Feature Placement Algorithms for High-Variability Applications in Cloud Environments

Hendrik Moens*, Eddy Truyen†, Stefan Walraven†, Wouter Joosen†, Bart Dhoedt* and Filip De Turck*
* Ghent University, IBBT Department of Information Technology
Gaston Crommenlaan 8/201, B-9050 Gent, Belgium
† Katholieke Universiteit Leuven, DistriNet Research Group, Dept. Computer Science
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
e-mail: hendrik.moens@intec.ugent.be

***Abstract*—While the use of cloud computing is on the rise, many obstacles to its adoption remain. One of the weaknesses of current cloud offerings is the difficulty of developing highly customizable applications while retaining the increased scalability and lower cost offered by the multi-tenant nature of cloud applications. In this paper we describe a Software Product Line Engineering (SPLE) approach to the modelling and deployment of customizable Software as a Service (SaaS) applications. Afterwards we define a formal feature placement problem to manage these applications, and compare several heuristic approaches to solve the problem. The scalability and performance of the algorithms is investigated in detail. Our experiments show that the heuristics scale and perform well for systems with a reasonable load.**

***Index Terms*—Distributed computing, Clouds, SPLE**

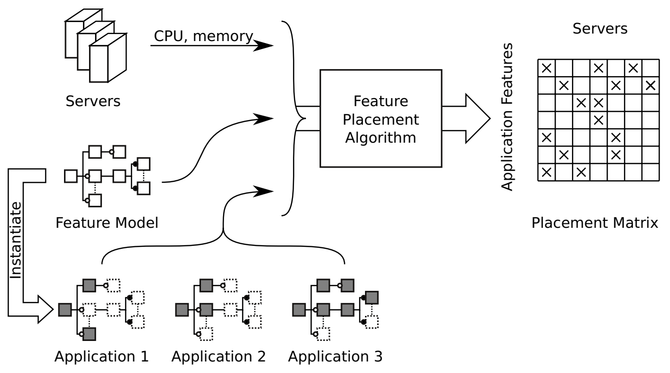


Fig. 1: A representation of the feature placement problem.

I. Introduction

Nowadays, there is a trend for moving applications to cloud infrastructure, consolidating hardware, saving costs and allowing applications to react faster to sudden changes in demands. Despite the many advantages of cloud computing, different obstacles to its adoption still exist.

Many existing cloud applications only deliver a limited amount of customizability, often using a one-size-fits-all approach or limiting customizations to mainly cosmetic changes. However, for some use cases in areas such as document processing, medical information management, and medical communication systems, applications must be tailored for specific customer needs. Often requiring different service configurations for different clients such as hospital-specific interfaces, custom workflows, and varying access and security policies. Current cloud platforms offer insufficient customizability for these cases. The CUSTOMSS[1] project seeks to create solutions to develop, deploy and manage highly configurable software and services on multi-tenant cloud infrastructures.

To build configurable desktop applications, the concepts of Software Product Line Engineering (SPLE)[2] are often used. In this approach, the software is modeled as a collection of features. By selecting and deselecting features, different software variants can be created. Features themselves are organized by relating them to each other in a feature model.

Most approaches in SPLE have focused on the development of statically configured products using core assets with static configuration of variation points. That is, all variations are instantiated before a product is delivered to customers and, once the decisions are made, it is difficult for users to alter them. This approach to SPLE can be used to create a large amount of applications, that can subsequently be run on cloud infrastructure. This approach would however create a unique application for every feature combination, making it impossible to exploit many of the interesting possibilities of clouds, such as multi-tenancy. In multi-tenant applications, multiple end users can make use of the same application instances, increasing the scalability of applications and lowering the cost per user. On the other hand, creating multi-tenant services to represent every feature, can cause individual feature instances to be underused, especially if many features and variants exist. For this purpose, current cloud application placement techniques [3], [4], that do not take relationships between services into account, are inadequate.

In this paper, we focus on the design of algorithms for placing high-variability applications on cloud infrastructure. The applications are built by composing them from a set of multi-tenant feature instances using a Service-Oriented Architecture (SOA). For this purpose we designed a variation of the application placement problem [5], which we refer to as the feature placement problem. An overview is shown in Figure 1. The feature placement problem determines which servers will execute which feature instances, taking into account the datacenter server configuration, applications to be placed, and the feature model of which the applications are instantiations. A single feature instance is capable of serving multiple applications, ensuring applications composed of a set of features are themselves multi-tenant. In this paper, the following research questions are addressed: (i) How can the feature placement problem be represented formally? (ii) Which heuristic and optimal approaches can be designed to solve this problem? and (iii) What is the performance of the heuristic solutions to this problem compared to the optimal solution, both in placement quality, and execution speed?

