Cost-Related Interface for Software product lines[☆]

Carlos Camacho[a,*], Luis Llana[a], Alberto Núñez[a]

[a] *Universidad Complutense de Madrid, Madrid 28040, Spain*

Abstract

Software Product Lines modeling improves software development processes by automating system debugging and analysis. The objective of this paper focuses on extending the formal framework `SPLA` to represent features such as cost objects and comparisons between products in terms of production costs. We illustrate this extension with a practical example by modeling the creation of valid run-lists for Chef, a widely used configuration management tool. Also, we execute our formal specification in a distributed system using SCOOP and we provide strategies to optimize the effort required to compute a `SPLA` term.

Keywords: Software Product Lines, Cost Models, Formal Methods, Feature Models, Configuration systems, Chef.io, run-list

1. Introduction

The main purpose of Software Product Lines (in short, `SPLs`) is to produce products while increasing productivity and shortening the time-to-market period. `SPLs` depend on which software products are being produced and which of them are better for a specific criterion. When products are represented in a product line organization, several modeling approaches can be used to increase both quality and productivity. In most cases this is represented in the form of features, relationships and constraints. For instance, some of these approaches are FODA [1], RSEB [2] and PLUSS [3, 4].

Automatic and formal approaches arise from these graphical representations to model variability and commonality of systems. Formal models allow for detecting errors in the early stages of the production process. Some of the existing approaches use algebras and semantics [5, 6, 7, 8], while others use propositional or first order logic [9, 10, 11, 12, 13].

Feature Oriented Domain Analysis [1] (in short, `FODA`) is a graphical representation of commonality and variability of systems. Figure 1 shows all `FODA` relationships and constraints. In order to perform automatic analysis, graphical

[☆]Research partially supported by the Spanish MEC project ESTuDIo (TIN2012-36812-C02-01), the Comunidad de Madrid project SICOMORo-CM (S2013/ICE-3006) and the UCM-Santander program to fund research groups (group 910606).

[*]Principal corresponding author

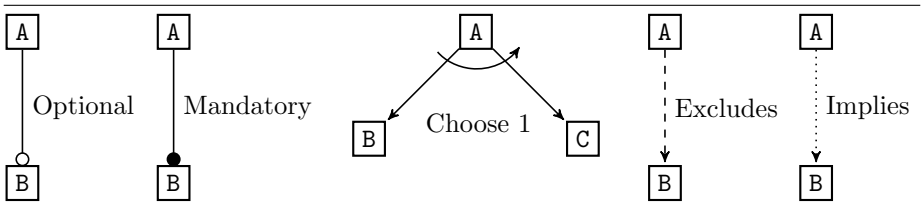

Figure 1: FODA Diagram representation.

representations must be transformed into mathematical entities [14]. Thus, it is necessary to provide the original FODA graphical representation with formal semantics base, where automated analysis can be performed [15]. This issue is solved by using SPLA [8], a formal framework to represent FODA diagrams using process algebras. SPLA can only be applied not only to FODA, but also to represent other feature-related problems and variability models.

Costs within our formal framework refer to the required effort to add a feature to a product under construction. This cost refers to many factors depending on the context of the product line organization. For example, the cost of adding a feature to a product can be equal to the number of lines of code of a software component [16, 17], or the effort, in terms of human hours, to develop that module. This effort is usually measured by using functional metrics [18, 19, 20]. The cost of adding third-party modules, both commercial and open source, to our SPL could be the time associated with integrating it into the product line organization.

The order in which features are computed is important in many software projects and has an important relevance to the final cost of the project. This order can be easily incorporated into the operational semantics of SPLA. In this paper we represent costs with natural numbers. This is not a drawback of our formalism because we can assume a minimum cost unit, and therefore, any cost can be represented as a multiple of this unit.

This paper uses a popular automation framework, called Opscode Chef [21], to present an example about a practical application of costs in our cost-based model. Chef is a software that provides the ability to create software modules for configuring both physical and virtual machines. The main objective of Chef is to translate to code the infrastructure of an organization. This enables to configure each node represented within a Chef environment, in a configuration state.

The main goal of this paper is to extend SPLA in order to include in the model both cost analysis and the capability for comparing valid products. This formal extension is targeted to represent features as cost objects [22] and product lines as cost-based systems [23, 24]. It will enable product-management decisions and cost estimations, even when products are in the earlier stages of the development process or in any iteration of the SPL.

We can summarize the contributions of this paper as follows:

- We have extended the algebra SPLA in order to support costs. Although this extension has been as conservative as possible with the existing language, a rule in the original operational semantics has been modified.
- We have presented a practical example of our approach by using a widely used configuration management system, called *Chef*, in order to calculate the run-list that provides the minimum cost given a set of *Chef Recipes*
- We have developed a prototype of our approach in order to compute valid products among a distributed system using SCOOP.
- We have studied several methods to optimize the computation of SPLA terms in order to reduce the computing effort required to produce the valid products set from our approach.

This paper is structured as follows. In Section 2 we review the use of costs in SPLs. In Section 3 we present some previous results from SPLA and an introduction about the automation platform used in our practical example. Section 4 describes our cost-based extension for computing SPLA terms. Section 5 presents an example of how our semantics extension can be applied to generate valid run-lists (products) for Chef. Section 6 describes our implementation and some considerations that can be taken into account in further implementations. Finally, Section 7 presents our conclusions and future work.

2. Related work

In this section we present some works related to a novel characteristic introduced in the paper, using costs and the order in which features are computed in SPLs.

2.1. "Cost" estimation in SPLs

This section introduces cost estimation analysis for software product lines. An important factor in the success of a software development project depends on the accuracy for estimating the project costs. Generally, the term *Cost* in SPLs is based on quantifying the effort to generate valid products inside a product line environment. In order to achieve this, all software components must be sized. For instance, these costs can be represented as development time, the number of code lines and difficulty to integrate a module into the product line environment.

Software cost estimation is based on code [16, 17] or functional size metrics [18, 19, 20]. The term "size" must be referenced as an important attribute of a software product and it does not always refer to the occupied space or the number of lines of code. Additionally, it may refer to the accumulative estimation of several aspects, like the difficulty in solving a problem, predicting how many developers are needed, defining the development environment and the hardware platform involved in the project or even analyzing how software

components can be integrated. Also, this effort may be formally represented depending on the manager's needs [25], the organizational requirements of product lines, scenarios or available components. This kind of analysis is based on the manager's personal convictions, and therefore, these types of approaches are not yet mature engineering processes. Other studies [22] measure effort levels by analyzing the product cycle evolution and, from that point of view, determining their costs.

2.2. SPLs and SAT-solvers

SAT-solving approaches have being extensively studied to validate feature models [26, 27, 28, 29]. These solutions rely in the definition of a boolean satisfiability problem representing a variability model condition. Our previous studies had demonstrated that SAT approaches are fast enough to compute variability models within the order of thousands features in milliseconds [8]. It can be also applied to compute the products from an SPL without ordering the features positions, in which case we have proven that acceptable computing times are achieved if the model is in the order of thousands of valid products.

In contrast with these works, our approach presents several improvements. First, our work addresses the worst-case scenario, in which we compose the valid products of an SPL by using the appearing order of the features (a special condition not addressed by other studies). Second, we propose several approaches to reduce the total computing time when feature models are processed.

2.3. Optimal cost scheduling

Several studies have been made on algorithms to solve cost-optimal reachability problems [30, 31, 32]. These studies use timed automata in order to model and represent the behavior of computer systems. For modeling real-time systems this approach has been established as a standard formalism using a set of mature checking and modeling tools as Uppaal Cora [33] or Kronos [34], tools for modeling, simulating and validating real-time systems [35].

It is important to remark that these studies rely on solving the problem by using a branch and bound algorithm, or any specific heuristic, in order to cut-off the computing time required to produce a valid output. On the contrary, our approach uses different techniques, which does not include heuristics, for reducing the total computing time.

2.4. Process algebras and cost analysis

Previous studies related to `SPL` modeling costs [16, 17, 18, 19, 20, 25, 22], measure the computed features as whole products. These approaches do not consider the order in which features are computed to generate a valid product. This feature is important since this computing order can affect the final cost of the product. On the contrary, our approach takes into account the order in which features are computed.

We take the benefits of process algebras studies to model `SPLs` using the computing order as an important factor when products are built. Several studies

on process algebras [36, 37, 38, 39] show that properties as costs can be used to model dynamic contexts.

Several formal approaches to model software product lines have been defined [1, 40, 41, 42]. Also studies exist about formalizing different types of feature models [15, 43, 44, 45, 46, 47, 48]. Using previous studies on process algebras and software product lines [8], we are able to model the order in which features are computed, a novel characteristic of software product line modeling not covered by previous `SPLs` research articles.

In the next section we present a brief description of Chef [21] and `SPLA` [8] to make this paper self-contained.

3. Preliminaries

In this section, we present all the required information to make this paper self-contained. Initially, we introduce Chef, an automation platform to code the infrastructure specification of a computing environment. Also, we show a description of `SPLA` [8], which represents our formal specification for Software Product Lines.

3.1. Opscode Chef

Chef is an automation tool [21] that provides the ability to orchestrate the execution of a set of instructions (recipes) in order to configure a specific computing node.

This system eases the process of centralizing the configuration of several nodes in a single repository. Thus, the system allows us to code our node infrastructure. All the centralized code can be applied to virtual or physical machines without any infrastructure condition.

The basic configuration of a Chef infrastructure consists of, at least, one workstation, one server and a set of registered nodes. Once these nodes have executed the chef-client command, the Chef server specify which recipes needs to be executed. Recipes are grouped in cookbooks and roles, depending on the system configuration. Once the Chef server calculates the list of recipes to be executed (run-list) the node proceeds to execute the corresponding steps in order to set the node in a desired state, it being a state in which all the recipes were correctly executed.

In Section 5 we present a complete example that illustrates how to create valid run-lists for a configuration system. However, for the sake of clarity, we first show how Chef components work to generate valid run-lists. Figure 2 describes all the components of a Chef system. All the Chef server components are explained in further detail in the official documentation [21]:

- Chef server. A Chef server is a configuration hub. It stores all the required information to configure the registered nodes.
- Node. A node is a computing resource registered with in the Chef server.

