Modelling and Analysing Highly-Configurable Services

Jesús García-Galán
Lero - The Irish Software Research Centre
Limerick, Ireland
Jesus.Galan@lero.ie

José María García
Universidad de Sevilla
Sevilla, Spain 41012
josemgarcia@us.es

Pablo Trinidad
Universidad de Sevilla
Sevilla, Spain 41012
ptrinidad@us.es

Pablo Fernández
Universidad de Sevilla
Sevilla, Spain 41012
pablofm@us.es

ABSTRACT

Since the emergence of *XaaS* and Cloud Computing paradigms, the number and complexity of available services have been increasing enormously. These services usually offer a plethora of configuration options, which can even include additional services provided as a bundled offer. In this scenario, usual tasks, such as description, discovery and selection, become increasingly complex due to the variability of the decision space. The notion of Highly-Configurable Service (HCS) has been coined to identify such group of services that can be configured and bundled together to perform demanding computing tasks. In this paper we characterize HCSs by means of an abstract model and a text-based, human-readable notation named SYNOPSIS that facilitates the execution of various service tasks. In particular, we validate the usefulness of our model when checking the validity of HCSs descriptions in SYNOPSIS, as well as selecting the optimal configuration with regards to user requirements and preferences by providing a prototype implementation.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Software product lines**; *Cloud computing*; *Specification languages*; Abstraction, modeling and modularity;

KEYWORDS

Configurable Services, Service Modelling, Validity Checking, Service Selection, Automated Analysis

ACM Reference format:
Jesús García-Galán, José María García, Pablo Trinidad, and Pablo Fernández. 2017. Modelling and Analysing Highly-Configurable Services. In *Proceedings of 21st International Conference on Software Product Line, Seville, Spain, 25–29 September, 2017 (SPLC'17),* 9 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION AND MOTIVATION

A Highly-Configurable Service (HCS) can be intuitively defined as a service which offers multiple configuration options [15]. Some authors simply employ the term "configurable service" in related scenarios, such as configurator systems [13], configurable processes in the cloud [21], and configuration of real-world services [2, 8], to name a few. However, to the extent of our knowledge, there is no clear and precise definition of what an HCS is. The number of actual configurations (a.k.a. decision space) provided by an HCS varies from just a few choices (e.g. 3 different plans in Dropbox) to thousands (16,991 possible configurations in Amazon EC2 [12]). Furthermore, some configurations may require the integration of additional services (e.g. some Amazon EC2 machines require the use of an EBS) or instances (load balancing for multiple virtual machines), resulting in an explosive growth in the number of configurations to be considered. This inherent complexity present in HCSs requires automated support and reasoning to leverage their configurability.

Variability in HCSs has been traditionally addressed in the same way as in other systems. Multiple authors have employed state-of-the-art Feature Models (FMs) to describe and reason on their configuration options [12, 17, 18, 23]. However, that requires service engineers and customers to speak in terms of variability and product lines rather than services and their characteristics. In addition, and even more importantly, such approaches are limited to single, atomic services, neglecting the inherent interplay of HCSs with other services within their ecosystem. In that sense, multi product lines [5, 6] have made some advances on the management of multiple and differing variability views. However, these approaches are still limited in terms of reasoning and do not properly cover the particularities of HCSs.

In this paper, we consider HCSs first-class citizens and propose a specific approach for their variability description, and analysis, which relies on existing FM reasoning support. First, we present SYNOPSIS, a Domain Specific Language (DSL) that enables the rigorous definition of HCSs. SYNOPSIS is focused on capturing the decision space while including interrelated concepts that play a fundamental role in relevant tasks during the service delivery workflow. Second, we introduce a catalogue of operations that enable the automated analysis of certain interesting properties when specifying HCSs, hence facilitating the automated execution of the aforementioned service tasks. These operations leverage existing configuration and automated reasoning support for Stateful Feature Models (SFMs) [19, 20], which extend traditional FMs with built-in configuration capabilities and have been already implemented and comprehensively tested [12]. For validating our solution we focus

SPLC'17, Seville, Spain

DOI: 10.1145/nnnnnnn.nnnnnnn

on two particular tasks: (1) validity checking for HCSs descriptions with respect to five different criteria, and (2) selection of the best configurations with respect to user needs.

The rest of the paper is structured as follows. In Section 2 we analyse existing approaches related to HCSs. Section 3 presents our proposed model and notation to characterise HCSs. Then, in Section 4 and Section 5 we discuss how to check the validity of SYNOPSIS documents and the definition of user needs, correspondingly. We present our automated solution to perform various analysis operations in Section 6. Based on that solution, we showcase our prototype implementation in Section 7. Finally, Section 8 discusses different aspects of our approach, and Section 9 concludes the paper and provide some insights on future work.

2 RELATED WORK

The intersection between variability and service-oriented computing has produced prolific research. Several authors [12, 18, 23] employ FMs to describe and analyse cloud services variability, in order to support user service configuration. Nguyen et al. [17] propose a feature-based framework for developing and maintaining customisable services, using FMs to capture variability and identify functional requirements for atomic services on the user side. Similarly, Walraven et al. [22] apply Software Product Line (SPL) principles to manage and leverage variability in multi-tenant Software-as-a-Service, enabling fine-grained service customisation. However, the semantic distance between FMs and HCSs remains significant, requiring service engineers to speak a language they are not familiar with. Furthermore, all these approaches consider atomic services, neglecting the interplay between different services and their potential multiple instances.

