algorithm.

Our main concern when developing the algorithm was that we needed to provide a much faster solution than brute force while, at the same time, being able to obtain good enough results. Therefore, we classified our problem as an *exploratory problem*. We are aware that there are many evolutionary algorithms that usually work better than a random based search. In our case, we needed a proposal able to *search* in a SPLA-CRIS expression. Fortunately, this kind of syntactical expressions can be transformed into a graph whose *final states* represent all the possible feature combinations that fulfil the expression restrictions. Since this graph can have cycles, we need to perform an extra step to unfold these cycles in order to be able to use a Genetic Programming based algorithm to search for feature combinations inside this graph. This operation would increase the complexity of the approach. In addition, since we are working with a search space based on a graph structure, an approach such as particle swarm optimisation algorithms would suffer because it needs extra adaptation phases that also will increase the complexity of the algorithm. In contrast, ACO can be easily applied to this scenario because our search space is represented as a graph where we are looking for a path from the root to a *final state*, representing a valid feature combination, with a cost as low as possible. So, in order to put into practice our approach we only needed an available interpreter [4] that transform the SPLA-CRIS expressions into appropriate graphs.

8 Conclusions and future work

Software Product Lines are a useful tool for developing software systems in an automatic way and testing them is a must. Integration testing is a process that `SPLs` should overcome: we test how well a new feature is integrated with the already existing features of the `SPL`. If we have the costs of testing each feature of the `SPL`, then we can select the product that contains the new feature that has a lower testing cost, so we can test its integration with the other features of the `SPL` in a quicker and/or cheaper way.

In this paper we have proposed a new framework for feature selection in `SPLs` having testing costs associated with the combination of features. This feature selection generates a product with low cost and a given feature. We have adapted ACO to deal with an *a priory* unknown search space. Therefore, our framework is able to obtain

new feature combinations for a given SPL without computing all the possible feature combinations, which is a time-consuming task. Besides, in order to assess the usefulness of the new framework, we have reported on our most representative experiments. These experiments show that our algorithm is well suited for this task and that it is preferable than other simpler algorithms. Finding sub-optimal solutions in a shorter time can be fundamental in some scenarios, as computing the optimal solution can require a huge amount of resources and time. In fact, in our own experiments we were able to compute the exact solution, by computing all the possible solutions, only for SPLs with a very small number of features. In addition, our experiments show that our algorithm is better than a random search, when giving the same time to both algorithms.

We have identified several research directions concerning applicability, scalability, suitability and adaptability of our framework. Concerning scalability, we will consider more complex SPLs and check whether our technique scales well. Although we will not be able to compare our ACO with brute force, because the latter will not compute the best solution, we want to explore the *limit* of our approach. In addition, we would like to use current mutation testing approaches [11, 28] to efficiently generate and process big amount of mutants representing either non-optimal or faulty selections of features.

With respect to suitability, we will consider two unrelated lines of work. First, although our ACO is well suited for this task, we would like to compare it with other heuristics that could work better than our proposal in this specific framework. Specifically, we would like to compare our ACO approach with other meta-heuristics based on Bee Swarm [23]. A second line of work to analyse the suitability of our framework is to consider SPLs with existing feature selections, produced by an expert, and compare their costs and the ones produced by our framework. In addition, as suggested by a reviewer, it would be interesting to take into account that products including features interacting with the new feature will be more likely to expose bugs, than products running the feature in isolation. Finally, concerning adaptability, we would like to assess the usefulness of our methodology in other frameworks. First, we would like to apply our framework to study formal models of cloud [6,27] and distributed [15,16] systems. We choose this type of systems because we are familiar with them and, more importantly, because they are highly configurable and, therefore, will induce SPLs with many features. Finally, we would like to evaluate whether it is possible to integrate our feature selection framework in existing tools like ProFeat [8].

Acknowledgements

We would like to thank the anonymous reviewers for the careful reading, the many constructive comments and the useful suggestions, which have helped us to further strengthen the paper.