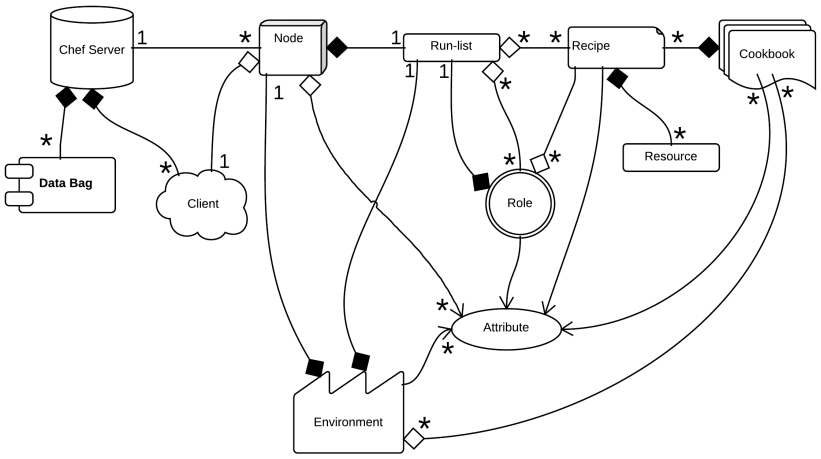

Figure 2: Chef architecture

- Ohai. Ohai is a software tool which retrieves hardware related information about a registered node.
- Run-list. A run-list is a list of recipes required to configure a node.
- Recipe. A recipe is the fundamental configuration element, specifies the resources to be called.
- Cookbook. Represents a set of recipes, resources, providers and attributes.
- Data-bag. A data-bag is a global variable accessible from the Chef server.
- Chef-client. Is a tool running locally in the registered nodes. Is in charge of calculate and execute the corresponding run-list.
- Environment. An environment is a logical binding between nodes, recipes, data-bags and attributes. They can be created to represent organizations workflows like integration, test and production environments.
- Attribute. An attribute is a variable accessible within recipes which can override nodes settings.
- Role. A role is a way group recipes into a single function.
- Resource. A resource defines the step required to execute an action. For example using a resource the user can specify what the recipe must do, however it does not specify how to do it.

The variability model and the cost model in Section 5 have been extracted from the Chef components of Figure 2. It is important to determine all the features, as well as the relationships, the constraints and the cost relations between these features, in order to accurately model a system.

3.2. SPLA

SPLA [8] defines a formal framework to automate the analysis of Software Product Lines. Basically, SPLA consists of a well-defined syntax, as an Extended BNF-like expression, and three semantics: operational, denotational and axiomatic.

3.2.1. SPLA *syntax*

The syntax is divided into two classes of operators, the FODA basic operators ($\cdot \vee \cdot$, $\cdot \wedge \cdot$, $\mathtt{A};\cdot$, $\overline{\mathtt{A}};\cdot$, $\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ \cdot$, $\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ \cdot$) and auxiliary operators ($\mathtt{nil}$, $\checkmark$, $\cdot \backslash \mathtt{A}$, $\cdot \Rightarrow \mathtt{A}$) used to define the language semantics. $\mathcal{F}$ denotes a finite set of features and A, B, C... denote isolated features.

Definition 1. *A* software product line *is a term generated by the following Extended BNF-like expression:*

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

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

Hereafter we briefly explain the operators of the algebra. There are two terminal symbols in the language, $\mathtt{nil}$ and $\checkmark$. The term $\mathtt{nil}$ represents an SPL with no products, while $\checkmark$ is an SPL that has only the empty product. The operator $\mathtt{A}; P$ indicates that A is mandatory while $\overline{\mathtt{A}}; P$ indicates that A is optional.These two operators add the feature A to any product that can be obtained from P. There are two binary operators: $P \vee Q$ and $P \wedge Q$. The first one represents the *choose-one* operator while the second one represents the *conjunction* operator. The operator $\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ P$ represents the *require*, while the operator $\mathtt{A} \not\Rightarrow \mathtt{B}\ \mathtt{in}\ P$ represents the *exclusion* constraint in FODA. Finally, there are two auxiliary operators: $P \Rightarrow \mathtt{A}$ indicates that feature A is mandatory and $P \backslash \mathtt{B}$ indicates that feature B is forbidden in P.

$$
\begin{array}{lll}
[\textbf{tick}]\ \checkmark \xrightarrow{\checkmark} \texttt{nil} & [\textbf{feat}]\ \texttt{A}; P \xrightarrow{\texttt{A}} P & [\textbf{ofeat1}]\ \overline{\texttt{A}}; P \xrightarrow{\texttt{A}} P \\
[\textbf{ofeat2}]\ \overline{\texttt{A}}; P \xrightarrow{\checkmark} \texttt{nil} & [\textbf{cho1}]\ \dfrac{P \xrightarrow{a} P_1}{P \vee Q \xrightarrow{a} P_1} & [\textbf{cho2}]\ \dfrac{P \xrightarrow{a} P_1}{Q \vee P \xrightarrow{a} P_1} \\
[\textbf{con1}]\ \dfrac{P \xrightarrow{\texttt{A}} P_1}{P \wedge Q \xrightarrow{\texttt{A}} P_1 \wedge Q} & [\textbf{con2}]\ \dfrac{P \xrightarrow{\texttt{A}} P_1}{Q \wedge P \xrightarrow{\texttt{A}} Q \wedge P_1} & [\textbf{con3}]\ \dfrac{P \xrightarrow{\checkmark} \texttt{nil}, Q \xrightarrow{\checkmark} \texttt{nil}}{P \wedge Q \xrightarrow{\checkmark} \texttt{nil}} \\
[\textbf{req1}]\ \dfrac{P \xrightarrow{\texttt{C}} P_1,\ \texttt{C} \neq \texttt{A}}{\texttt{A} \Rightarrow \texttt{B}\ \texttt{in}\ P \xrightarrow{\texttt{C}} \texttt{A} \Rightarrow \texttt{B}\ \texttt{in}\ P_1} & [\textbf{req2}]\ \dfrac{P \xrightarrow{\texttt{B}} P_1}{\texttt{B} \Rightarrow \texttt{A}\ \texttt{in}\ P \xrightarrow{\texttt{A}} \texttt{B}; P_1} & [\textbf{req3}]\ \dfrac{P \xrightarrow{\checkmark} \texttt{nil}}{\texttt{A} \Rightarrow \texttt{B}\ \texttt{in}\ P \xrightarrow{\checkmark} \texttt{nil}} \\
[\textbf{excl1}]\ \dfrac{P \xrightarrow{\texttt{C}} P_1,\ \texttt{C} \neq \texttt{A} \wedge \texttt{C} \neq \texttt{B}}{\texttt{A} \not\Rightarrow \texttt{B}\ \texttt{in}\ P \xrightarrow{\texttt{C}} \texttt{A} \not\Rightarrow \texttt{B}\ \texttt{in}\ P_1} & [\textbf{excl2}]\ \dfrac{P \xrightarrow{\texttt{A}} P_1}{\texttt{A} \not\Rightarrow \texttt{B}\ \texttt{in}\ P \xrightarrow{\texttt{A}} P_1 \backslash \texttt{B}} & [\textbf{excl3}]\ \dfrac{P \xrightarrow{\texttt{B}} P_1}{\texttt{A} \not\Rightarrow \texttt{B}\ \texttt{in}\ P \xrightarrow{\texttt{B}} P_1 \backslash \texttt{A}} \\
[\textbf{excl4}]\ \dfrac{P \xrightarrow{\checkmark} \texttt{nil}}{\texttt{A} \not\Rightarrow \texttt{B}\ \texttt{in}\ P \xrightarrow{\checkmark} \texttt{nil}} & [\textbf{forb1}]\ \dfrac{P \xrightarrow{\texttt{B}} P_1,\ \texttt{B} \neq \texttt{A}}{P \backslash \texttt{A} \xrightarrow{\texttt{B}} P_1 \backslash \texttt{A}} & [\textbf{forb2}]\ \dfrac{P \xrightarrow{\checkmark} \texttt{nil}}{P \backslash \texttt{A} \xrightarrow{\checkmark} \texttt{nil}} \\
[\textbf{mand1}]\ \dfrac{P \xrightarrow{\checkmark} \texttt{nil}}{P \Rightarrow \texttt{A} \xrightarrow{A} \checkmark} & [\textbf{mand2}]\ \dfrac{P \xrightarrow{\texttt{A}} P_1}{P \Rightarrow \texttt{A} \xrightarrow{\texttt{A}} P_1} & [\textbf{mand3}]\ \dfrac{P \xrightarrow{\texttt{B}} P_1,\ \texttt{A} \neq \texttt{B}}{P \Rightarrow \texttt{A} \xrightarrow{\texttt{B}} P_1 \Rightarrow \texttt{A}}
\end{array}
$$

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

Figure 3: SPLA operational semantics rules.

3.2.2. Operational Semantics

Operational semantics adds meaning to SPLA terms. Basically, it defines how terms must be processed. The operational semantics of SPLA is presented as a labelled transition system for any term $P \in \texttt{SPLA}$. The transitions are *annotated* with the set $\mathcal{F} \cup \{\checkmark\}$, with $\mathcal{F}$ being the set of features and $\checkmark \notin \mathcal{F}$. In particular, if $\texttt{A} \in \mathcal{F}$, the transition $P \xrightarrow{\texttt{A}} Q$ means that there is a product of P that contains the feature A. Transitions of the form $P \xrightarrow{\checkmark} \texttt{nil}$ mean that a product has been produced. The formal operational semantic rules of the algebra are presented in Figure 3. Rule [**req2**] has changed in this operational semantics with respect to the one presented in [8]. This change will be discussed later in Section 4.3

Definition 2. *Let $P, Q \in \texttt{SPLA}$ and $\texttt{A} \in \mathcal{F} \cup \{\checkmark\}$. There is a transition from P to Q labeled with the symbol* A*, denoted by $P \xrightarrow{\texttt{A}} Q$, if it can be deduced from the rules in Figure 3.* □

Before giving any properties of this semantics let us explain the rules in Figure 3. First we have the rule [**tick**]. The intuitive meaning of this rule is that we have reached a point where a product of the SPL has been computed. Let us note that nil has no transitions, this means that nil does not have any valid products.

Rules [**feat**], [**ofeat1**], and [**ofeat2**] deal directly with the computation of features. Rule [**ofeat2**] means that we have a valid product discarding an optional feature. The prefix operator $\overline{\texttt{A}}; P$ indicates that feature A is a prerequisite to all products derived by P; so, if we discard feature A we also have to discard all these products.

Rules [**cho1**] and [**cho2**] deal with the *choose-one* operator. These rules indicate that the computation of $P \vee Q$ must choose between the features in P or the features in Q.

Rules [**con1**], [**con2**], and [**con3**] deal with the *conjunction* operator. The main rules are [**con1**] and [**con2**]. These rules are symmetrical to each other. They indicate that any product of $P \wedge Q$ must have the features of P and Q. Rule [**con3**] indicates that both members have to agree in order to *deliver* a product.

Rules [**req1**], [**req2**], and [**req3**] deal with the *require* constraint. Rule [**req1**] indicate that $\texttt{A} \Rightarrow \texttt{B in } P$ behaves like P as long as feature A has not been computed. Rule [**req2**] indicates that A is a prerequisite of B: if the inner component can produce B, we must assure that A is produced first. Finally [**req3**] marks that a product has been delivered.

Rules [**excl1**] to [**excl4**] deal with the exclusion constraint. Rule [**excl1**] indicates that $\texttt{A} \not\Rightarrow \texttt{B in } P$ behaves like P as long as P does not compute feature A or B. Rule [**excl2**] indicates that once P produces A, feature B must be forbidden. Rule [**excl3**] indicates just the opposite: when feature B is computed, then A must be forbidden. This rule might be surprising, but there is no reason to forbid $\texttt{A} \not\Rightarrow \texttt{B in } P$ to compute feature B. So if $\texttt{A} \not\Rightarrow \texttt{B in } P$ computes feature B then feature A must be forbidden. Otherwise the exclusion constraint would not have been fulfilled.