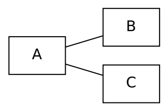
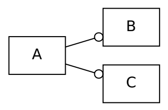
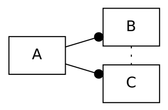
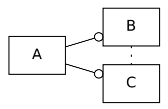
In the next section, we will discuss related work. Afterwards, in Section III we describe the feature modelling approach, and how it can be applied to cloud applications. In Section IV we formally describe the feature placement problem. This is followed by Section V, where we describe different approaches to solve the placement problem. In Section VI we describe the setup of the evaluation. Subsequently, in Section VII we evaluate the heuristics. Finally, Section VIII contains our conclusions.

II. Related work

Industrial research has been done on configuration policies and methodologies to support customizations of Software as a Service (SaaS). In [6], Zhang et al. discuss a policy-based framework for publishing customization options of web services and building customizations on top of this, enabling clients to build their own customizations. They however do not take multi-tenancy and runtime aspects into account, nor do they propose a software development methodology to create the customizable applications. Sun et al. [7] proposed an approach choosing configuration over customization to create modifiable applications, and propose a software development methodology to develop such applications. We, by contrast, focus on the customization aspect by using SPLE methods in combination with a SOA development approach. In Mietzner et al. [8] an approach for modelling customizable applications built using SOA is described. The application is linked to a feature model, allowing automatic generation of deployment scripts. Our approach is similar in its use of SOA in the proposed development approach. We however focus on the runtime management of customizable applications, proposing optimal and heuristic algorithms to determine where to run specific features. Recent work in the SPLE community [9], [10] further works towards the development of customizable SaaS applications, but to the authors' knowledge there has been no work concerning the runtime management of these applications.

The application placement problem has previously been described formally [3], [4], [5], [11], and many different approaches to application placement in clouds have been developed over the recent years. Specific requirements have however led to the creation of many application placement variants, each focusing on different parameters. Whalley et al. [12] extended a Virtual Machine (VM) management system to take into account the complexities of software licensing. In a similar way, Breitgand et al. [13] added the consideration of Service Level Agreements (SLAs) to the placement problem. The consideration of energy consumption and carbon emissions was added in [14] using a system that works in parallel with existing datacenter brokering systems. Similarly, we extend the generic application placement problem formulation to place the features of applications in a cloud environment.

TABLE I: Graphical representation of feature models, description of relations, and formal representation.

	Mandatory If the parent is selected the child must be selected as well. $\mathbf{Mandatory}(f_A, f_B)$ $\mathbf{Mandatory}(f_A, f_C)$
	Optional If the parent is selected the optional children can be selected. $\mathbf{Optional}(f_A, f_B)$ $\mathbf{Optional}(f_A, f_C)$
	Alternative If the parent is selected exactly one of the child nodes must be selected. $\mathbf{Alternative}(f_A, \{f_B, f_C\})$
	Or If the parent is selected at least one of the child nodes must be selected. $\mathbf{Or}(f_A, \{f_B, f_C\})$

III. Feature modelling concepts

Using SPLE, an application is modeled as a collection of features and relations between these features. Sometimes the inclusion of some features can imply the inclusion of other features and conversely the inclusion of some features can exclude other features. To make it easier to reason on these relations, feature models are often created in a hierarchical fashion. Typically, four different relation types: *mandatory*, *optional*, *alternative*, and *or* are used. Feature models can be represented graphically. When doing so, we use the notation used in [15]. Table I contains the different relation types, a description and graphical representation, and a formal notation which will be used later on in this paper.

An example feature model is shown in Figure 2. The figure shows an illustrative fragment of the feature model for a medical data processing application. The application contains an *interfacing engine* feature to connect to individual hospitals, which is capable of handling input in one or more different formats. Additional *encryption* can optionally be added to the interfacing engine. Finally, parts of the application can be hosted at the hospital or they can be hosted by the application provider. An application created for a hospital using their own datacenter and a hospital specific interface will differ significantly from the application created for a hospital using public cloud infrastructure and a HL7 data interface.

Features can be implemented in various ways. Sometimes a feature can be implemented by simply changing configuration files. This could for example be changing the logo of an applications. More complicated changes can be created by adding code changes. The most complicated changes lead to completely different modules being used by the applications. The first method is variation by configuration, the latter two variation types are considered customization [7].

In this paper, we only consider customization as it leads to the creation of applications that are different at the code level. Configuration-based features can already be adapted into a cloud context using existing software development techniques [7]. Because of this, the feature models used further on in this paper will only contain features that cause changes in the executed code.

The development of applications will be driven using the feature model, building an application using a SOA, in which the individual services map to the different features defined in the feature model. Deploying the application then comes down to allocating feature instances and connecting them. We assume that the individual services are multi-tenant and can serve multiple applications. The allocation of the different feature instances is the main focus of this paper.