In the SPL community, the interplay between different variability sources has been considered with multi-SPLs [5, 6]. A multi-SPL encompasses multiple variability views of different and interconnected parts of the system, which may be described using differing notations. Acher et al. [1] propose the FAMILIAR DSL for the management of large scale SPLs, which also includes support for configuration, composition and reasoning. Galindo et al. [7] present Invar, an integrative approach for product configuration which supports different types of variability models and notations. However, existing reasoning support for multi-SPLs still target the specific parts and views of the system, instead of the system as a complete bundle.

Some works have considered the interplay of different services from the perspective of process-based service compositions. Nguyen et al. [16] present a process development methodology to explicitly consider composition variability in business process models, by using FMs and extending the Business Process Model Notation. Alférez et al. [3] go further and explore the adaptation of service compositions to deal with runtime variability. Nonetheless, this view focuses on putting services together to achieve a particular process or goal, neglecting the inherent relations among those services.

3 MODELLING HIGHLY-CONFIGURABLE SERVICES

In order to address the need for a precise definition and comprehensive model of HCSs, we propose a DSL that provides a complete support to describe HCSs and to perform automated analysis over those descriptions. Figure 1 presents our metamodel that organises and binds all the concepts that are relevant to HCSs domain.

Essentially, a service is described by a set of terms. In the case of a configurable service, some of these terms are decision terms[1] with at least two possible values. If the provider explicitly offer different alternatives for a decision term (e.g., disk type), we say the term is selectable, and derived in other case (e.g., cost). The different decision terms and their dependencies (i.e., constraints) conform the so-named *decision space* of the service. Such decision space encompasses all the available configurations, i.e. valid combinations of configurable term values.

When a configurable service interrelates additional configurable services, we consider the whole bundle of services as a Highly-Configurable Service. HCSs can potentially have multiple items (i.e., service instances), where each of these can have a different configuration (e.g., two disk drives with different capacity). While the types of the interrelated services must be known beforehand, their item cardinality can be unbound, letting consumers decide the exact number of items in their configuration (e.g., multiple disk drives associated to the same virtual machine).

As a concrete syntax to describe HCSs, we devised *SYNOPSIS* (SimplY a NOtation to sPecify Service configuratIonS), which is a text-based, human-readable notation to describe the decision space of services. It supports the specification of the configuration capabilities as modelled in Fig. 1, common to services of big providers such as Amazon, Rackspace or Microsoft, while remaining provider-agnostic. Listing 1 shows the SYNOPSIS description of a simple block storage service, which we will use as running example to explain our model and concrete notation in the following.

3.1 Configurable Services

A configurable service in SYNOPSIS has two sections to declare terms and dependencies. The terms are categorised into two groups: selectable (`%SelectableTerms` section) and derived (`%DerivedTerms` section). In the dependencies section (`%Dependencies`) we can describe the relationships among the different terms.

If some of the service terms have two or more possible values, i.e. they are configurable, we say the service is *configurable*, and hence its terms are *decision terms*. For example, Amazon EBS (a block storage service) and Spotify (a music streaming service) are both configurable services. These services can be contracted with different options, each providing specific values for the rest of decision terms. In the case of our simple storage block example, the GB cost per month depends on the region and the use of SSD. Consequently, both region and SSD are presented by the provider as direct choices, i.e. *selectable terms*, while the GB cost depends on the region and SSD values chosen, so it is a *derived term*.

[1] We employ configurable term and decision term as synonyms in this paper.

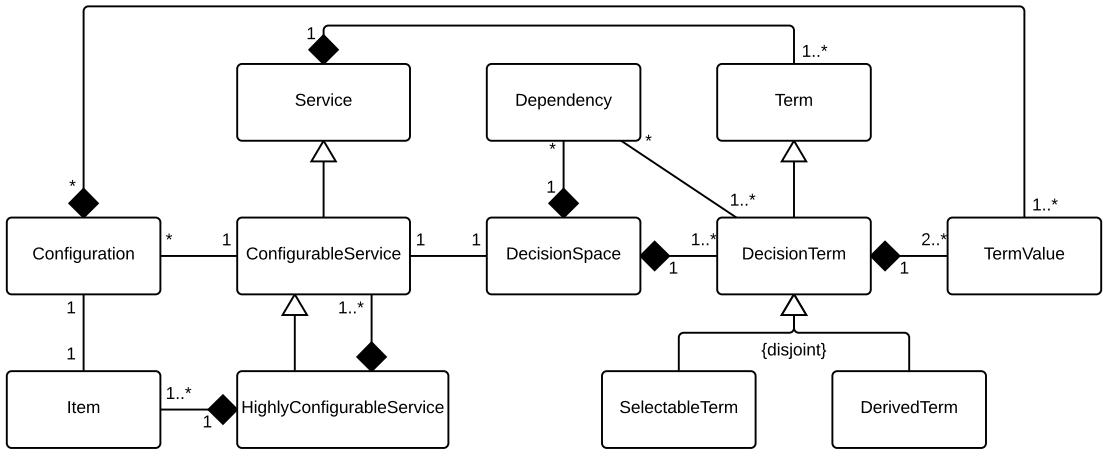


Figure 1: Highly-Configurable Services Metamodel.