Rules [**forb1**] and [**forb2**] deal with the auxiliary operator $P\backslash\mathtt{A}$ that forbids the computation of feature $\mathtt{A}$. Let us note that there is no rule that computes $\mathtt{A}$. This means that if feature $\mathtt{A}$ is computed by P, the computation is blocked and no products can be produced.

Rules [**mand1**], [**mand2**], and [**mand3**] deal with the auxiliary operator $P \Rightarrow \mathtt{A}$. Rule [**mand1**] indicates that feature $\mathtt{A}$ must be computed before *delivering* a product. Rules [**mand2**] and [**mand3**] indicates that $P \Rightarrow \mathtt{A}$ behaves like P. We need two rules in this case because when feature $\mathtt{A}$ is computed it is no longer necessary to continue considering this feature as mandatory and so the operator can be removed from the term.

Once the operational semantics of the algebra is defined, we can obtain the traces and the valid products.

Definition 3. *A trace is a sequence $s \in \mathcal{F}^*$. The empty trace is denoted by ϵ. Let s_1 and s_2 be traces, we denote the concatenation of s_1 and s_2 by $s_1 \cdot s_2$. Let $\mathtt{A} \in \mathcal{F}$ and let s be a trace, we say that* $\mathtt{A}$ is in the trace *s, written $\mathtt{A} \in s$, iff there exist traces s_1 and s_2 such that $s = s_1 \cdot \mathtt{A} \cdot s_2$.*
We can extend the transitions in Definition 2 to traces. Let $P, Q, R \in \mathtt{SPLA}$, we inductively define the transitions $P \xrightarrow{s} R$ as follows: $P \xrightarrow{\epsilon} P$; if $P \xrightarrow{\mathtt{A}} Q$ and $Q \xrightarrow{s} R$, then $P \xrightarrow{\mathtt{A}\cdot s} R$. □

Those traces ending with ✓ are the only ones considered as valid products. It is important to remark that, since ✓ symbol is not a feature. It is not included in the product.

Definition 4. *Let $P \in \mathtt{SPLA}$ and $s \in \mathcal{F}^*$,*

- *s is a* successful trace *of P, written $s \in \mathtt{tr}(P)$, iff $P \xrightarrow{s} Q \xrightarrow{\checkmark} \mathtt{nil}$.*
- *The* set induced by *s, written $[s]$, is the set obtained from the elements of the trace without considering their position in the trace.*
- *Let $P \in \mathtt{SPLA}$, we define* the products of *P, written $\mathtt{prod}(P)$, as $\mathtt{prod}(P) = \{[s] \mid s \in \mathtt{tr}(P)\}$.*

□

4. Cost-related extension

In this section we present the extension of SPLA that allows us to manage costs using the operational semantics described in Section 3. In order to be as conservative as possible with respect to SPLA, the operational semantics will remain unmodified. In order to fulfill this, we define new transitions that allow the computation of costs. However, before defining these new transitions, we have to define a *cost* function.

4.1. Cost function

The addition of a software component to a product has an associated cost. However, this cost is not constant, and it depends on other software components in the product. We present the software components as features and the product under construction as a trace. Therefore, the cost function has two parameters: a trace containing the previously computed features and the feature being produced. This function returns the cost required for adding a feature to the computed features trace.

In this paper we assume that costs can be represented by natural numbers. Although all the results presented in this paper would be equally valid if we had chosen non-negative real numbers to represent costs, we use natural numbers, instead of real numbers, because the latter requiere more complexity to be computed. Moreover, the adoption of natural numbers does not restrict the expressive power of the model in practical cases.

In some cases, it is possible that the cost of a feature cannot be computed from its trace. For instance, before introducing a software module, it is necessary to install its prerequisites. So, we will introduce a new symbol $\perp$ to represent this undefinedness. We consider the set $\mathbb{N}_\perp = \mathbb{N} \cup \{\perp\}$, and therefore, we extend the arithmetic to this symbol as follows: $\forall x \in \mathbb{N}_\perp : \; x + \perp = \perp + x = \perp$. We also assume that $x \leq \perp$ for any $x \in \mathbb{N}_\perp$.

Definition 5. *A* cost function *is a function:*

$$\mathtt{c} : \mathcal{F}^* \times \mathcal{F} \mapsto I\!N_\perp$$

□

Example 1. Let us consider the term $P = \mathtt{A}; (\mathtt{B}; \checkmark \vee \mathtt{C}; (\overline{\mathtt{D}}; \checkmark \wedge \mathtt{E}; \checkmark))$. The set of products of P are:

$$\mathtt{prod}\,(P) = \big\{\{\mathtt{A}, \mathtt{B}\}, \{\mathtt{A}, \mathtt{C}, \mathtt{E}\}, \{\mathtt{A}, \mathtt{C}, \mathtt{D}, \mathtt{E}\}\big\}$$

If we consider the order of the features within the products, we observe that `A` is always the first produced feature, then `B` or `C`, and finally `D` and `E`. However, the formalism does not establish an order between `D` and `E`. Now we can consider the cost function:

$$\mathtt{c}(s, x) = \begin{cases} \perp & \text{if } x = \mathtt{B} \text{ and } \mathtt{A} \notin s \\ 2 & \text{if } x = \mathtt{D} \text{ and } \mathtt{E} \notin s \\ 1 & \text{otherwise} \end{cases}$$

Although there is no pre-established order for producing features `D` and `E`, producing `D` before `E` is more expensive than producing `E` before `D`. Let us note that the cost function defines values for traces that are not generated by the term P, for instance $\mathtt{c}(\epsilon, \mathtt{B})$ or $\mathtt{c}(\mathtt{B}, \mathtt{A})$. Cost functions can be independent from concrete terms, and this particular cost function will be used again in Example 2. □

4.2. Cost transitions

We define the transition system to compute costs as a graph, where each node corresponds to a state of the labelled transition system, induced from a term $P \in \texttt{SPLA}$. Intuitively, each state corresponds to a phase of the product construction. Thus, we need to store, in each state, the components that have been incorporated to the product (the trace) and the current cost of the product. Each node is represented as a triple (P, s, n) where P is the SPLA under evaluation, s is a trace of features and n is a natural number indicating the current cost of the product.

Definition 6. *Let $P \in \texttt{SPLA}$ and let $\texttt{c}$ be a cost function, the* graph *that represents the* costs *of P, with respect to $\texttt{c}$, is the least graph $(N, \to)$ satisfying:*

[**ini**] $(P, \epsilon, 0) \in N$.

[**cost1**] *If $(Q, s, n) \in N$, $Q \xrightarrow{\texttt{A}} Q_1$ and $\texttt{A} \notin s$, then $(Q_1, s\texttt{A}, n + \texttt{c}(s, \texttt{A})) \in N$ and $(Q, s, n) \to (Q_1, s\texttt{A}, n + \texttt{c}(s, \texttt{A}))$.*

[**cost2**] *If $(Q, s, n) \in N$, $Q \xrightarrow{\texttt{A}} Q_1$ and $\texttt{A} \in s$, then $(Q_1, s\texttt{A}, n) \in N$ and $(Q, s, n) \to (Q_1, s\texttt{A}, n)$.*

□

Rule [**ini**] indicates that initially there are no features computed, and therefore, the cost is 0. Rule [**cost1**] corresponds to the case when a new feature of an SPLA is computed, it is added to the trace s and the cost of producing this feature is added to the accumulated cost of the product. This condition applies only if $\texttt{A}$ is not present in s. Rule [**cost2**] indicates that, if the feature has been previously produced, the cost does not change.

The following proposition indicates that, once an impossible configuration is reached, in terms of costs, it is pointless to continue computing features. In this case, this configuration is denoted as $\bot$.

Proposition 1. *Let $P, Q \in \texttt{SPLA}$ and $s = \texttt{A}_1 \cdots \texttt{A}_l \in \mathcal{F}^*$, then $P \xrightarrow{s} Q$ iff there is $n_1, \ldots n_l \in \mathbb{N}_\bot$ and $Q_0, Q_1 \ldots Q_k$ such that $P = Q_0$, $Q = Q_l$ and*

$$(Q_0, \epsilon, 0) \to (Q_1, \texttt{A}_1, n_1) \to \cdots \to (Q_l, \texttt{A}_1\texttt{A}_2 \cdots \texttt{A}_l, n_l)$$

Moreover, if there exists $1 \leq i \leq l$ such that $n_i = \bot$, then $n_j = \bot$ for $i \leq j \leq k$. □

Lemma 1. *Let $P \in \texttt{SPLA}$, $\texttt{c}$ be a cost function, and s be a trace such that there are two computations:*

$$P = P_0 \xrightarrow{\texttt{A}_1} P_1 \cdots \xrightarrow{\texttt{A}_n} P_n$$

and

$$P = Q_0 \xrightarrow{\texttt{A}_1} Q_1 \cdots \xrightarrow{\texttt{A}_n} Q_n$$

with $s = \mathtt{A}_1 \cdots \mathtt{A}_n$. *Then if we consider the corresponding cost transitions:*

$$(P_0, \epsilon, 0) \to (P_1, \mathtt{A}_1, c_1) \cdots \to (P_n, s, c_n)$$
$$(Q_0, \epsilon, 0) \to (Q_1, \mathtt{A}_1, c'_1) \cdots \to (Q_n, s, c'_n)$$

then $c_n = c'_n$.
Proof. It is only necessary to take into account that the cost only depends on the trace, it does not depend on the intermediate terms. □

Definition 7. *Let us consider cost transitions that compute* s. *Let* $P \in \mathtt{SPLA}$, $\mathtt{c}$ *be a cost function and* $s \in \mathtt{tr}(P)$. *We define the cost of producing* s, *written* $\mathtt{tc}(P, s)$, *as follows:*

$$(P_0, \epsilon, 0) \to (P_1, \mathtt{A}_1, c_1) \cdots (P_n, s, c_n) \quad P_n \xrightarrow{\checkmark} \mathtt{nil}$$

then $\mathtt{tc}(P, s) = c_n$. □

The cost of producing a product is not unique because this cost depends on the trace that generated it. So forth, we need to consider a set of costs for any product.

Definition 8. *Let* $\mathtt{c}$ *be a cost function. Let us consider the function* $\mathtt{c}_{\mathtt{SPLA}} : \mathtt{SPLA} \times \mathcal{P}(\mathcal{F}^*) \mapsto \mathcal{P}(I\!N_\perp)$ *defined as follows:*

$$\mathtt{c}_{\mathtt{SPLA}}(Q, p) = \{ n \mid n = \mathtt{tc}(P, s) \text{ and } [s] = p \}$$

□

Let us note that if $p \notin \mathtt{prod}(P)$ we obtain $\mathtt{c}_{\mathtt{SPLA}}(P, p) = \varnothing$. Therefore, we are able to compare two Software Product Lines. One product line is better than another if it can produce the same products at a lower cost.