It is important to make a distinction between internal and external variability [2]. External variability is visible for end users and communicated to them, whereas internal variability only leads to changes that are visible to developers and usually pertains to non-functional system qualities. This enables a configurator to leave the internal variability undecided, creating open variation points [8], which allows the placement algorithm to fill in these variation points when an application is deployed or moved. This way, an application with regular availability requirements could use high availability instances when such instances exist with remaining capacity, instead of creating a new instance with a lower reliability, thus lowering the total resource usage.

IV. Formal problem description

In this section we will formally describe the feature placement problem. For ease of reference, the variables used in the model are listed in Table II.

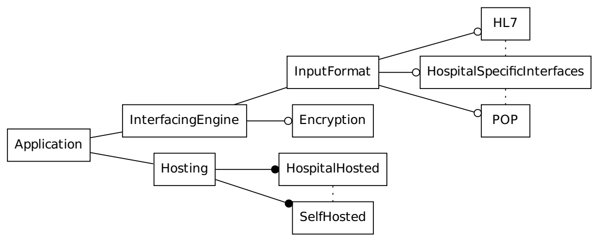


Fig. 2: A feature model fragment for a medical data processing application.

TABLE II: The different symbols used in Section IV.

Input Variables	
Symbol	Description
S	The set of servers
Ra_s^γ	The available resources on a server s for a resource type $\gamma \in \{CPU, mem\}$
$\mathcal{F}$	The feature model used by the applications
F	The set of features contained in $\mathcal{F}$
$\mathcal{R}$	The set of relations contained in $\mathcal{F}$, using relations as described in Table I
A	The set of applications
$\text{sel}(a)$	The features that must be included for application a
$\text{excl}(a)$	The features that must not be included for application a
$FI_{f_1}^{CPU}(f_2)$	The impact on the CPU requirement for feature f_2 if feature f_1 is included in the selected features of an application
IR_f^{mem}	The memory requirement of a single instance of a feature f
$C^V(f,a)$	The cost of failing to place a feature f for an application a
$C^V(a)$	The cost of failing to place an application a
Decision Variables	
Symbol	Description
$M_{s,f,a}^{CPU}$	The amount of CPU to be allocated for a given server s, feature f and application a
$\Phi_{f,a}$	A boolean variable indicating whether application a includes feature f
$IC_{s,f}$	The instance count is a boolean variable, indicating whether a feature f is instantiated on a server s
Auxiliary Variables	
Symbol	Description
$AI_{f,a}^{CPU}$	The application impact. It contains the actual CPU impact per feature f of a specific application a.
p_a	A boolean variable indicating whether the CPU demand of any of the features of an application a is not correctly provisioned.
$p_{f,a}$	A variable indicating whether the CPU demand of a single feature f of an application a is not correctly provisioned

A. Variable description

1) Input variables: Each problem has a set of servers S with an amount of available resources. There are two resource types: memory and CPU. For a server $s \in S$ the available resources are given by Ra_s^{mem} and Ra_s^{CPU} for memory and CPU respectively. The goal of the optimization is to allocate the required CPU capacity for applications at a minimal cost. To achieve this, the amount of CPU is measured globally across all servers and optimized. Feature instances are allocated on different servers and consume memory on each server. For a placement to be valid, every feature instance must be assigned its required amount of memory.

The problem also contains a set of applications A that must be placed. Each of the applications is a specific instantiation of a global feature model $\mathcal{F}$. This feature model contains a set of features F and a collection of relations $\mathcal{R}$, formally describing the feature model tree. The possible relations are described in Section III and Table I. We note that this approach still allows the placement of entirely distinct applications with separate feature models $\mathcal{F}_i$ by creating a set containing the roots of every feature model, R, and linking these different feature models in a global feature model by the addition of a new root feature r and a relation $\textbf{Alternative}(r, R)$.

Every application $a \in A$ contains a set $\mathrm{sel}(a) \in F$ with features that have been selected in the application and a set $\mathrm{excl}(a) \in F$, containing features that have been excluded ($\mathrm{sel}(a) \cap \mathrm{excl}(a) = \emptyset$). The configuration of both is assumed to be valid according to $\mathcal{F}$.

It is possible for features to impact the CPU needs of other features. For instance, adding the *encryption* feature to the application in Figure 2 can increase the load on the *interfacing engine*, and applications hosted by the application provider will require more CPU resources than applications partially hosted at the client site. We represent this using a feature impact matrix FI. $FI_{f_1}^{CPU}(f_2)$ represents the CPU impact of feature f_1 on feature f_2. Every instance of a feature f also requires a specific amount of memory IR_f^{mem}. In this paper we assume that the resource need of an application is defined entirely by the selected application features, if needed an additional demand variable could also be added.

To be able to optimize application placement, the cost of failed placement must also be used as an input. The cost of failure can be considered in two different ways:

- The cost of violating the SLA for a specific feature f of an application a is given by $C^V(f, a)$. This can be used if failure of specific features needs to be taken into account.
- The cost of violating the SLA for *any* feature of an application is given by $C^V(a)$.