Listing 1: Simple Block Storage Service in SYNOPSIS.

```
Service SimpleBlockStorage {
  %SelectableTerms
    SSD: boolean;
    Size: int [1,1000];
    Region: {"USA", "EU", "JP"};
  %DerivedTerms
    // euros/GB per month
    costGBMonth: real [0.00,0.15];
    // euros per month
    volumeCostMonth: real [0.00,150];
  %Dependencies
    // pricing
    volumeCostMonth == costGBMonth * Size;
    table (Region, SSD -> costGBMonth) {
      "USA", true   -> 0.1;
      "EU", true    -> 0.12;
      "JP", true    -> 0.15;
      "USA", false  -> 0.05;
      "EU", false   -> 0.06;
      "JP", false   -> 0.08;
    }
}
```

3.2 Decision Terms

Decision terms are declared in SYNOPSIS in the terms sections, denoted by the `%SelectableTerms` and `%DerivedTerms` tags for the two disjoint categories. The main difference between them is that selectable terms determine the different configurations of the service. In the case of the service of Listing 1, we consider that configurations are determined by the SSD, Region and Size terms. However, all these term types are described in the same way, using a declaration syntax close to programming languages. The only requirement is that each term should have at least two different term values.

Listing 1 shows the four different types of decision terms. For example, `SSD` is a boolean selectable term that indicates if the storage is ssd-based, `Size` is an integer selectable term that indicates the storage capacity, `Region` is an enumerated selectable term that declares the available regions of the service, and `costGBMonth` is a real derived term to describe the cost hour of the different configurations.

A user must choose one and only one term value for each selectable term to define a *configuration*. Thus, in our example a user can only choose a region among three options: U.S.A., E.U. and Japan, while SSD offers a boolean choice and the size of the storage has to be within the specified limits. The GB cost and volume cost per month are derived terms that are not configurable but their value changes depending on the said selectable terms.

3.3 Dependencies

Our model defines all the values that selectable and derived terms can take. However, not any combination of them is allowed and values can be bound in different forms, affecting to the way a service can be configured. For example, cost terms depend on the size, region and ssd terms. In order to represent such dependencies, we define a set of constraints that restricts the *decision space*.

SYNOPSIS provides a set of expressions and operators in order to define the dependencies in the decision space. They include the classic logical, relational and arithmetic operators, and also aggregation functions to relate HCS global terms to standard terms. Every dependency declared in SYNOPSIS should be logical, although it may be composed by other expression types. Table 1 summarises such expressions.[2,3]

Additionally, SYNOPSIS provides tables to declare groups of dependencies which involve the same terms with different values. Listing 1 shows an example of these tables, which are useful to describe, for instance, the pricing policies. In the table constructor we declare the configurable terms and the dependency relationship (from left to right, separated by the implication -> symbol). Each additional row provides the values for each term.

[2] $\mathbb{E}$ represents the set of all enumerated values of the document.
[3] sum, max and min functions aggregate standard terms into global terms.

Table 1: Dependency Expression Types for SYNOPSIS

Type	Expressions
Boolean	$B ::= b \mid t_b \mid B \,\&\&\, B \mid B \,\|\|\, B \mid !B \mid B \texttt{->} B \mid B \texttt{<->} B \mid I > I \mid I >= I \mid I < I \mid I <= I \mid I == I \mid I! = I \mid R > R \mid R >= R \mid R < R \mid R <= R \mid R == R \mid R! = R \mid E == E \mid E! = E$
Integer	$I ::= i \mid t_i \mid I + I \mid I - I \mid I * I \mid I/I \mid -I \mid I^I \mid sum(t_i) \mid max(t_i) \mid min(t_i)$
Real	$R ::= r \mid t_r \mid R + R \mid R - R \mid R * R \mid R/R \mid -R \mid sum(t_r) \mid max(t_r) \mid min(t_r)$
Enumerated	$E ::= e \mid t_e$

t_b any boolean term, t_i any integer term, t_r any real term, t_e any enumerated term.

$b \in \{true, false\}$, $i \in \mathbb{Z}$, $r \in \mathbb{R}$, $e \in \mathbb{E}$

3.4 Highly-Configurable Services

The configuration capabilities of a service may go further, by considering multiple service items (i.e., instances) of the service, and even additional linked services. This leads to the so-named Highly-Configurable Services (HCS). While some configurable services do not allow this – e.g. Dropbox or Spotify – others do, such as EC2 or Heroku. In the case of EC2, we can contract different computing instances of different types and in different regions, and even additional storage through the *Elastic Block Storage* (EBS) service (which resembles our example in Listing 1), all of them related to the same Amazon account. In the case of Heroku, we can also contract different Dynos and Postgres of different types. Different items of the same service may be interrelated by means of dependencies. For instance, Amazon provides a volume-based discount which depends on the total cost of all the items contracted.

Listing 2: Volume Storage HCS in SYNOPSIS.

```
Highly-configurable Service VolumeStorage{
  %Services
  storage: SimpleBlockStorage[1,*];

  %GlobalTerms
  totalCostMonth: real [0.00,10000.00];
  discount: real [0.00,1000.00];

  %Dependencies
  // cost aggregation
  totalCostMonth == sum(storage.volumeCostMonth);
  // discount policy
  totalCostMonth > 3000 ->
    discount == totalCostMonth*0.1;
}
```

Listing 2 presents an HCS in SYNOPSIS, which has three sections to declare the component services and their cardinality, global terms and global dependencies. For the two latter sections, the syntax is the same as for the configurable services described before. In the `%Services` section we declare the services that compose the HCS, their cardinality (lower and upper bounds for the items), and an alias to refer to it. The specific number of service items to hire is up to the consumer.