Definition 9. *Let* $P, Q \in \mathtt{SPLA}$ *and let* $\mathtt{c}$ *a cost function. We say that* P *is better than* Q *with respect to* $\mathtt{c}$, *written* $P \leq_{\mathtt{c}} Q$ *iff* $\mathtt{prod}(Q) \subseteq \mathtt{prod}(P)$ *and for any* $p \in \mathtt{prod}(Q)$ *the following holds:*

$$\min\{n \mid n \in \mathtt{c}_{\mathtt{SPLA}}(P, p)\} \leq \min\{n \mid n \in \mathtt{c}_{\mathtt{SPLA}}(Q, p)\}$$

□

The conditions $p \in \mathtt{prod}(Q)$ and $\mathtt{prod}(Q) \subseteq \mathtt{prod}(P)$ imply that the involved sets $\{n \mid n \in \mathtt{c}_{\mathtt{SPLA}}(P, p)\}$ and $\{n \mid n \in \mathtt{c}_{\mathtt{SPLA}}(Q, p)\}$ are not empty and therefore there exists the minimum of both sets.

4.3. New rule [**req2**]

At this point, it is worth discussing the change in the operational semantics (Figure 3) with respect to the original operational semantics in [8]. Thus, [**req2**] rule has been changed, the old rule was the following:

$$[\mathbf{req2'}] \quad \frac{P \xrightarrow{\mathtt{A}} P_1}{\mathtt{A} \Rightarrow \mathtt{B} \ \mathtt{in}\ P \xrightarrow{\mathtt{A}} P_1 \Rightarrow \mathtt{B}}$$

Rule [**req2′**] presents some problems when computing costs, as the following example illustrates.

Example 2. Let us consider the cost function in Example 1 and the term $P = \mathtt{B} \Rightarrow \mathtt{A}\ \mathtt{in}\ \mathtt{B}; \checkmark$. By applying the transition rules [**ini**], [**cost1**] and [**cost2**] from Figure 3 we obtain the transitions:

$$(\mathtt{B} \Rightarrow \mathtt{A}\ \mathtt{in}\ \mathtt{B}; \checkmark, \epsilon, 0) \rightarrow (\checkmark \Rightarrow \mathtt{A}, \mathtt{B}, \bot)$$

Thus, the cost of computing the product [BA] is $\bot$. □

Intuitively, the *requires* operator establishes a pre-condition between features. More precisely, $\mathtt{B} \Rightarrow \mathtt{A}\ \mathtt{in}\ P$ indicates that A is a pre-condition of B. This is the reason of changing the old rule [**req2′**] by the new rule [**req2**].

This change is not very important from a theoretical point because, as Proposition 2 shows, it does not alter the original semantics. In order to justify this assessment we consider two transition systems: one with the old rule [**req2′**] and the new one presented in this paper rule [**req2**]. In order to differentiate the transition systems, the transitions of the old one are denoted by $P \xrightarrow{\mathtt{A}}_o Q$, and the transitions of the new one are denoted by $P \xrightarrow{\mathtt{A}}_n Q$. In the following Lemma 2 and Proposition 2, we consider $\mathtt{prod}_n(P)$ and $\mathtt{tr}_n(P)$ as the set of products and traces of P obtained by using the new transition system, while $\mathtt{prod}_o(P)$ and $\mathtt{tr}_o(P)$ correspond to those sets computed by following the old transition system in [8].

Lemma 2. *Let* $P \in \mathtt{SPLA}$, $\mathtt{A}, \mathtt{B} \in \mathcal{F}$ *and* $s \in \mathcal{F}^*$ *be a valid trace.*

1. *If* $s \in \mathtt{tr}_o(A \Rightarrow \mathtt{P})$ *then either* $s \in \mathtt{tr}_o(P)$ *and* $\mathtt{A} \in [s]$ *or there exists* $s' \in \mathcal{F}^*$ *such that* $s = s' \cdot \mathtt{A}$ *and* $s' \in \mathtt{tr}_o(P)$.
2. *If* $[s] \in \mathtt{prod}_o(P)$ *and* $\mathtt{A} \in [s]$ *then* $[s] \cup \{\mathtt{B}\} \in \mathtt{prod}_o(\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ P)$.

Proof. Let us consider the part 1 of the Lemma 2. Let us proceed by induction on the length of s ($|s|$). The case when $|s| = 0$ is not possible because there are no rules allowing the *mandatory* operator produce the ✓ symbol. Thus, the base case is $|s| = 1$. The only possibilities is to apply rules [**mand1**] to obtain $P \Rightarrow \mathtt{A} \xrightarrow{\mathtt{A}}_n \checkmark$, or [**mand2**] to obtain $P \Rightarrow \mathtt{A} \xrightarrow{\mathtt{A}}_n P_1 \xrightarrow{\checkmark}_n \mathtt{nil}$. In the first case it is enough to consider $s' = \epsilon$ to obtain the result and in the second case $s = \mathtt{A}$ and $\mathtt{A} \in \mathtt{tr}_o(P)$.

Let us consider $|s| > 1$. Thus, there are 3 cases depending on the first rule applied. If the first rule is [**mand1**] the result is obtained as before. If the first rule is [**mand2**], we have $P \Rightarrow \mathtt{A} \xrightarrow{\mathtt{A}}_n P_1$, and $s = \mathtt{A} \cdot s_1$, where $s_1 \in \mathtt{tr}_o(P_1)$. Because of the premise of [**mand2**], we have $P \xrightarrow{\mathtt{A}}_n P_1$, so $s \in \mathtt{tr}_o(P)$. Finally, if the first rule is [**mand3**], we obtain $P \Rightarrow \mathtt{A} \xrightarrow{\mathtt{B}}_n P_1 \Rightarrow \mathtt{A}$. Then $s = \mathtt{B} \cdot s_1$ with $s_1 \in \mathtt{tr}_o(P_1 \Rightarrow \mathtt{A})$. Because of the premise of the rule, we obtain $P \xrightarrow{\mathtt{B}}_n P_1$, and by the induction hypothesis we obtain the result.

The proof of part 2 is just a consequence of the soundness and completeness result of the denotational semantics (Theorem 1) in [8]. □

Proposition 2. *Let* $P \in \mathtt{SPLA}$*, then* $\mathtt{prod}_n(P) = \mathtt{prod}_o(P)$.

Proof. First let us consider $p \in \mathtt{prod}_o(P)$, by definition there is some s such that $s \in \mathtt{tr}_o(P)$ and $p = [s]$. We prove that $p \in \mathtt{prod}_n(P)$ by induction on the length of s. If $|s| = 0$ then $P \xrightarrow{\checkmark} \mathtt{nil}$, since rule [**req2**] does not enable that transition, so we obtain $P \xrightarrow{\checkmark}_n \mathtt{nil}$. So, $s \in \mathtt{tr}_n(P)$ and therefore $p \in \mathtt{prod}_n(P)$.

Let us consider $|s| > 0$, and the rule applied to derive the first transition. For all the cases, but when the rule is [**req2′**], the result is trivial by induction since these rules coincide in both transition systems. Let us suppose that the rule to obtain the first transition of s is [**req2′**]. In this case we obtain: $P = \mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ P_1$, $P_1 \xrightarrow{\mathtt{A}}_o Q$, $P \xrightarrow{\mathtt{A}}_o Q \Rightarrow \mathtt{B}$, $s = \mathtt{A} \cdot s_1$ (where $s_1 \in \mathtt{tr}_o(Q \Rightarrow \mathtt{B})$) and $p = \{\mathtt{A}\} \cup [s_1]$. Since $s_1 \in \mathtt{tr}_o(Q \Rightarrow \mathtt{B})$, by Lemma 2.1, there are two cases (1) $s_1 \in \mathtt{tr}_o(Q)$ and $\mathtt{B} \in [s_1]$ or (2) there exist s_1' such that $s_1' = s_1 \cdot \mathtt{B}$ and $s_1' \in \mathtt{tr}_o(Q)$. In case (1), by the induction hypothesis we obtain $s_1 \in \mathtt{tr}_n(Q)$. Then $\mathtt{B} \cdot s_1 \in \mathtt{tr}_o(\mathtt{B}; Q)$, since $\mathtt{B} \in [s_1]$ we obtain $[\mathtt{B} \cdot s_1] = [s_1]$, so $[s_1] \in \mathtt{prod}_n(\mathtt{B}; Q)$. In case (2), by induction $s_1' \in \mathtt{tr}_n(Q)$, and then $\mathtt{B} \cdot s_1' \in \mathtt{tr}_n(\mathtt{B}; Q)$ and $[s_1] = [s_1'] \cup \{\mathtt{B}\}$. Thus, we can conclude also in this case $[s_1] \in \mathtt{prod}_n(\mathtt{B}; Q)$. In the new transition system, we can apply [**req2**] to obtain $P \xrightarrow{\mathtt{A}}_n \mathtt{B}; Q$, so $\{\mathtt{A}\} \cup [s_1] \in \mathtt{prod}_n(P)$.

Now let us consider $p \in \mathtt{prod}_n(P)$, so there exists a trace $s \in \mathtt{tr}_n(P)$ such that $p = [s]$. We show that $p \in \mathtt{prod}_o(P)$ by induction on the length of s. The case base is just as the previous one. The inductive case is also made by considering the first rule applied to derive s, and again all cases are trivial except the case when the first rule applied is [**req2′**]. In this case we have that $P = \mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ P_1$, $P_1 \xrightarrow{\mathtt{A}}_n P_1'$ and $P \xrightarrow{\mathtt{B}}_n \mathtt{A}; P_1'$. Now we obtain that there exist a trace s_1 such that $s = \mathtt{B} \cdot \mathtt{A} \cdot s_1$ and $[\mathtt{A} \cdot s_1] \in \mathtt{prod}_n(P_1)$. By the induction hypothesis we obtain $[\mathtt{A} \cdot s_1] \in \mathtt{prod}_o(P_1)$ and then, by Lemma 2.2, $[s] = [\mathtt{A} \cdot s_1] \cup \{\mathtt{B}\} \in \mathtt{prod}_o(\mathtt{A} \Rightarrow \mathtt{B}\ \mathtt{in}\ P_1)$. □

With Rule [**req2**], we are able to compute the costs of the products of the term in Example 2. Let us remark that a product can be obtained with different traces: $P = \{\mathtt{A}, \mathtt{B}\} = [\mathtt{AB}] = [\mathtt{BA}]$. First, let us consider the following lemma, which establishes that the cost of producing a trace does not depend on the intermediate states of the computation producing the trace s.

Table 1: Cost function

c	A	B	C	D	E
ϵ	1	1	1	1	1
A	$\perp$	22	20	125	134
AC	12	15	$\perp$	13	2
ACE	$\perp$	11	$\perp$	332	$\perp$
ACD	$\perp$	132	$\perp$	$\perp$	581

4.4. Rules execution example

In this Section we present an example to show how cost is managed in our framework. In order to show its applicability, we model a Software Product Line that consists of five functional components (A,B,C,D and E). We consider the SPLA term from Example 1:

$$P = \mathtt{A}; (\mathtt{B}; \checkmark \vee \mathtt{C}; (\overline{\mathtt{D}}; \checkmark \wedge \mathtt{E}; \checkmark))$$

We have analyzed this term by using the cost function generated from the values shown in Table 1.