2) Decision variables: The output of the placement algorithm is an allocation matrix M^{CPU}. For a server s, feature f, and application a, $M_{s,f,a}^{CPU}$ contains the amount of CPU that needs to be allocated.

Another output is the feature matrix Φ, indicating which applications are selected and excluded for a given application. For application a and feature f, $\Phi_{f,a} = 1$ if the application contains the feature and $\Phi_{f,a} = 0$ if it does not. At the start of the algorithm a large part of this matrix can be filled in by using $\mathrm{sel}(a)$ and $\mathrm{excl}(a)$.

Finally, the variable $IC_{s,f}$ determines the amount of instances of feature f on server s. This variable is needed to determine the total memory usage of feature f on server s.

3) Auxiliary variables: Next to the in- and output variables, additional variables are needed to construct the final cost function and constraints. First there is the application impact matrix $AI_{f,a}^{CPU}$ which contains for every feature f and application a the actual CPU requirement. It can be constructed using the selected features and the feature impacts matrix.

Secondly, a set of boolean variables is needed to express whether an application is correctly provisioned. We make a distinction between two different groups:

- For every application a there is a variable p_a, indicating whether the provisioning of an application has failed. If $p_a = 1$, a feature exists that has not been allocated sufficient CPU resources.
- For every application a and feature f there is a variable $p_{f,a}$. This variable indicates whether a single feature of the application is insufficiently provisioned.

TABLE III: Conversion of $\mathcal{F}$ to constraints.

Relation	Conversion
$\mathbf{Mandatory}(f_A, f_B)$	$f_A = f_B$
$\mathbf{Optional}(f_A, f_B)$	$f_A \geq f_B$
$\mathbf{Alternative}(f_A, \{f_B, f_C\})$	$f_A = f_B + f_C$
$\mathbf{Or}(f_A, \{f_B, f_C\})$	$f_A \geq f_B$
	$f_A \geq f_C$
	$f_A \leq f_B + f_C$

B. Constraint details

1) Feature-based constraints: The feature matrix Φ is used to indicate whether a feature f is present in an application a. For an application a we add the constraints $\Phi_{f,a} = 1$ if $f \in \mathrm{sel}(a)$ and $\Phi_{f,a} = 0$ if $f \in \mathrm{excl}(a)$.

The relations between features $\mathcal{R}$ must also be converted into constraints. Elements of $\mathcal{R}$ define relations between individual features. As the constraints of the feature model affect all applications, they must be applied to all application features in the feature matrix. Because of this, we define $f_i = \Phi_{i,*}$ a row of the feature matrix. We describe the conversion for the relation types to constraints in Table III.

2) Application resource requirement constraints: Application resource requirements can be determined using the feature impact matrix FI. Each feature f can have resource requirements, but it can also impact resource requirements of other features. If feature f is selected, it's impact matrix, FI_f^{CPU} will be added to the total resource requirement for the application. A feature f_i can only affect a feature f_j if f_i requires f_j according to the feature model as otherwise the feature impact matrix would be able to add feature constraints not included in the feature model.

Using the selected features Φ and the feature impact matrices FI_f^{CPU}, an application impact matrix $AI_{f,a}^{CPU}$ can be constructed. This application impact matrix, expressed in Equation (1), displays the actual CPU requirements for individual features f, of an application a.

$$AI_{f,a}^{CPU} = \sum_{f' \in F} \Phi_{f',a} \times FI_{f'}^{CPU}(f) \tag{1}$$

3) Resource constraints: A set of constraints are added due to the limited amount of resources available in de model. CPU and memory constraints are both expressed for every server s, but both are expressed in different way: The used CPU is determined using the allocation matrix M^{CPU}, of which the requirement is aggregated over all features and applications. This is expressed in Equation (2). Memory limitations follow from the instance count IC for the service, indicating whether a service is allocated, and the required amount of memory per-instance, as shown in Equation (3).

$$\sum_{f \in F} \sum_{a \in A} M_{s,f,a}^{CPU} \leq Ra_s^{CPU} \tag{2}$$

$$\sum_{f \in F} (IR_f^{mem} \times IC_{s,f}) \leq Ra_s^{mem} \tag{3}$$

4) Application provisioning constraints: Additional constraints are needed to ensure the variables $p_{f,a}$ and p_f, introduced in Section IV-A3, correctly express whether the application and features are insufficiently provisioned. Logically, we can express this using Equation (4).

$$p_{f,a} \equiv \sum_{s \in S} M^{CPU}_{f,s,a} < AI^{CPU}_{a,f} \tag{4}$$

This statement can be turned into constraints using the transformation of Equation (5) to Equation (6), with $x \in \{0,1\}$, $\mathbf{M}$ a number larger than any possible value of $\mathbf{expr}$. If $x = 0$, it follows from Equation 6 that $\mathbf{expr} \leq 0$, while $x = 1$ yields the constraint $\mathbf{expr} \leq \mathbf{M}$, which is always true. Consequently, this transformation holds only in optimizations where the objective function value improves when $x = 0$.

$$\begin{aligned} x &\equiv \mathbf{expr} \geq 0 && (5) \\ \mathbf{expr} &\leq x \times \mathbf{M} && (6) \end{aligned}$$

Once the different $p_{f,a}$ variables, we can use these to determine the value of p_a, as the failure of a single feature implies the failure of the entire application. This can be expressed by adding the constraint $p_a \geq p_{f,a}$ for every feature f and application a.