Global terms affect the whole aggregation of the configurable services in an HCS. This kind of term is necessary to describe, for instance, the total cost or the discount of an HCS, which depends on all the aggregated services. These terms are declared in the `%GlobalTerms` section of the HCS. Besides the standard operators for dependencies, global terms have available especial aggregation expressions (e.g. `sum`), already enumerated in Sec. 3.3.

Listing 2 shows a couple of HCS global terms: the total cost and the discount. The value of `totalCostMonth` is calculated based on the aggregation of the `volumeCostMonth` of each storage item, while the `discount` is calculated as a 10% of the total cost when it exceeds 3000 euros.

4 VALIDITY CRITERIA

A configurable service may present different anomalies regarding its configuration capabilities. For example, it is possible that some values of a configurable term cannot be selected under any circumstance, or that a configurable term is actually not configurable. In order to illustrate these anomalies, we define the validity criteria for configurable services using the Simple Block Storage Service of Listing 1 as a running example. The validity criteria are categorised in three levels, which in some cases resemble the well-known anomalies for traditional FMs [4]:

(1) **Warning level**, which encompasses anomalies that do not damage the configuration capabilities of the service (*redundant dependencies*);
(2) **Term error level**, which encompasses anomalies that damage the configuration capabilities of the service, and in particular of given values and terms (*dead values* and *false decision terms*). Regarding traditional FMs, dead values are equivalent to dead features, and false decision terms are equivalent to false optionals.
(3) **Service error level**, where the errors of this level make the service not configurable (*false configurable service*) or directly inconsistent (*inconsistent service*). Regarding FMs, an inconsistent service is equivalent to a void FM.

4.1 Warning Level

At the first validity level, i.e. the warning level, the anomalies detected do not affect the configuration capabilities of the service, but may complicate the understanding of the decision space. In particular, we have identified one possible anomaly at the warning level: the redundant dependency.

A *redundant dependency* has no effect on the decision space of the service. If such dependency is removed, the resultant decision space remains unaltered. For instance, the dependency `SSD == false -> costGBMonth < 0.1;` would be redundant for the storage service of Listing 1. The dependency says "if the disk is not SSD-based, the GB cost/month should be lower than 0.1 $", while at the same time we say in the pricing table of Listing 1 that the prices for non SSD-based volumes are 0.05, 0.06 and 0.08 per GB. In this way, such dependency does not modify the decision space, and can be classified as redundant.

4.2 Term Error Level

We name this second validity level as the term error level. Although at this level the service still presents multiple configurations, these errors damage its configuration capabilities. We identify two types

of errors that affect single values and terms: dead values and false decision terms.

A *dead value* in a selectable term is a value which cannot be selected under any circumstances, i.e. there is no configuration in the decision space where that value can be chosen. In this way, although the value can be apparently chosen, existing dependencies make it non selectable. In Listing 3 we show an example of dead values for the storage service of Listing 1. We find a constraint that denies the selection of USA as a region, making such value dead. At the same time, JP region is also dead, since only SSD-based instances can be selected in such location, but the cost/GB should be at most 0.12 $, while in the case of SSD is 0.15 $.

Listing 3: Example of Dead Value and False Decision Term

```
Region == "JP" -> SSD == true;
Region == "JP" -> costGBMonth <= 0.12;
Region != "USA";
```

If all the term values but one of a given decision term are dead, we say the term is a *false decision term*. Although the term can apparently be configurable, there is no possible decision: the consumer is forced to select the same particular value in every configuration in the decision space. Consequently, a false decision term makes all the remaining alternatives for its configuration option to be dead.

The example in Listing 3 also generates a false decision term. Given that the term `Region` only has three values, the death of two of them makes the term a false decision term. In this case, the consumer cannot choose among three regions, but has to select ''EU'' always.

4.3 Service Error Level

Finally, at the third validity level, namely service error level, the service presents one or none configurations, so consequently these errors are the most critical ones. We identify two types of service errors: false configurable service and inconsistent service.

A service is a *false configurable service* when there is only a single available configuration. In other words, all the decision terms of a false configurable service are false decision terms, i.e. there are no real choices since the decision space contains only one possible configuration. In Listing 4 we show a set of dependencies that make the Simple Block Storage Service a false configurable service. In this way, only one configuration can be selected: `Region == ''EU''`, `SSD == true`.

Listing 4: Example of False Configurable Service.

```
Region == "JP" -> SSD == true;
Region == "JP" -> costGBMonth <= 0.12;
Region != "USA";
SSD == false -> Region == "USA";
```

When all the values of a decision term are dead, we say that the service is an *inconsistent service*. This means that there is no available configuration for the service (i.e. its decision space is empty), and consequently it cannot be delivered to the consumer. For instance, if we add to the Volume Storage Service the additional dependency `costGBMonth >= 0.2`, the service becomes inconsistent due to the conflict with the pricing defined in the table.

5 DESCRIBING USER NEEDS FOR SELECTION

Automated selection of the most suitable configuration requires the description of the user needs, over concrete configurable services. In most of the cases, users only have to assign a value to a subset of these terms (usually selectable terms). The remaining terms depend on them, but are an important source of information in order to make a decision about the configuration that best suits their needs. So for example a user may make a decision about a Simple Block Storage service based on the GB cost.

The user needs can be defined as a set of constraints that helps to reduce the decision space to those values that satisfy the provided constraints. For this particular purpose, we extend SYNOPSIS notation in order to allow users to express such constraints. Listing 5 presents an example that includes most of the constructions of the language. As shown, a user can express needs in terms of service items, requirements and preferences.