This table shows the cost for some of the possible values in $\mathcal{F}^* \times \mathcal{F}$. We define $\mathtt{c}(s, x) = \perp$ for the rest of the pairs $(s, x) \in \mathcal{F}^* \times \mathcal{F}$ that do not appear in this table. Let us note that the cost function gives values to some possibilities not allowed by the term P. For instance $\mathtt{c}(\mathtt{AC}, \mathtt{B}) = 15$, this indicates the possible value of producing B if previously we have produced A and C in general. This cost function could also be used in a term like $Q = \mathtt{A}; (\mathtt{B} \wedge \mathtt{C})$, but in the case of the term P above this concrete value is never computed because in P the features C and B are never in the same product.

Figure 4 and Figure 5 describe how P is processed by using the operational semantics presented in Figure 3, where the cost processing rules are also applied as long as the semantic rules are being computed.

In this example, the first processed feature is Feature A using [**feat**] rule. Once this feature has been processed, we apply rule [**cost1**]. Thus, we obtain the transition $Stage_1$:

$$(\mathtt{A}; (\mathtt{B}; \checkmark \vee \mathtt{C}; (\overline{\mathtt{D}}; \checkmark \wedge \mathtt{E}; \checkmark)), \epsilon, 0) \rightarrow (\mathtt{B}; \checkmark \vee \mathtt{C}; (\overline{\mathtt{D}}; \checkmark \wedge \mathtt{E}; \checkmark), \mathtt{A}, 1)$$

Next, we apply rule [**cho1**] to obtain $Stage_2$:

$$(\mathtt{B}; \checkmark \vee \mathtt{C}; (\overline{\mathtt{D}}; \checkmark \wedge \mathtt{E}; \checkmark), \mathtt{A}, 1) \rightarrow (\checkmark, \mathtt{AB}, 23))$$

and finally we apply [**cho2**] to obtain $Stage_3$:

$$(\mathtt{B}; \checkmark \vee \mathtt{C}; (\overline{\mathtt{D}}; \checkmark \wedge \mathtt{E}; \checkmark), \mathtt{A}, 1) \rightarrow (\overline{\mathtt{D}}; \checkmark \wedge \mathtt{E}; \checkmark, \mathtt{AC}, 21)$$

As a result, we confirm that the cost of the trace AB is 23. Since this is the only way to obtain the product $\{\mathtt{A}, \mathtt{B}\}$, its cost is 23.

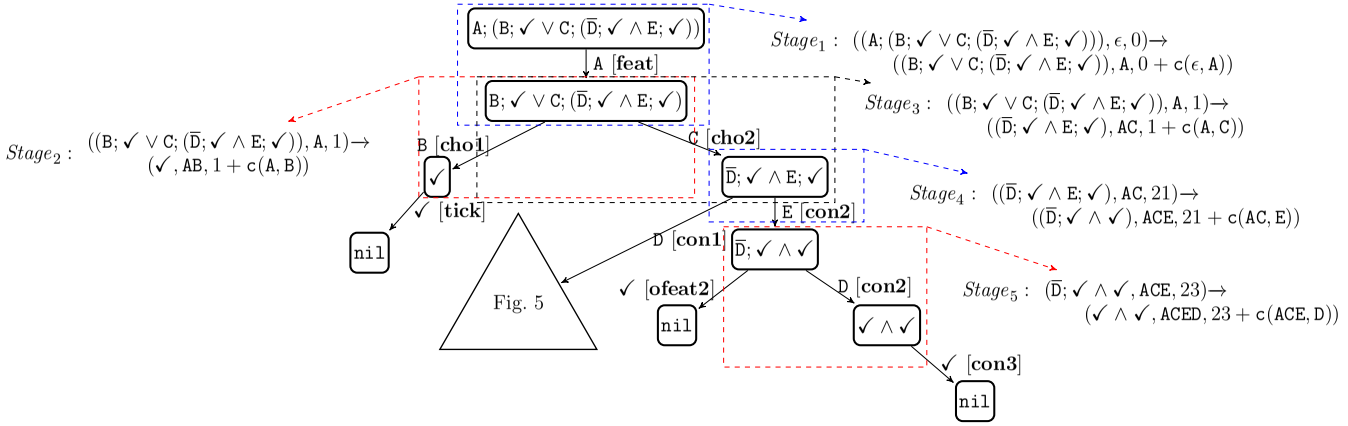

Figure 4: FODA cost-related example 1/2.

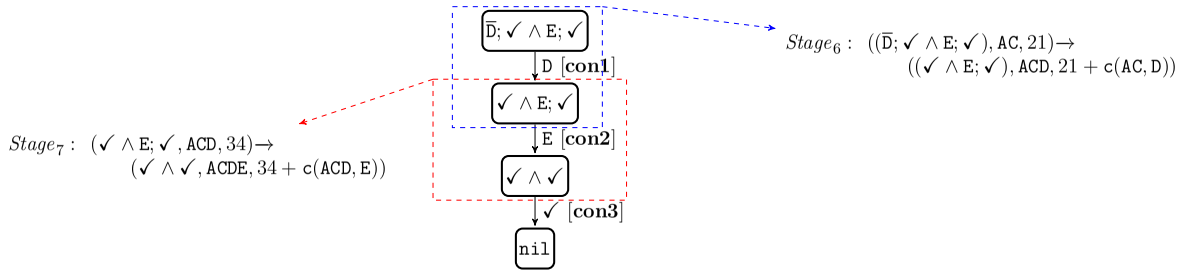

Figure 5: FODA cost-related example 2/2.

The term $\overline{\mathtt{D}}; \checkmark \wedge \mathtt{E}; \checkmark$ has two outgoing transitions: one obtained from applying rule [**con1**], which leads to the term $\checkmark \wedge \mathtt{E}$, and the other applying rule [**con2**], leading to the term $\overline{\mathtt{D}}$. The term $\checkmark \wedge \checkmark$ obtains the cost transition $State_6$ (see Figure 5):

$$(\overline{\mathtt{D}}; \checkmark \wedge \mathtt{E}; \checkmark, \mathtt{AC}, 21) \rightarrow (\checkmark \wedge \mathtt{E}; \checkmark, \mathtt{ACD}, 34)$$

and $Stage_4$:

$$(\overline{\mathtt{D}}; \checkmark \wedge \mathtt{E}; \checkmark, \mathtt{AC}, 21) \rightarrow (\overline{\mathtt{D}}; \checkmark \wedge \checkmark, \mathtt{ACE}, 23)$$

From this last term we can apply two rules: [**ofeat2**] and [**con2**]. The first one leads to $\checkmark$, where we obtain the product $\{\mathtt{A}, \mathtt{C}, \mathtt{E}\}$ with cost 23. This is the only way to obtain this product. From the second rule we obtain the cost transition $Stage_5$:

$$(\overline{\mathtt{D}}; \checkmark \wedge \checkmark, \mathtt{ACE}, 23) \rightarrow (\checkmark \wedge \checkmark, \mathtt{ACED}, 355)$$

As a result, we obtain the product $\{\mathtt{A}, \mathtt{C}, \mathtt{D}, \mathtt{E}\}$ with cost 355. In this case this is not the only way to produce it because if we go back to the cost transition $Stage_6$, by applying rule [**con2**], we obtain the cost transition:

$$(\checkmark \wedge \mathtt{E}; \checkmark, \mathtt{ACD}, 34) \rightarrow (\checkmark \wedge \checkmark, \mathtt{ACDE}, 615)$$

In this case, we obtain the product $\{\mathtt{A}, \mathtt{C}, \mathtt{D}, \mathtt{E}\}$ with cost 615, and therefore, the $\mathtt{c}_{\mathtt{SPLA}}$ function is defined as follows:

p	$\mathtt{c}_{\mathtt{SPLA}}(P, p)$
$\{\mathtt{A}, \mathtt{B}\}$	$\{23\}$
$\{\mathtt{A}, \mathtt{C}, \mathtt{E}\}$	$\{23\}$
$\{\mathtt{A}, \mathtt{C}, \mathtt{E}, \mathtt{D}\}$	$\{355, 615\}$

Let us consider the SPLA terms R and Q defined as follows:

$$Q = \mathtt{A}; (\mathtt{B}; \checkmark \vee \mathtt{C}; (\overline{\mathtt{D}}; \mathtt{E}; \checkmark \vee \mathtt{E}; \checkmark)) \quad R = \mathtt{A}; (\mathtt{B}; \checkmark \vee \mathtt{C}; \mathtt{E}; \overline{\mathtt{D}}; \checkmark)$$

It is easy to check that $\mathtt{prod}(P) = \mathtt{prod}(Q) = \mathtt{prod}(R)$. However in P, the subterm $\overline{\mathtt{D}}; \checkmark \wedge \mathtt{E}; \checkmark$ indicates that there is no pre-established way to produce features E and D. On the contrary, this order has been established in Q and R. Since producing E before D is cheaper than producing D before E, we obtain:

$$R \leq_{\mathtt{c}} P \leq_{\mathtt{c}} Q$$

This indicates that the SPL represented by R is preferable to P and Q, and P is preferable to Q.

5. Opscode Chef run-list calculation

In this section we present an elaborated example extracted from a real context. This example shows how to calculate, using our formal framework, the minimum cost run-list for the nodes inside a Chef server infrastructure given an organization context.

5.1. Product line organization

The main purpose of this model is to create valid run-lists for a set of Chef nodes. The organization requirements are described next.

- There are two environments: production environment and test environment. In order to model this condition we need to add constraints to our variability model, which are taken from the system configuration. The syntax of the environment description is shown below:

```
name "environment_name"
description "environment_description"
cookbook OR cookbook_versions  "cookbook" OR "cookbook" => "cookbook_version"
default_attributes "node" => { "attribute" => [ "value", "value", "etc." ] }
override_attributes "node" => { "attribute" => [ "value", "value", "etc." ] }
```

  In this case we can set up the environment requirements, which overrides attributes from others recipes. The `Production` environment is presented as follows:

```
name "production"
description "Production␣environment"
cookbook_versions({
    "nginx" => "<=␣1.1.0",
    "mysql" => "<=␣4.2.0",
    "apache" => "=␣0.4.1"
})
override_attributes ({
    "apache" => {
        "listen" => [ "80", "443" ]
    },
    "mysql" => {
        "root_pass" => "XXXXX"
    }
})
```

  The `Test` environment is shown below:

```
name "test"
description "Test␣environment"
cookbook_versions({
    "nginx" => "<=␣2.1.0",
    "mysql" => "<=␣6.2.0",
    "apache" => "=␣1.4.1"
})
override_attributes ({
    "apache" => {
        "listen" => [ "8080", "4430" ]
    },
    "mysql" => {
        "root_pass" => "XXXXX"
    }
})
```

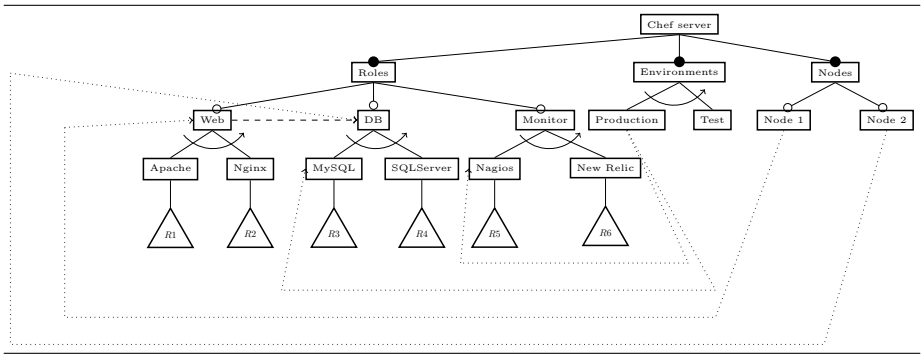

Figure 6: FODA Opscode Chef feature diagram.