C. Optimization objective

The goal is to minimize the cost of non-realized demand. Using the variables p_a and $p_{f,a}$, the cost of application failure C^V_a, and the cost of feature failure $C^V_{f,a}$, we can express the cost of non-realized demand:

$$C_D = \sum_{a \in A} \left(p_a \times C^V_a + \sum_{f \in \mathrm{sel}(a)} p_{f,a} \times C^V_{f,a} \right) \tag{7}$$

Equation (7) considers all the applications and adds costs based on the failure to provision entire applications and the failure to provision individual application features.

V. Solution techniques

A. Integer Linear Programming (ILP)

The formal formulation, discussed in the previous section, can be used to define an ILP. This program can be solved using a commercial ILP solver, and yields the optimal problem solution using Simplex and Branch and Bound algorithms.

B. Heuristic solutions

Algorithm 1 shows the body of a heuristic solution to the feature-based application placement problem. The algorithm is based on the classic first-fit algorithm for bin packing. The algorithm iterates over a list of all application features, ordered before place is called using a **featureOrder** function, and tries to place them on the servers one by one. The order in which servers are visited is determined using a **serverOrder** function. A findServer operation iterates over a list of servers and returns the first server on which a given resource demand can be placed. The feature is placed on the server returned by this findServer operation and the placement continues for the next application. To determine exactly which features are to be placed, an additional function, **featureConversion** is executed before placement. This method ensures all features in a feature model are either selected or excluded.

```
Data: problem $P$
Data: Instance Count for a feature on a server $IC_{s,f}$
Data: current placement matrix $M_{s,f,a}$
Data: list of applications and features to place $List^{a,f}$
Data: list of servers with remaining resources $List^s$
if $List$ is empty then
    return ($IC$, $M$);
else
    $(a, f) \leftarrow$ head of $List^{a,f}$;
    sort $List^s$ using serverOrder;
    $s \leftarrow \mathrm{findServer}(List^s, \mathrm{remainingDemand}(a, f))$;
    if no $s$ found then
        placeNext $\leftarrow$ tail of $List^{a,f}$;
        return place(placeNext);
    end
    if no remaining capacity for $f$ on currentServer then
        create new instance of $f$;
    end
    add remaining demand to this instance;
    adjust remaining CPU to place;
    if all CPU placed then
        placeNext $\leftarrow$ tail of $List^{a,f}$;
    else
        placeNext $\leftarrow List^{a,f}$;
    end
    return place(placeNext);
end
```

Algorithm 1: The place function executed by the heuristic.

The effectiveness of the algorithm is largely determined by the **featureOrder**, **serverOrder**, and **featureConversion** functions. In the following sections different possible implementations for these functions will be presented.

1) Feature ordering: The Cost Feature Order (CFO) approach orders application features according to their cost of failure ($C^V(f, a) + C^V(a)$), placing the most expensive application features first.

We also designed an Instance Count Based (ICB) approach where the order of features to be instantiated is determined by the amount of instances required for them, thereby placing applications with more features that are more difficult to place first, ensuring their requirements will not be violated.

2) Server ordering: We consider an *Instance Based* server ordering, which orders servers according to the best fit for the feature f and application a that is to be placed: applications having instances of f with remaining capacity will be preferred, and of those the server with the best fit will be selected.

3) Feature model conversion: The Cheapest Application (CA) approach seeks the cheapest feature combination considering only the application itself, and not the other applications present in the model. This implies that every feature configu-

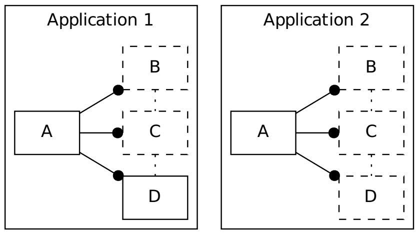


Fig. 3: Different feature model selections for two applications. Features with a solid border are selected, features with a dotted border are undecided.

ration can be determined when applications are added to the system, and not when placement executes.

Using Cheapest Shortened (CS), the cheapest feature combination is determined, taking into account all of the applications that must be placed. This approach is capable of yielding better results as all of the different applications and their combinations are taken into account, but has a higher computational cost. As the amount of combinations increases exponentially, at each point in time the list of possibilities is shortened. In the evaluations we use two variants, one shortening to 10 elements and the other using 100 elements.