Listing 5: User needs on the Simple Block Storage Service.

```
Needs on VolumeStorage{
  %Items
    storage["vol1","vol2"];
  %Requirements
    storage["vol1"].Size == 500;
    storage["vol1"].Region == "USA";
    storage["vol2"].Size >= 200;
  %Preferences
    Favorites(storage["vol2"].SSD);
    Dislikes(storage["vol2"].Region, "JP");
    Lowest(VolumeStorage.totalCostMonth);
}
```

In the header of the user needs description, the `%Items` section specifies how many items we want of each service, assigning an alias for each of them which is used for later reference. For instance, in Listing 5 we declare two SimpleBlockStorage items, `vol1` and `vol2`.

Requirements are defined as a constraint on different items terms which must be satisfied by a given configuration. The available operators and expressions to define requirements are the same already presented in Table 1. In Listing 5 we can see three requirements for the size and region of the two items previously declared.

In the third and last section of the user needs document, the preferences over the items are declared. The available constructs to define preferences are a subset of the Semantic Ontology of User Preferences (SOUP) [9], employed for the ranking of services. For our case, we have adapted five SOUP preferences in order to describe fuzzy user preferences on the configurable terms of a service item. The preference operators are as follows:

- *Favorites* defines a boolean or enumerated term value desired by the user. It receives the term and specific value as inputs – in the case of a boolean term, only the term is required. For example,
  `Favorites(storage["vol1"]. Region,"US").`
- *Dislikes* defines a boolean or enumerated term value not desired by the user. It receives the term and specific value as inputs – in the case of a boolean term, only the term is required. For example,
  `Dislikes(storage["vol1"]. SSD).`

Table 2: User preferences mapping.

Preference	Term type	Correspondence
Likes($term = v$)	Enumerated	$term = v \rightarrow p = 1$
Dislikes($term = v$)	Enumerated	$term \neq v \rightarrow p = 1$
Highest($term$)	Range	$p = \frac{value - lowerBound}{upperBound - lowerBound}$
Lowest($term$)	Range	$p = \frac{upperBound - value}{upperBound - lowerBound}$
Around($term, v$)	Range	$p = inverseDistance(value, v)$

- *Highest* defines a preference on the highest possible value for a given real or integer term. It receives the term as input. For example, `Highest(storage ["vol1"].Size)`.
- *Lowest* defines a preference on the lowest possible value for a given real or integer term. It receives the term as input. For example, `Lowest(VolumeStorage.totalCostMonth)`.
- *Around* defines a preference on a real or integer term to be around a specific value defined by the user. It receives the term and the specific value as inputs. For example, `Around(storage["vol1"].costGBMonth,0.1)`.

User preferences are mapped into real values and aggregated using a function, as described in prior work [11]. Table 2 shows in particular how these are mapped into real values between 0 and 1, so they can be later aggregated using a function to balance preferences satisfaction and rank service configurations. In Listing 5 we can see some of these preferences. While the two first preferences are expressed on item level terms, the last one refers to an HCS term. For that case, we employ the name of the service as an alias to refer to the global terms.

6 AUTOMATED ANALYSIS

In this Section we propose an operational semantics [14] of HCSs which primary goal is providing these models with an automated support for certain analysis operations. These operations will enable to check the satisfaction of the proposed validity criteria and support the search of the most suitable configuration among other capabilities.

The target domain of the proposed semantics are Stateful Feature Models (SFMs) [19, 20]. SFMs are intended to describe all the possible configurations of variability-intensive systems and assist their configuration. An SFM is divided in two parts: the Feature Model (FM), and a Configuration Model (CM). The FM defines the variable parts of a system in terms of features which are hierarchically organised by means of different kinds of relationships, forming a tree-like structure. Attributes can be linked to features to indicate any further information that could be relevant to configure the system. The CM enables a user to specify their needs, selecting or removing features and adding constraints for attribute values. The main strong point of mapping HCSs to SFMs is taking advantage of the Automated Analysis of Stateful Feature Models (AASFM) [19], which provides a wide catalogue of analysis operations in terms of which the demanded HCS analysis operations can be solved.

We present three mapping tables that summarise the procedure to generate a SFM from an HCS. First, Table 3 shows how to map from a single configurable service to a SFM. We create a root feature for every service, and a child (mandatory) feature for each selectable term. Besides, each term value is mapped into a feature. These features are grouped to their corresponding selectable term feature by means of an alternative relationship. This structure obligues a user to select one value for each selectable term in order to define a configuration for the service. The derived terms are mapped as global attributes linked to the root feature with their corresponding types. Additionally, p attributes are added to allow a user to indicate their preference for a given selectable or derivable term.