- There are 2 nodes: a Web server and a database server for each environment. In order to model this condition, once we have our Chef system configured, we bootstrap the chef client as follows:

```
knife bootstrap web.example.com -r 'role[webserver]' -e Production
knife bootstrap db.example.com -r 'role[database]' -e Production
```

  This configures the recipes from the *webserver* role into web.example.com by using the attributes from the *Production* environment. Similarly, it sets db.example.com as a database server in the production environment.

- The nodes have to be monitored. In order to achieve this, we need to change the bootstrap command to:

```
knife bootstrap web.example.com -r 'role[webserver]','role[monitor]' -e Production
knife bootstrap db.example.com -r 'role[database]','role[monitor]' -e Production
```

  In this case we add the *monitor* role to the Chef nodes.

- The Web server and the database server must run in separate physical nodes. This condition is inherent to the user requirements. This will be achieved only by keeping the Web server and the database roles separate from each other when the Chef client is being bootstrapped.

- In the production environment the database server must be MySQL and the monitor software must be Nagios. In order to achieve this, we can access environmental variables from recipes. Once the variables are set, we are able to execute the right actions.

```
if node.chef_environment == "production"
   # do something, i.e. Install MySQL, Nagios.
else
  # do other thing.
end
```

Also, we need to model how variability and commonality interact with all functional and non-functional components inside Chef architecture. Given the Chef architecture presented in Figure 2, we take the relationships that we need to model into the product line environment.

- A Chef system can have 0 or more Roles, 0 or more cookbooks and at least 1 environment.
- A Node can have only 1 environment.
- A Node can have 0 or more Roles.
- A Node can have 0 or more cookbooks.
- A Role can have 0 or more cookbooks.
- A Node has only 1 Run-list.

```
#
# Cookbook Name:: installers_custom
# Recipe:: sqlserver
#
#Steps
#1- Download ISO according OS (w2008 or w2012) if not downloaded.
#2- Unzip ISO if not unzipped.
#3- Install SQL Server if not installed.
#
#Step 1
case node['platform']
when 'windows'
        require 'chef/win32/version'
        windows_version = Chef::ReservedNames::Win32::Version.new
        if windows_version.windows_server_2008_r2?
                unless File.exists?('C:/en_sql_server_2008.iso')
                remote_file 'C:/en_sql_server_2008.iso' do
                                source 'http://software.example.com/en_sql_server_2008.iso'
                                owner 'Administrator'
                        end
        end

                #Step 2                unless File.exists?('C:/SQL_SERVER')
                        batch 'unzip SQL Server' do
                                code <<-EOH
                                7zG x -y "C:/en_sql_server_2008.iso" -o"C:/SQL_SERVER"
  .
  .
  .

#Step 3
if File.exists?('C:/SQL_SERVER')
        batch 'unzip SQL Server' do
                code <<-EOH
                "C:/SQL_SERVER/setup.exe" /QUIET
                EOH
        end
end
```

Thus, in order to create a variability model, we merge these relationships and conditions. This information has been translated directly to a FODA diagram (see Figure 6). This diagram shows the run-lists $R1$, $R2$, $R3$, $R4$, $R5$ and $R6$. For

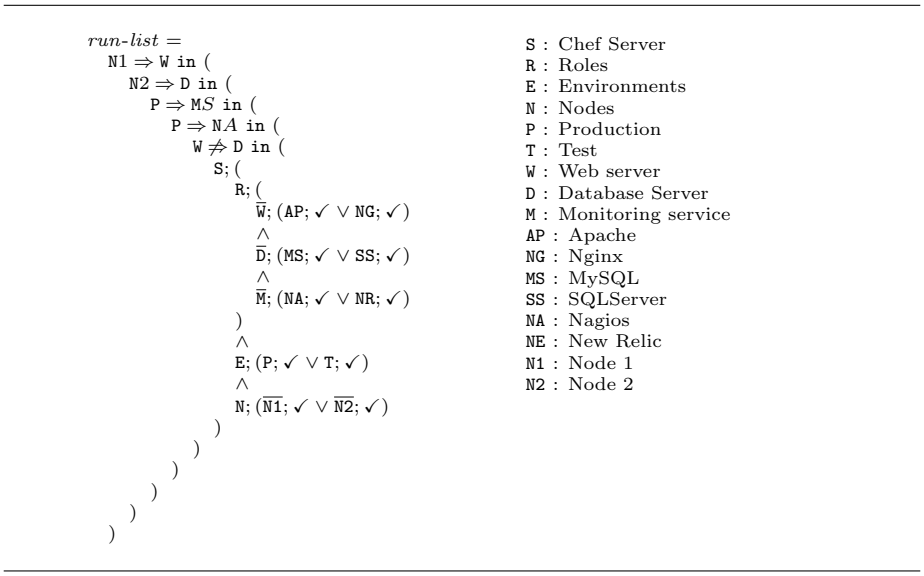

Figure 7: FODA to SPLA term.

instance, the possible content of $R4$ having the instructions to install SQLServer, depends on the operating system configured in previous listing. These run-lists are a sequential list of mandatory recipes that do not have any influence in the variability model, and therefore, these run-lists have to be removed from our analysis. These run-list will be taken into account when the cost model is analyzed.

The SPLA specification has been gathered from the diagram shown in Figure 6. Thus, this specification can be translated for creating the model defined in Figure 7. Moreover, the ChefServer, Roles, Cookbooks and Environments are parts of the domain analysis of the SPL, but they do not necessarily affect the functional behavior of the products as they represent category features. Consequently, they can be removed without changing the variability model. The term with this simplification is obtained from Figure 7. This simplification have a great impact on the overall performance of our tool, as we will see in Section 6.

5.2. Features cost specification

Once the variability model of our example has been defined, we are able to build the cost model. For the sake of simplicity we describe the model for the database node inside the Production environment.

The cost model is defined by taking into account the complexity of the recipes associated to a node. The variability model does not restrict the order in which the features Nagios and MySQL are introduced in the product. However, this order is important. For example, if we introduce the feature Nagios before the feature MySQL, the following process is executed: the feature MySQL must be

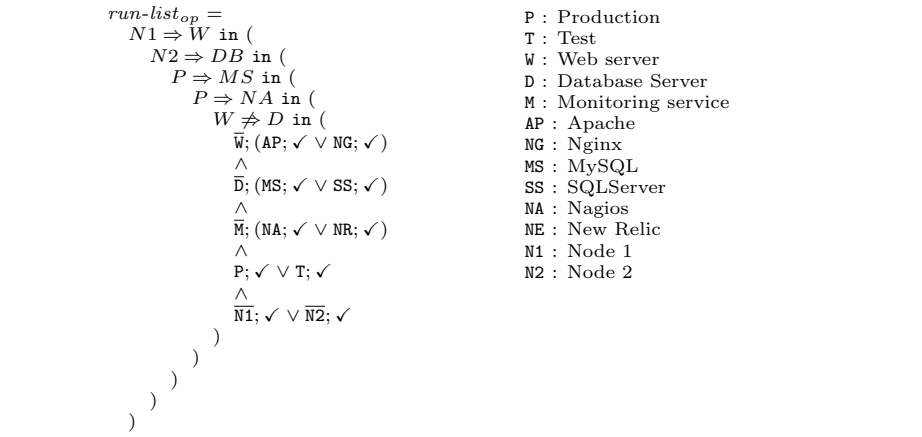

Figure 8: SPLA optimized term.

monitored, once the feature MySQL is introduced, the recipes associated to the feature Nagios must be recalculated. On the contrary, these recipes do not have to be recalculated if we introduce the feature MySQL before the feature Nagios. Similarly, the same occurs with features Nagios and Apache. Finally, we reflect the fact that the Web server and the Database server must be in different nodes by indicating that the cost of introducing the feature Apache (resp. MySQL) when the feature MySQL (resp. Apache) has previously been included is $\perp$.

$$\mathtt{c}(s,x) = \begin{cases} Rule1 : 7 & \text{if } x = \mathtt{MySQL} \text{ and } \mathtt{Nagios} \in s \\ Rule2 : 7 & \text{if } x = \mathtt{Apache} \text{ and } \mathtt{Nagios} \in s \\ Rule3 : 4 & \text{if } x = \mathtt{Nagios} \text{ and } \mathtt{MySQL} \in s \\ Rule4 : 4 & \text{if } x = \mathtt{Nagios} \text{ and } \mathtt{Apache} \in s \\ Rule5 : \perp & \text{if } x = \mathtt{Apache} \text{ and } \mathtt{Mysql} \in s \\ Rule6 : \perp & \text{if } x = \mathtt{Mysql} \text{ and } \mathtt{Apache} \in s \\ Rule7 : 0 & \text{otherwise} \end{cases}$$

- [Rule1:] MySQL is being computed and Nagios was already computed. This rule will determine that 7 recipes must run in order to have the MySQL service monitored with Nagios, as the monitoring software was previously installed.
- [Rule2:] Apache is being computed and Nagios was already computed. Same cost as rule Rule1, 7 recipes must run in order to have the Apache service monitored.
- [Rule3:] Nagios is being computed and MySQL was already computed. In this case the execution is less expensive as we need to run only 4 recipes

to have the MySQL service monitored.

- [Rule4:] `Nagios` is being computed and `Apache` was already computed. The same cost as rule Rule3 as we need to run only 4 recipes to have the Apache service monitored.
- [Rule5 and Rule6:] The Web server and database server must be in different nodes.
- [Rule7:] The cost of producing any other feature in any other order is 0.

5.3. Execution and computing results

This section describes how a `SPLA` term is computed. This example generates a trace tree containing all valid and invalid states. These traces cannot be graphically represented in this paper due to their size. In this case we show some representative traces of `tr`(*run-list*) for the node containing the database in the production environment.

$$\texttt{tr}(\textit{run-list}) = \left\{ \begin{array}{l} \ldots \\ \texttt{['Node2', 'Prod', 'DB', 'MySQL', 'Monitor', 'NewRelic', 'Nagios']} \\ \texttt{['Node2', 'Prod', 'Monitor', 'DB', 'Nagios', 'MySQL']} \\ \ldots \\ \texttt{['Node2', 'Prod', 'Monitor', 'DB', 'NewRelic', 'MySQL', 'Nagios']} \\ \ldots \end{array} \right\}$$

Once the costs of the traces have been calculated, we obtain the corresponding products with their associated costs.

p	$\mathtt{c}_{\mathtt{SPLA}}(P,p)$
...	...
{Node2, Prod, DB, MySQL, Monitor, NewRelic, Nagios}	$\{4,7\}$
{Node2, Prod, Monitor, DB, Nagios, MySQL}	$\{4,7\}$
...	...