In the context of these conversions we consider cheapest to be the combination requiring the smallest amount of memory for all required instances. Figure 3 shows two applications with the same feature model. Application 1 is fully configured while Application 2 is only partially configured, and still has open variation points. Feature model conversion will need to select either feature B, C or D for Application 2. Using CA, this will be the cheapest of the different alternatives. CS on the other hand may select feature D, even if it isn't the cheapest when it is considered on its own, ensuring less different feature instances have to be created.

Calculating CA or CS is of and exponential complexity, severely limiting the size of the feature models that can be considered. Part of the required information can however be calculated before placement. In the case of CA, the cheapest application can be calculated when applications are added to the system, using the stored result when the actual placement is executed. For CS, a list containing the best feature combinations can be determined when the application is added. During the execution, the latter variant will still require computations to determine the cheapest combination based on the generated list. The prepared CA (pCA) and prepared CS (pCS) approaches execute the algorithm assuming this information has been calculated and stored when applications are added to the system.

VI. Evaluation setup

We implemented the Integer Linear Programming (ILP) problem and the heuristics using Scala. The ILP solver uses CPLEX[16] as its backend. We also implemented a problem generator, capable of creating a wide range of random problems. The generator creates a collection of servers, a feature model, and a set of applications.

First, the servers S are generated. For these tests we assume a uniform server configuration with 4000MiB memory and a 2000MHz processor.

To create a random feature model $\mathcal{F}$, first, a collection of features F is generated with random memory requirements from a set $\{$500MiB, 1000MiB, 2000MiB, 2500MiB$\}$. Subsequently a feature model tree $\mathcal{R}$ is created. This is done by iteratively selecting nodes that are not in the tree yet and adding them in a relation with a node in the tree as the parent node. To start this process, a random feature is selected as root of the feature tree. There is an equal chance of picking any of the four relation types, and *Alternative* and *Or* relations have between two and six child nodes. Feature models generated in this fashion are similar in structure to those used for the applications in the CUSTOMSS project, and enable us to evaluate the algorithms for a larger set of configurations.

Next, we generate the impact matrix FI^{CPU}. Each feature impacts itself and has a chance of impacting any feature required by it. This is enforced by only letting a feature impact parent features. The CPU impact of a feature is randomly chosen from the set $\{$100MHz, 200MHz, 500MHz, 1000MHz$\}$. As stated earlier, we assume a homogeneous host capacity.

Selecting features is done by randomly selecting or excluding features, and checking the validity of the resulting feature model with SAT4J[17], an open source SAT solver. This ensures that the selection is feasible according to feature model $\mathcal{F}$. Features are randomly removed from either the collection of selected features, or from the collection of excluded features. All dependent features are removed as well, ensuring an open variation point is added.

Finally, random applications A are generated using the generated feature selections. Each application and application feature is also assigned costs for failure, randomly chosen from the set $\{1, 20, 50, 100, 500, 1000\}$. The applications with cost of failure 1 could correspond with a free service that is being offered: the placement algorithm should try its best to place it, but the cost of failure is minimal.

The performance tests of the algorithms were executed on a Linux server with an Intel Core i3 CPU (2.93GHz) with 4GiB of memory, and using Scala version 2.9.0.1.

VII. Evaluation results

We first compare the cost of using the heuristics. We generated 1000 problem models with each between 10 and 100 applications, features and servers. For each of these models we determined the *load* of the problem. We do this by filling in the feature model for every application using the same approach as CA, filling in the open variation points with the cheapest alternative, determining the total demand of all features for all applications, and dividing it by the total available resources. A higher load indicates that it is more difficult to place all applications on the servers.

We then filter out all applications with a load > 3, as we believe such heavy loads would be better handled using

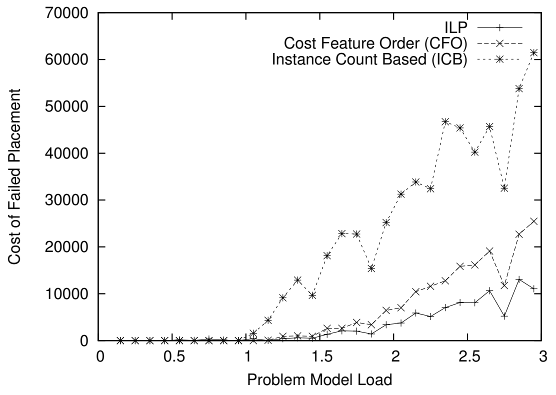


Fig. 4: The quality of different feature orderings using the CS100 feature conversion.