Table 3: Mapping CSs into SFMs

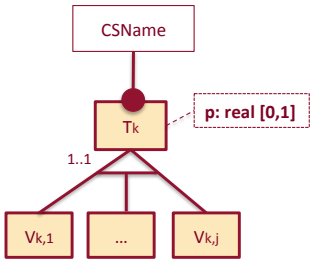
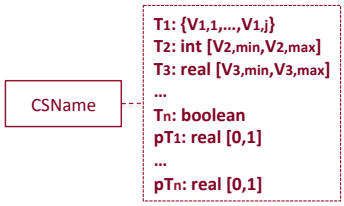
HCS	SFM
Service CSName { ... }	
%SelectableTerms T_k: $\{v_{k,1}, ..., v_{k,j}\}$;	
%DerivedTerms T_1: $\{v_{1,1}, ..., v_{1,j}\}$; T_2: int$[v_{2,min}, v_{2,max}]$; T_3: real$[v_{3,min}, v_{3,max}]$; ... T_n: boolean;	
%Dependencies C_1; ... C_n;	Constraints on the *SFM* features/attributes

Table 4: Mapping HCSs into SFMs

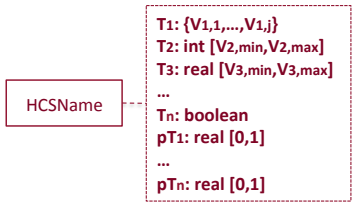
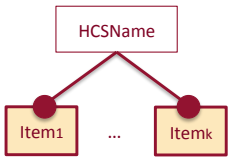
HCS	SFM
Highly-configurable Service HCSName { ... }	
%Terms T_1: $\{v_{1,1}, ..., v_{1,j}\}$; T_2: int$[v_{2,min}, v_{2,max}]$; T_3: real$[v_{3,min}, v_{3,max}]$; ... T_n: boolean;	
%Dependencies C_1; ... C_n;	Constraints on the *SFM* features/attributes

Table 5: Mapping items, requirements and preferences into SFMs

HCS	SFM
Items	
##On user HCS def. **%Services** CSName[n,m] CSAlias; ##On user needs def. **%Items** CSAlias[$Item_1$,...,$Item_k$];	
Requirements	
%Requirements R_1; ... R_n;	Constraints on the *SFM* features/attributes
Preferences	
Favorites(T_k,$V_{k,j}$)	$V_{k,j} = sel \Leftrightarrow T_k.p = 1 \wedge V_{k,j} = rem \Leftrightarrow T_k.p = 0$
Dislikes(T_k,$V_{k,j}$)	$V_{k,j} = rem \Leftrightarrow T_k.p = 1 \wedge V_{k,j} = sel \Leftrightarrow T_k.p = 0$
Highest(T_k)	$HCSName.p_{T_k} = \frac{V_k - V_{k,min}}{V_{k,max} - V_{k,min}}$
Lowest(T_k)	$HCSName.p_{T_k} = \frac{V_{k,max} - V_k}{V_{k,max} - V_{k,min}}$
Around(T_k,$V_{k,j}$)	$HCSName.p_{T_k} = \frac{max(V_{k,j} - V_{k,min}, V_{k,max} - V_{k,j}) - \lvert V_k - V_{k,j} \rvert}{max(V_{k,j} - V_{k,min}, V_{k,max} - V_{k,j})}$
Preferences Composition	
	$HCSName.p = \sum_k T_k.p + \sum_i HCSName.p_{T_i}$

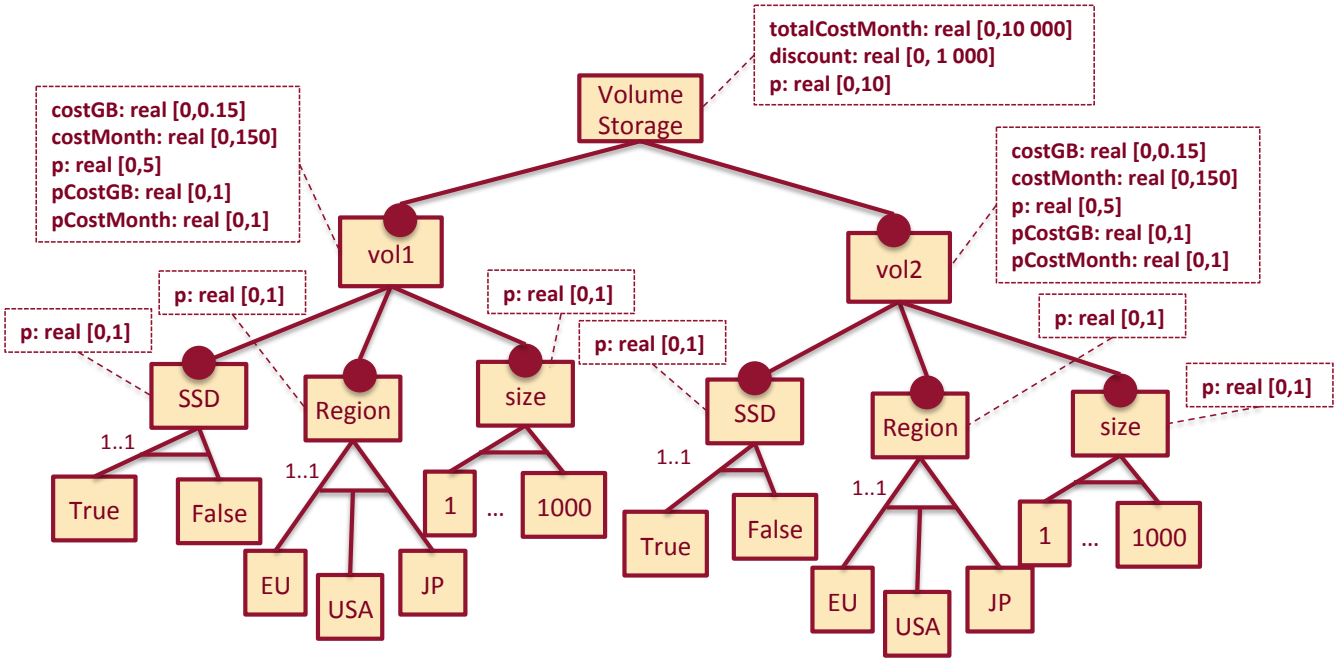


Figure 2: Volume Storage HCS translated into a SFM.