References

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

2. D. Benavides, P. Trinidad, A. Ruiz Cortés, and S. Segura. FaMa. In R. Capilla, J. Bosch, and K. C. Kang, editors, *Systems and Software Variability Management - Concepts, Tools and Experiences*, pages 163–171. Springer, 2013.
3. M. Benito-Parejo and M. G. Merayo. An evolutionary algorithm for selection of test cases. In *22nd IEEE Congress on Evolutionary Computation, CEC'20*, pages E–24535: 1–8. IEEE Computer Society, 2020.
4. C. Camacho, L. Llana, and A. Núñez. Cost-related interface for software product lines. *Journal of Logic and Algebraic Methods in Programming*, 85(1):227–244, 2016.
5. J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*, 104:207–235, 2018.
6. 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*, 163:110522:1–25, 2020.
7. I. do Carmo Machado, P. A. da Mota Silveira Neto, and E. Santana de Almeida. Towards an integration testing approach for software product lines. In *IEEE 13th Int. Conf. on Information Reuse & Integration, IRI'12*, pages 616–623. IEEE, 2012.
8. 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.
9. M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, 2004.
10. M. Eriksson, J. Borstler, and K. Borg. The PLUSS approach - domain modeling with features, use cases and use case realizations. In *9th Int. Conference on Software Product Lines, SPLC'06, LNCS 3714*, pages 33–44. Springer, 2006.
11. P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. Wodel-Test: a model-based framework for language-independent mutation testing. *Software and Systems Modeling*, 20(3):767–793, 2021.
12. M. Griss, J. Favaro, and M. D'Alessandro. Integrating feature modeling with the RSEB. In *5th Int. Conf. on Software Reuse, ICSR'98*, pages 76–85. IEEE Computer Society, 1998.
13. C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon. Bypassing the combinatorial explosion: Using similarity to generate and prioritize T-Wise test configurations for software product lines. *IEEE Transactions on Software Engineering*, 40(7):650–670, 2014.
14. R. M. Hierons, M. Li, X. Liu, S. Segura, and W. Zheng. SIP: optimal product selection from feature models using many-objective evolutionary optimization. *ACM Transactions on Software Engineering and Methodology*, 25(2):17:1–17:39, 2016.
15. 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.
16. 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.
17. 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.
18. A. Ibias and L. Llana. Feature selection using evolutionary computation techniques for software product line testing. In *22nd IEEE Congress on Evolutionary Computation, CEC'20*, pages E–24502: 1–8. IEEE Computer Society, 2020.
19. A. Ibias and M. Núñez. Using a swarm to detect hard-to-kill mutants. In *2020 IEEE Int. Conf. on Systems, Man and Cybernetics, SMC'20*, pages 2190–2195. IEEE Computer Society, 2020.

20. A. Ibias, P. Vazquez-Gomis, and M. Benito-Parejo. Coverage-based grammar-guided genetic programming generation of test suites. In *23rd IEEE Congress on Evolutionary Computation, CEC'21*, pages 2411–2418. IEEE, 2021.
21. M. Jaffar-ur Rehman, F. Jabeen, A. Bertolino, and A. Polini. Testing software components for integration: a survey of issues and techniques. *Software Testing, Verification and Reliability*, 17(2):95–133, 2007.
22. 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.
23. D. Karaboga and B. Akay. A survey: algorithms simulating bee swarm intelligence. *Artificial Intelligence Review*, 31(1):61–85, 2009.
24. R. Lachmann, S. Beddig, S. Lity, S. Schulze, and I. Schaefer. Risk-based integration testing of software product lines. In *11th Int. Workshop on Variability Modelling of Software-Intensive Systems, VaMoS'17*, pages 52–59. ACM Press, 2017.
25. R. E. Lopez-Herrejon, J. Ferrer, F. Chicano, A. Egyed, and E. Alba. Comparative analysis of classical multi-objective evolutionary algorithms and seeding strategies for pairwise testing of software product lines. In *16th IEEE Congress on Evolutionary Computation, CEC'14*, pages 387–396. IEEE, 2014.
26. J. D. McGregor. Testing a software product line. In P. Borba, A. Cavalcanti, A. Sampaio, and J. Woodcook, editors, *Testing Techniques in Software Engineering: 2nd Pernambuco Summer School on Software Engineering, PSSE'07*, pages 104–140. Springer, 2010.
27. A. Núñez, P. C. Cañizares, M. Núñez, and R. M. Hierons. TEA-Cloud: A formal framework for testing cloud computing systems. *IEEE Transactions on Reliability*, 70(1):261 – 284, 2021.
28. M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman. Mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275 – 378. Elsevier, 2019.
29. S. Reis, A. Metzger, and K. Pohl. Integration testing in software product line engineering: A model-based technique. In *10th Int. Conf. on Fundamental Approaches to Software Engineering, FASE'07, LNCS 4422*, pages 321–335. Springer, 2007.
30. D. S. Rodrigues, M. E. Delamaro, C. G. Corrêa, and F. L. S. Nunes. Using genetic algorithms in test data generation: A critical systematic mapping. *ACM Computing Surveys*, 51(2):article 41, 2018.
31. A. S. Sayyad, J. Ingram, T. Menzies, and H. H. Ammar. Optimum feature selection in software product lines: Let your model and values guide your search. In *1st Int. Workshop on Combining Modelling and Search-Based Software Engineering, CMSBSE'13*, pages 22–27. IEEE Computer Society, 2013.
32. 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.
33. J. Shi, M. B. Cohen, and M. B. Dwyer. Integration testing of software product lines using compositional symbolic execution. In *15th Int. Conf. on Fundamental Approaches to Software Engineering, FASE'12, LNCS 7212*, pages 270–284. Springer, 2012.
34. M. Steindl and J. Mottok. Optimizing software integration by considering integration test complexity and test effort. In *10th Int. Workshop on Intelligent Solutions in Embedded Systems, WISES'12*, pages 63–68. IEEE Computer Society, 2012.
35. T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.
36. D. Wang, D. Tan, and L. Liu. Particle swarm optimization algorithm: an overview. *Soft Computing*, 22:387–408, 2018.