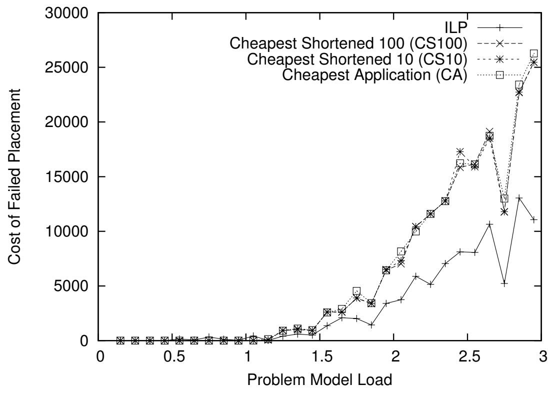


Fig. 5: The quality of different feature conversions using the CFO feature ordering.

admission policies. In the case of our model, this left 160 problem models to be considered.

CPLEX was used to solve the feature placement for the selected problems. For some randomly generated problems there was however insufficient memory, while larger problem models could be solved without difficulty. Similarly, CPLEX had trouble to evaluate some of the problems due to the finite precision representation of double values, yielding lower quality solutions that sometimes contain non-integer results for integer variables, causing constraint violations. These problematic models were removed and replaced with equally large models that did not yield any problems for CPLEX.

The probability of impacting other features mainly influences the problem model load, as a high impact chance implies a higher CPU demand, making placement more complicated. In the application cases, features commonly impact each other, as for example an encryption feature can impact many different components and a BPEL engine used in an application feature can be influenced by the features that make use of it. Because of these consideration we use an impact chance of 50%. We repeated the following tests for different impact chances, but this did not significantly change the results.

A. *Evaluation of the feature orderings*

We first compared the different feature orderings. A comparison of both is shown in Figure 4. In this evaluation the feature configuration is filled in using Cheapest Shortened 100 (CS100). The load is defined as explained earlier in this section, and the different problems are aggregated in bins of size 0.1. The cost determined by the cost of failed placement as defined in Section IV-C.

The data points in Figure 4 are subject to a large standard deviation. This is to be expected as we make use of a large amount of different feature models and problem configurations. Despite this, some interesting trends can be discerned: CFO, which takes the costs of failed placement into account yields significantly better results than the ICB approach. We also see that CFO yields results very close to those of the ILP solution for problems having a load of up to 2, and even for higher loads, the results remain quite good.

B. *Evaluation of the feature conversions*

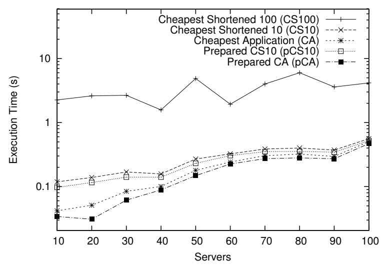
The effects of feature conversion are shown in Figure 5. The impact of the feature conversion on the cost is more limited than the impact of feature orderings. In general, CS yields better results than CA, but some corner cases can be found there CA actually performs better. The difference between the two cheapest combination variations is even smaller, with CS100 sometimes improving the results of CS10 but, sometimes performing worse. This implies shortening to only 10 elements, and considering only a small collection of good alternative feature model configurations, is sufficient to improve the quality of the placement.

C. *Execution speed considerations*

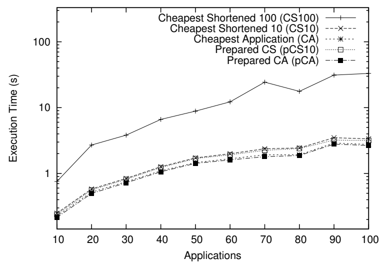
We compare the execution speed as a function of server counts, feature counts, and application counts. Each time, we vary one of the parameters while keeping the others fixed. Every data point in the graphs in this section is an average of 10 different executions. Only the feature conversion function has an impact on the performance as the different feature orderings merely by changing the order in which features are considered by the algorithm.

As is typical for ILP solution algorithms, there is a high variability in the performance of executions, with some problems taking hours to solve. Because of this, we do not include the execution speed of the ILP solver in these evaluations.

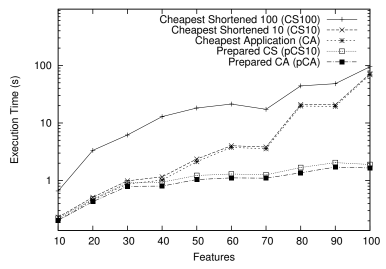
As shown in Figure 6a, increasing the amount of servers only slowly increases the execution time of the algorithm, as the main execution time follows from generating the different feature combinations, which is dependent on the amount of features and must be done for every application. The performance of the CS solution is strongly dependent on the amount of feature models considered: considering 100 solutions leads to a much larger computation cost while considering only 10 solutions only yields a minor overhead when compared to the



(a) Execution speed as a function of server counts (10 executions, $|F| = 20$ and $|A| = 20$)



(b) The execution speed as a function of application counts (10 executions, $|S| = 100$ and $|F| = 20$)



(c) The execution speed as a function of feature counts (10 executions, $|S| = 100$ and $|A| = 20$)

Fig. 6: Execution speed evaluation results.