It is important to remark that the run-list creation must be idempotent. This means that no matter how many times we execute chef-client, the system output is always the same. For instance, if a recipe install MySQL, no matter how many times you execute chef-client, the first time it is executed, MySQL is installed and subsequent executions will be omitted as the node is already in a desired state. This in the case when it is done in a natural way as this problem is treated as a deterministic state machine.

6. Implementation

This section presents our implementation that deals with the formal framework presented in this paper. The tool can be downloaded from the project main page `http://simba.fdi.ucm.es/at`. This software is licensed under GPL v3 (more details in `http://www.gnu.org/copyleft/gpl.html`).

This tool has been developed completely in Python and it generates a Finite State Machine to model all valid states. While features are computed, the cost and the trace are stored in each term for further use. The file `http://simba.fdi.ucm.es/at/splacris/splacris.zip` contains the API documentation, a README.txt file with instructions on how to use the program, example files and the core tool.

6.1. Considerations

Our implementation suffers one of the classical computational problems of this kind of tools: the calculation of all possible valid traces from the variability model. One of the main problems is the behavior of the *parallel* operator $\wedge$: [**con1**], [**con2**], [**con3**] in the operational semantics.

The traces generated from $P \wedge Q$ must have the features of P and Q in all possible combinations. This creates a combinatorial explosion of traces, since the combinations of K features without repetition has $K!$ different combinations.

In order to reduce the effect of this combinational explosion, there are some considerations that can be taken into account when the product line is being designed in order to optimize the execution time.

A way to handle this expected behavior is to prune the traces tree while the terms are being computed. This can be achieved by using several approaches, like *Costs Thresholds*. In this case, products cannot be produced with costs higher than HC or lower than LC. We can also prune traces by adding bottoms ($\perp$) in the *Cost function* so forth those traces can never end in a valid product. Also the terms are independent as they have the cost information inside them, hence the application can be developed to support distributed computing, in order to share the computing effort in a distributed system.

6.1.1. Feature model optimization

`FODA` diagrams have a very strict representation in which all functional and non-functional parts of the product line organization must be represented. When we compute a parallel operator in an `SPLA` term, there is a performance penalty each time a parallel feature is included in the model. This is an inherent condition of the use of the parallel operator in the `SPLA` operational semantics.

Thus, using our formal framework we increase the model's flexibility as we are not forced to have optional or mandatory features inside a parallel relationship, we can also have different `SPLA` terms inside. These types of conditions enable us to optimize the number of features inside a feature diagram. Let us recall the example in Section 5. We first presented the first term *run-list* in Figure 7 that results from the `FODA` diagram in Figure 6. As we have previously presented, the products of this term contain the features `ChefServer`, `Roles`,

`Cookbooks` and `Environments` that are part of the domain analysis of the `SPL`. However, they do not necessarily affect the functional behavior of the products as they represent categories of features. Thus, we presented the `SPLA` term *run-list*$_{op}$ in Figure 8 where these features are removed. The set of products of both terms are essentially the same because we removed only features that represent categories.

This simple modification in the model has great impact on performance. The computation of the term *run-list* generates 97648 valid products taking 2340.49 seconds, while the computation of the *run-list*$_{op}$ generates 10572 valid products taking 491.41 seconds. This result shows that a correct analysis of the product line organization can affect directly the performance of the algorithm execution. In this particular case by removing these non-functional features the time performance is increased by 79% and the number of products computed are reduced by 89,1%.

6.1.2. Distributed computing

`SPLA` terms can be computed in a distributed system in a natural way. In this case all system states can be treated as separate components because their computation does not rely on other processing terms. For instance, let us consider a term like the following:

$$P = (F1; P2) \vee (F3; P4) \vee (F2; ((F5; P5) \vee (F6; P6)))$$

where $P2$, $P4$, $P5$ and $P6$ are valid `SPLA` terms. These terms may represent a terminal state in which a valid trace will be computed. Computing these terms in a distributed system can be achieved in a natural way as all terms are independent from each other. Also, if we store in the object that represents a `SPLA` instance the term, the previously computed trace and the cost to compute that trace, then we have self-contained all the information to distribute the computing effort among a distributed system. We refer to the implementation, available at `http://simba.fdi.ucm.es/at/splacris/splacris.zip`.

In order to execute our implementation in a distributed system we use SCOOP (Scalable COncurrent Operations in Python) [49] which is a Python framework for automatically distributing dynamic tasks. In this case, we recursively spawn a list of substates into a set of SCOOP workers, which are automatically distributed among the available resources using dynamic load balancing. A worker is a computing instance used to execute a task. Multiple workers are distributed in a computing pool in order to complete a task and in one computing node can be one or more workers.

We have tested our implementation in an 8-node cluster, where each node has the following configuration:

Operating System: Ubuntu 14.04-trusty

RAM: 4GB

Processor: Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz

	1 Worker	2 Workers	4 Workers	8 Workers	16 Workers	32 Workers
1 Node	2010.44763303	1060.80047798	586.014445066	543.712262154	521.616870165	521.292215109
2 Nodes	2059.06947899	1046.87119007	575.115453959	296.285589933	366.384442091	341.007520199
4 Nodes	2031.25184584	1093.52087283	586.344302893	320.010899067	300.745616913	382.748430014
8 Nodes	2207.35427213	1143.951792	576.370896101	495.507214785	308.374300957	287.340030909

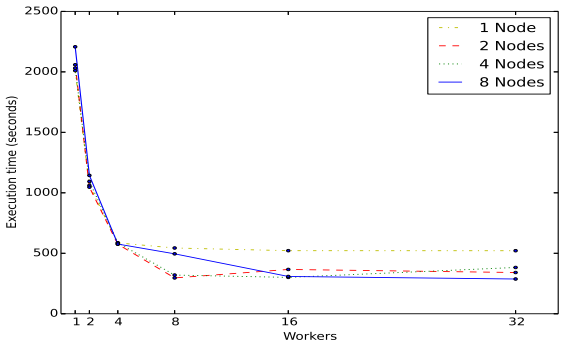

Figure 9: Cluster execution times for 1, 2, 4 and 8 nodes; and 1, 2, 4, 8, 16 and 32 workers.

CPU cores: 4

The `SPLA` term used to be computed with our distributed implementation is *run-list* (Figure 7). This term generates a total of 97648 valid products. Figure 9 shows the execution times when computing all the products for *run-list* for all the different configuration between workers and nodes. The best results are obtained by using 8 nodes and 32 workers (287.34 seconds), although the gain is not very relevant with respect to simpler configurations. For instance, the gain of the configuration with 8 nodes and 32 workers with respect to the one with 4 nodes and 16 workers is only 13.41 seconds (4.67% of performance improvement).

6.1.3. Thresholds

Thresholds can be used as a solution to minimize the effort to compute an `SPLA` term. In this case, we are able to discard a trace that is beyond the threshold while we are computing it. For example, let us consider the term

$$P = \overline{\texttt{F1}}; \checkmark \wedge \overline{\texttt{F2}}; \checkmark \wedge \overline{\texttt{F3}}; \checkmark \wedge \overline{\texttt{F4}}; \checkmark \wedge \overline{\texttt{F5}}; \checkmark \wedge \overline{\texttt{F6}}; \checkmark)$$

where the cost of producing a feature is always 1. In this case the total cost of a product will be equal to the number of features of the product.

In this example P generates 1957 valid products taking a total execution time of 6.02 seconds. These results are presented in Figure 10 (left), without taking care about thresholds.

Next, we define a minimum threshold of 1 and a maximum threshold of 2. Due to we defined that the cost of computing a feature is equal to 1, only products with one or two features are generated. In this case, the number of

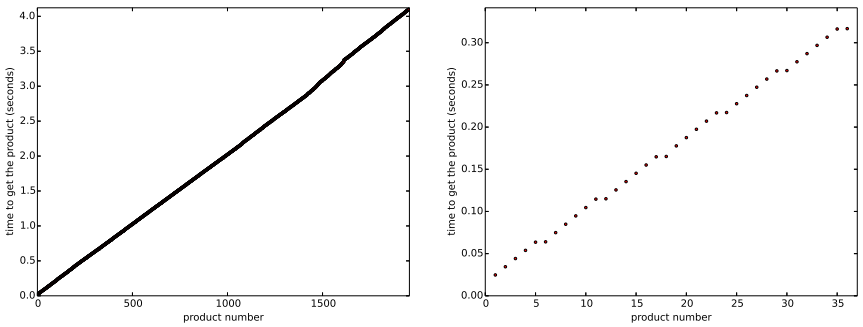

Figure 10: Cumulative time graphic for computing products without thresholds (left) and with thresholds Min=1, Max=2 (right).

valid products is 36, which it is displayed in Figure 10 (right) with an execution time of 0.45 seconds.

It is important to note the spaces between the dots in Figure 10 (right). These spaces correspond to the time spent to generate the next valid trace (product). In order to generate a trace, the production cost must be between 1 and 2, while all the other products will not be valid. Once the production cost exceed the threshold, the computing of that branch will end and no more product will be computed from there.

7. Conclusions and Future work

In this paper, we have extended the basic `SPLA` representation to handle costs. Benefits from this extension can be applied in many decision-making processes. An important characteristic of this extension is that the order in which the features are introduced is important. Thus, products with the same features can have different costs due to the ordering of features. This characteristic can be very relevant for a number of software projects.

We have presented a tool that supports the theory in this paper. This tool has the typical combinatorial problems of the *parallel* operator in process algebras. We have outlined ways to optimize future implementations and we have shown practical examples in which `SPLA` flexibility helps to optimize the product line.

Our future research work includes the introduction of probabilities. We think that the inclusion of probabilities opens a wide range of applications to `SPLA`. For instance, let us consider an SPL where the features correspond to software modules and the probability indicates the probability of a feature being included in a product. Thus, it is reasonable to spend more time by testing the modules with greater probabilities.

The transitions of our model are similar to those in [30], so it would be possible to use mature analysis tools for model checking as Uppaal Cora [33]. However, the main problem is the possible state explosion when transforming terms from our formalism into a priced automaton. A way to achieve this is by using simulation semantics as we indicate in the following paragraph.

Process algebras handle situations in which quantitative analysis on dynamic properties are studied [38]. We take the advances of process calculi to model a novel characteristic on SPLs, the order in which features are computed. This will allow to model conditions in which the final cost of the product depends on the order of the computed features.

During last decades a huge amount of results have been developed in the process algebra community. We hope that some of these works can be applied in the analysis of SPLs. In a recent work [50], the authors have shown that the number of states of a specification can be dramatically reduced when a simulation relation is applied. We are planning to adapt this result in our context: We think that it could be possible to generate a simplified version of an SPLs that simulates the original one; then other kind of analysis could be performed in the simplified version.