Second, the HCS and the user needs must be mapped together, since the specific number of instances to be created for each configurable service within the HCS must be known to generate the adequate SFM. Tables 4 and 5 show how HCS elements and user needs are mapped into a SFM. For each item created in user needs, a SFM is created following the mapping proposed for configurable services. Then, an HCS root feature is created as a way to bind all the item-specific root features by means of mandatory relationships. Then, attributes are created for HCS-specific terms following the same process than for configurable services. Finally, user needs, either requirements or preferences are mapped into SFM constraints as shown in Table 5.

Figure 2 showcases the result of applying the presented mappings to the whole example scenario that we used throughout the paper, described in Listings 1, 2, and 5.

The resulting SFM can be analysed using existing AASFM tools. So for example, a false configurable service can be detected if the resulting SFM is void. Table 6 shows a correspondence table with the AASFM operations that can be used to perform HCS analysis operations.

Table 6: Analysis operation correspondences

HCS operation	SFM operation
Inconsistent service	Void SFM
Dead value	Dead feature
False decision term	False-optional feature
Redundant dependency	Product listing (composite)
Best configuration	Best product

7 PROTOTYPE IMPLEMENTATION

In order to validate our approach, a prototype workbench has been developed[4]. This workbench consists of an on-line modeling environment and an analysis engine to get the optimal configuration for a given user need over an HCS. The tool has been developed within the context of the IDEAS framework that provides a microservices architecture of standardized modules that is based on REST interfaces. Each module is associated with specific languages (such as SYNOPSIS). The structure of an IDEAS module is comprised of two sets of operations: (i) Language management operations that provide syntax checking and the marshalling and unmarshalling of the different formats and (ii) analysis operations that provide specific functionality to extract information over the document loaded. In this work, the tool developed is based on a new HCS module that implements the different operations for a SYNOPSIS file modeling an HCS or the definition of User Needs.

The key analysis developed is the `Optimal Configuration` operation, which develops an analysis over the HCS possible configurations in order to find the optimal configuration that matches a certain User Needs. Figure 3 shows an screenshot of the tooling with a particular example of VolumeStorage service and the result of the optimal configuration operation over a specific user need. The analysis operation is based on the transformation to the SFM

[4]This tool is available at http://www.isa.us.es/IDEAS/HCS

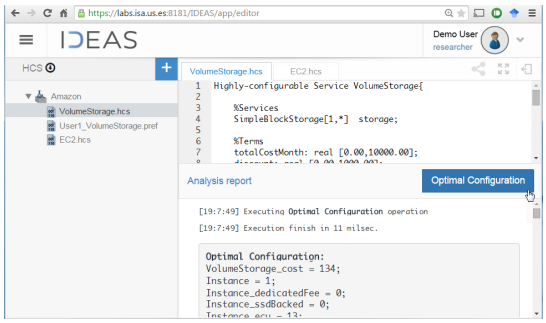


Figure 3: Prototype Implementation

formalization presented in Sec. 6; once the model is transformed, the tool invokes a reasoning engine for the SFM paradigm [12].

8 DISCUSSION

As many other domain-specific approaches, SYNOPSIS is focused and tailored to a specific, service-oriented audience. From the contents in previous sections, the reader can infer that SFMs are sufficient to deal with the description and configuration of HCSs. Although this is true, our intention with SYNOPSIS is to provide a simplified interface, which masks the details of variability management and speaks in terms of services. In this way, we are able to leverage the expressiveness and reasoning potential from the world of variability, while offering a simple façade for the world of services.

Our approach, at this stage, still presents several limitations. HCSs are intrinsically dynamic, evolving in environments where new services replace old ones and extend the ecosystem constantly. While our notion of HCSs enables the adaptation of the number of service instances, the type of these must be known beforehand. This supposes a limitation, refraining consumers from mixing and matching configurable services, instead of providing them with default HCSs. Additionally, we need further evaluation for our proposal. In this paper, we have presented a running example to showcase how SYNOPSIS works for the description of HCSs, and how we can implement automated analysis relying on existing reasoning support for SFMs. However, we need to evaluate how good these modelling and analysis capabilities are compared to traditional service-oriented and variability approaches, respectively.

9 CONCLUSIONS

HCSs analysis is a demanding task due to the high complexity and variability that this type of services deliver. In order to automate analysis operations, such as validity checking or selection of the best configuration, a precise definition of HCSs is needed. In this paper we propose a DSL that provides a comprehensive HCS model and a concrete notation (namely SYNOPSIS) to describe configurable and highly configurable services. We offer a tooling support that allows the user to perform automated analysis on HCSs, based on a mapping to SFMs. By using this formalism, we can reuse extensively evaluated techniques to execute analysis operations [12].

As future work, we plan to support additional preference constructs defined in SOUP [9], including composite preferences that

will allow the user to express richer and more complex needs. Moreover, we will implement additional analysis operations into the IDEAS module, such as obtaining all possible configurations and checking for further validity criteria. Finally, we are also devising other extensions to the model, in particular to adapt the configuration of HCSs for addressing specific compliance concerns that might arise due to geographical or regulatory restrictions [10].