CA approach. Considering the minor performance improvement caused by using 100 entries compared to using 10 entries we suggest using the latter variant. The last two variants, pCS and pCA show the effect on placement execution if part of the computation is prepared before execution. When it comes to varying server counts the impact of this change is limited.

When the number of applications is increased, as shown in Figure 6b, we observe a steady increase in execution time, as for every new application, feature model combinations must be considered. The benefit of preparing part of the computation before placement using pCS or pCA is limited.

The CS100, CS10 and CA algorithms scale badly in the number of features in the feature model, as shown in Figure 6c. In this case, preparing part of the computation before the actual execution, as done in pCS10 and pCA, causes a significant improvement in execution performance, greatly improving the execution speeds and scalability of the algorithm, and making it capable of provisioning applications containing large numbers of features.

VIII. Conclusions

In this paper we addressed three issues. First, we discussed an approach for managing applications with high variability using feature modelling techniques. We then presented a formal description of the feature placement problem, used to place these applications on cloud infrastructures, and developed heuristic solutions. Finally, we studied the performance of the heuristics, comparing them to an optimal ILP-based algorithm. We found that the best of the heuristics perform close to the optimal solution and and scale well, executing within 1s for the considered evaluation scenarios.

In future work we will extend the presented problem to take quality metrics such as reliability into account. Within the scope of the CUSTOMSS project, the designed algorithms will be incorporated in the overall framework.

Acknowledgement

Hendrik Moens is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). This research is partly funded by the IBBT CUSTOMSS[1] project. This work was carried out using the Stevin Supercomputer Infrastructure at Ghent University.

References

[1] (2011) CUSTOMSS: CUSTOMization of Software Services in the cloud. [Online]. Available: http://www.ibbt.be/en/projects/overview-projects/p/detail/customss

[2] K. Pohl, G. Böckle, and F. Van Der Linden, *Software product line engineering: foundations, principles, and techniques*. Springer, 2005.

[3] F. Wuhib, R. Stadler, and M. Spreitzer, "Gossip-based Resource Management for Cloud Environments," in *6th International Conference on Network and Service Management (CNSM)*, 2010, pp. 1–8.

[4] Y. Li, "Self-Adaptive Resource Management for Large-Scale Shared Clusters," *Science And Technology*, vol. 25, no. 2009, pp. 945–957, 2010.

[5] C. Tang *et al.*, "A scalable application placement controller for enterprise data centers," in *16th international conference on World Wide Web*, 2007, pp. 331–340.

[6] K. Zhang *et al.*, "A Policy-Driven Approach for Software-as-Services Customization," in *9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE)*. IEEE, Jul. 2007, pp. 123–130.

[7] W. Sun *et al.*, "Software as a Service: Configuration and Customization Perspectives," in *IEEE Congress on Services Part II (services-2)*. IEEE, Sep. 2008, pp. 18–24.

[8] R. Mietzner *et al.*, "Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications," in *ICSE Workshop on Principles of Engineering Service Oriented Systems*, vol. 215483. IEEE, May 2009, pp. 18–25.

[9] M. Abu-Matar and H. Gomaa, "Feature Based Variability for Service Oriented Architectures," in *9th Working IEEE/IFIP Conference on Software Architecture*. IEEE, Jun. 2011, pp. 302–309.

[10] S. T. Ruehl and U. Andelfinger, "Applying Software Product Lines to create Customizable Software-as-a-Service Applications," in *Proceedings of the 15th International Software Product Line Conference (SPLC)*, 2011, pp. 16:1–16:4.

[11] C. Adam and R. Stadler, "Service Middleware for Self-Managing Large-Scale Systems," *IEEE Transactions on Network and Service Management*, vol. 4, no. 3, pp. 50–64, Dec. 2007.

[12] I. Whalley and M. Steinder, "Licence-aware management of virtual machines," in *Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2011, pp. 169–176.

[13] D. Breitgand and A. Epstein, "SLA-aware Placement of Multi-Virtual Machine Elastic Services in Compute Clouds," in *Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011)*, 2011, pp. 161–168.

[14] C. Peoples, G. Parr, and S. McClean, "Context-Aware Characterisation of Energy Consumption in Data Centres," in *3rd IEEE/IFIP International Workshop on Management of the Future Internet (ManFI)*, 2011, pp. 1246–1253.

[15] pure-systems GmbH, *pure::variants User's Guide*, version 3.0 ed. [Online]. Available: http://www.pure-systems.com/Documentation.116.0.html

[16] (2011) IBM ILOG CPLEX 12.2. [Online]. Available: http://www-01.ibm.com/software/integration/optimization/cplex-optimizer

[17] (2011) SAT4J 2.2.2. [Online]. Available: http://www.sat4j.org/