8. References

[1] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, Feature-Oriented Domain Analysis (FODA) feasibility study, Tech. Rep. CMU/SEI-90-TR-21, Carnegie Mellon University (1990).

[2] M. Griss, J. Favaro, Integrating feature modeling with the RSEB, in: 5th International Conference on Software Reuse, ICSR'98, 1998, pp. 76–85. doi:10.1109/ICSR.1998.685732.

[3] K. Kollu, Evaluating the pluss domain modeling approach by modeling the arcade game maker product line, Ph.D. thesis, Umea University (2005).

[4] M. Eriksson, J. Borstler, K. Borg, The pluss approach - domain modeling with features, use cases and use case realizations, in: 9th International Conference on Software Product Lines, SPLC'06, Springer-Verlag, 2006, pp. 33–44. doi:10.1007/11554844_5.

[5] J. Sun, H. Zhang, H. Wang, Formal semantics and verification for feature modeling, in: 10th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS'05, IEEE Computer Society Press, 2005, pp. 303–312. doi:10.1109/ICECCS.2005.48.

[6] P. Höfner, R. Khédri, B. Möller, Feature algebra, in: 14th International Symposium on Formal Methods, FM'06, Vol. 4085 of LNCS, Springer, 2006, pp. 300–315. doi:10.1007/11813040_21.

[7] P. Höfner, R. Khédri, B. Möller, An algebra of product families, Software and System Modeling 10 (2) (2011) 161–182. doi:10.1007/s10270-009-0127-2.

[8] C. Andres, C. Camacho, L. Llana, A formal framework for software product lines, Information and Software Technology 55 (11) (2013) 1925–1947. doi:10.1016/j.infsof.2013.05.005.

[9] M. Mannion, Using first-order logic for product line model validation, in: 2nd International Software Product Line Conference, SPLC'02, Springer, 2002, pp. 176–187.

[10] K. Czarnecki, A. Wasowski, Feature diagrams and logics: There and back again, in: 11th International Software Product Line Conference, SPLC'07, IEEE Computer Society Press, 2007, pp. 23–34. doi:10.1109/SPLC.2007.19.

[11] P. Asirelli, M. H. ter Beek, S. Gnesi, A. Fantechi, A deontic logical framework for modelling product families, in: 4th International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS'10, 2010, pp. 37–44.

[12] P. Asirelli, M. H. ter Beek, A. Fantechi, S. Gnesi, A logical framework to deal with variability, in: 8th International Conference on Integrated Formal Methods, IFM'10, Springer, 2010, pp. 43–58. doi:10.1007/978-3-642-16265-7_5.

[13] J. Nummenmaa, T. Nummenmaa, Z. Zhang, On the use of ltss to analyze software product line products composed of features, in: F. Sun, T. Li, H. Li (Eds.), Knowledge Engineering and Management, Vol. 214 of Advances in Intelligent Systems and Computing, Springer Berlin Heidelberg, 2014, pp. 531–541. doi:10.1007/978-3-642-37832-4_48.

[14] S. Nakajima, Semi-automated diagnosis of foda feature diagram, in: 25th ACM Symposium on Applied Computing, SAC'10, ACM Press, 2010, pp. 2191–2197. doi:10.1145/1774088.1774550.

[15] Y. Bontemps, P. Heymans, P. Schobbens, J. Trigaux, Semantics of FODA feature diagrams, in: 1st Workshop on Software Variability Management for Product Derivation – Towards Tool Support, SPLCW'04, Springer, 2004, pp. 48–58.

[16] V. Nguyen, S. Deeds-rubin, T. Tan, B. Boehm, A sloc counting standard, in: COCOMO II Forum 2007, 2007, pp. 1–16.

[17] B. Boehm, C. Abts, A. Brown, S. Chulani, Software Cost Estimation with COCOMO II, 1st Edition, Prentice Hall Press, 2009.

[18] M. Jorgensen, A review of studies on expert estimation of software development effort, Journal of Systems and Software 70 (1-2) (2004) 37–60. doi:10.1016/S0164-1212(02)00156-5.

[19] C. Gencel, How to use cosmic functional size in effort estimation models?, in: Software Process and Product Measurement, Vol. 5338 of LNCS, Springer Berlin Heidelberg, 2008, pp. 196–207. doi:10.1007/978-3-540-89403-2_17.

[20] I. Hussain, L. Kosseim, O. Ormandjieva, Approximation of cosmic functional size to support early effort estimation in agile, Data Knowl. Eng. 85 (2013) 2–14. doi:10.1016/j.datak.2012.06.005.

[21] J. Robbins, A. Jacob, Opscode chef, http://www.getchef.com/ (2014).

[22] H. Schackmann, H. Horst, A cost-based approach to software product line management, in: Proceedings of the International Workshop on Software Product Management, IWSPM '06, IEEE Computer Society Press, 2006, pp. 13–18. doi:10.1109/IWSPM.2006.1.

[23] B. Boehm, A. Brown, R. Madachy, Y. Yang, A software product line life cycle cost estimation model, in: Empirical Software Engineering, 2004. ISESE 04. Proceedings. 2004 International Symposium on, 2004, pp. 156–164. doi:10.1109/ISESE.2004.6.

[24] A. Nolan, S. Abrahao, Dealing with cost estimation in software product lines: Experiences and future directions., in: SPLC, Vol. 6287 of LNCS, Springer, 2010, pp. 121–135.

[25] G. Bockle, P. Clements, J. McGregor, D. Muthig, K. Schmid, A cost model for software product lines, in: F. Linden (Ed.), Software Product-Family Engineering, Vol. 3014 of LNCS, Springer Berlin Heidelberg, 2004, pp. 310–316. doi:10.1007/978-3-540-24667-1_23.

[26] N. Anquetil, U. Kulesza, R. Mitschke, A. Moreira, J.-C. Royer, A. Rummler, A. Sousa, A model-driven traceability framework for software product lines, Software and Systems Modeling 9 (4) (2010) 427–451. doi:10.1007/s10270-009-0120-9.

[27] T. K. Satyananda, D. Lee, S. Kang, S. I. Hashmi, Identifying traceability between feature model and software architecture in software product line using formal concept analysis, in: Proceedings of the The 2007 International Conference Computational Science and Its Applications, ICCSA '07, IEEE Computer Society, Washington, DC, USA, 2007, pp. 380–388. doi:10.1109/ICCSA.2007.47.

[28] K. Czarnecki, K. Pietroszek, Verifying feature-based model templates against well-formedness ocl constraints, in: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, GPCE '06, ACM, New York, NY, USA, 2006, pp. 211–220. doi:10.1145/1173706.1173738.

[29] D. Batory, D. Benavides, A. Ruiz, Automated analysis of feature models: challenges ahead, Communications of the ACM 49 (2006) 45–47. doi:10.1145/1183236.1183264.

[30] G. Behrmann, K. G. Larsen, J. I. Rasmussen, Optimal scheduling using priced timed automata, SIGMETRICS Performance Evaluation Review 32 (4) (2005) 34–40. doi:10.1145/1059816.1059823.

[31] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, F. Vaandrager, Minimum-cost reachability for priced time automata, in: M. Di Benedetto, A. Sangiovanni-Vincentelli (Eds.), Hybrid Systems: Computation and Control, Vol. 2034 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2001, pp. 147–161. doi:10.1007/3-540-45351-2_15.

[32] R. Alur, S. La Torre, G. Pappas, Optimal paths in weighted timed automata, in: M. Di Benedetto, A. Sangiovanni-Vincentelli (Eds.), Hybrid Systems: Computation and Control, Vol. 2034 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2001, pp. 49–62. doi:10.1007/3-540-45351-2_8.

[33] Uppaal cora, http://people.cs.aau.dk/~adavid/cora/index.html (2005).

[34] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, S. Yovine, Kronos: A model-checking tool for real-time systems, in: A. Ravn, H. Rischel (Eds.), Formal Techniques in Real-Time and Fault-Tolerant Systems, Vol. 1486 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1998, pp. 298–302. doi:10.1007/BFb0055357.

[35] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, W. Yi, M. Hendriks, Uppaal 4.0, in: Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems, QEST '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 125–126. doi:10.1109/QEST.2006.59.

[36] S. Nishizaki, H. Kiyoto, Formal framework for cost analysis based on process algebra, in: M. Zhao, J. Sha (Eds.), Communications and Information Processing, Vol. 288 of Communications in Computer and Information Science, Springer Berlin Heidelberg, 2012, pp. 110–117.

[37] H. Kiyoto, S. Nishizaki, Extended process algebra for cost analysis, Foundations of Computer Science and Technology 2 (2) (2012) 197–214. doi:10.5121/ijfcst.2012.2201.

[38] Process Algebras for Quantitative Analysis.

[39] P. Buchholz, P. Kemper, Quantifying the dynamic behavior of process algebras, in: PAPM-PROBMIV 2001, Aachen, Vol. 2165 of LNCS, 2001, pp. 184–199. doi:10.1007/3-540-44804-7_12.

[40] J. Bosch, Design and use of software architectures: adopting and evolving a product-line approach, Addison-Wesley, 2000.

[41] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, J. Zettel, Component-based product line engineering with UML, Addison-Wesley, 2002.

[42] C. Krzysztof, H. Simon, W. Ulrich, Staged configuration through specialization and multilevel configuration of feature models, Software Process: Improvement and Practice 10 (2) (2005) 143–169.

[43] D. Batory, Feature models, grammars, and propositional formulas, in: 9th International Software Product Line Conference, SPLC'05, Springer, 2005, pp. 7–20. doi:10.1007/11554844_3.

[44] K. Czarnecki, S. Helsen, Feature-based survey of model transformation approaches, IBM Systems Journal 45 (3) (2006) 621–646.

[45] P. Heymans, P. Schobbens, J. Trigaux, Y. Bontemps, R. Matulevicius, A. Classen, Evaluating formal properties of feature diagram languages, IET Software 2 (3) (2008) 281–302.

[46] M. Mendonça, A. Wasowski, K. Czarnecki, SAT-based analysis of feature models is easy, in: 13rd International Software Product Line Conference, SPLC'09, 2009, pp. 231–240. doi:10.1145/1753235.1753267.

[47] D. Benavides, S. Segura, A. Ruiz, Automated analysis of feature models 20 years later: A literature review, Information Systems 35 (6) (2010) 615–636. doi:10.1016/j.is.2010.01.001.

[48] P. Y. Schobbens, P. Heymans, J.-C. Trigaux, Y. Bontemps, Generic semantics of feature diagrams, Computer Networks 51 (2) (2007) 456–479. doi:10.1016/j.comnet.2006.08.008.

[49] Y. Hold-Geoffroy, O. Gagnon, M. Parizeau, Once you scoop, no need to fork, in: Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment, ACM, 2014, p. 60. doi:10.1145/2616498.2616565.

[50] C. Gregorio-Rodríguez, L. Llana, R. Martínez-Torres, Extending mcrl2 with ready simulation and iocos input-output conformance simulation, in: SAC '15, ACM, 2015, to appear.