ACKNOWLEDGMENTS

This work has been partially supported by the European Commission (FEDER), the Spanish and the Andalusian R&D&I programmes (grants P12-TIC-1867, TIN2015-70560-R), SFI grant 13/RC/2094 and ERC Advanced Grant no. 291652 (ASAP).

REFERENCES

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B France. 2013. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming* 78, 6 (2013), 657–681.

[2] Hans Akkermans, Ziv Baida, Jaap Gordijn, Nieves Peña, Ander Altuna, and Iñaki Laresgoiti. 2004. Value Webs: using ontologies to bundle real-world services. *IEEE Intelligent Systems* 19, 4 (July 2004), 57–66. DOI: https://doi.org/10.1109/MIS.2004.35

[3] Germán H Alférez, Vicente Pelechano, Raúl Mazo, Camille Salinesi, and Daniel Diaz. 2014. Dynamic adaptation of service compositions with variability models. *Journal of Systems and Software* 91 (2014), 24–47.

[4] David Benavides, Sergio Segura, and Antonio Ruiz Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (9 2010), 615–636.

[5] Jan Bosch. 2009. From software product lines to software ecosystems. In *Proceedings of the 13th international software product line conference*. Carnegie Mellon University, 111–119.

[6] Deepak Dhungana, Dominik Seichter, Goetz Botterweck, Rick Rabiser, Paul Grunbacher, David Benavides, and Jose A Galindo. 2011. Configuration of multi product lines by bridging heterogeneous variability modeling approaches. In *Software Product Line Conference (SPLC), 2011 15th International*. IEEE, 120–129.

[7] José A Galindo, Deepak Dhungana, Rick Rabiser, David Benavides, Goetz Botterweck, and Paul Grünbacher. 2015. Supporting distributed product configuration by integrating heterogeneous variability modeling approaches. *Information and Software Technology* 62 (2015), 78–100.

[8] José M. García, Pablo Fernandez, Carlos Pedrinaci, Manuel Resinas, Jorge Cardoso, and Antonio Ruiz-Cortés. 2017. Modeling Service Level Agreements with Linked USDL Agreement. *IEEE Transactions on Services Computing* 10, 1 (2017), 52–65.

[9] José M. García, Martin Junghans, David Ruiz, Sudhir Agarwal, and Antonio Ruiz-Cortés. 2013. Integrating Semantic Web Services Ranking Mechanisms Using a Common Preference Model. *Knowledge-Based Systems* (2013).

[10] Jesús García-Galán, Liliana Pasquale, George Grispos, and Bashar Nuseibeh. 2016. Towards adaptive compliance. In *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 108–114.

[11] Jesús García-Galán, Liliana Pasquale, Pablo Trinidad, and Antonio Ruiz Cortés. 2016. User-centric Adaptation Analysis of Multi-tenant Services. *Transactions on Autonomous and Adaptive Systems* (2016).

[12] J. García-Galán, P. Trinidad, O. F. Rana, and A. Ruiz-Cortés. 2016. Automated Configuration Support for Infrastructure Migration to the Cloud. *Future Generation Computer Systems* (2016).

[13] Mikko Heiskala, Juha Tiihonen, and Timo Soininen. 2005. A conceptual model for configurable services. In *Papers from the Configuration Workshop at IJCAI'05*, Dietmar Jannach and Alexander Felfernig (Eds.). 19–24.

[14] A. H. M. Ter Hofstede and H.A. Proper. 1998. How to Formalize It? Formalization Principles for Information System Development Methods. *Information and Software Technology* 40 (1998), 519–540.

[15] Steffen Lamparter, Anupriya Ankolekar, Rudi Studer, and Stephan Grimm. 2007. Preference-based selection of highly configurable web services. In *Proceedings of the 16th international conference on World Wide Web*. ACM, 1013–1022.

[16] Tuan Nguyen, Alan Colman, and Jun Han. 2011. Modeling and managing variability in process-based service compositions. In *International Conference on Service-Oriented Computing*. Springer, 404–420.

[17] Tuan Nguyen, Alan Colman, and Jun Han. 2016. A Feature-Based Framework for Developing and Provisioning Customizable Web Services. *IEEE Transactions on Services Computing* 9, 4 (2016), 496–510.

[18] Clément Quinton, Nicolas Haderer, Romain Rouvoy, and Laurence Duchien. 2013. Towards multi-cloud configurations using feature models and ontologies. In *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*. ACM, 21–26.

[19] Pablo Trinidad, Antonio Ruiz-Cortés, and David Benavides. 2013. Automated Analysis of Stateful Feature Models. In *Seminal Contributions to Information Systems Engineering*. Springer, 375–380.

[20] Pablo Trinidad, Antonio Ruiz-Cortés, and Jesús García Galán. 2014. Configurable Feature Models. In *Actas de las XIX Jornadas de Ingeniería del Software y Bases de Datos*. 335–348.

[21] Wil MP Van Der Aalst. 2010. Configurable services in the cloud: Supporting variability while enabling cross-organizational process mining. In *On the Move to Meaningful Internet Systems: OTM 2010*. Springer, 8–25.

[22] Stefan Walraven, Dimitri Van Landuyt, Eddy Truyen, Koen Handekyn, and Wouter Joosen. 2014. Efficient customization of multi-tenant Software-as-a-Service applications with service lines. *Journal of Systems and Software* 91 (2014).

[23] Erik Wittern, Jörn Kuhlenkamp, and Michael Menzel. 2012. Cloud service selection based on variability modeling. In *Service-Oriented Computing*. Springer.