Journal of Logical and Algebraic Methods in Programming 85 (2016) 173–199

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Resource–usage–aware configuration in software product lines

Damiano Zanardini [a,*], Elvira Albert [b], Karina Villela [c]

[a] Technical University of Madrid, Spain
[b] Complutense University of Madrid, Spain
[c] Fraunhofer IESE Kaiserslautern, Germany

ARTICLE INFO

Article history:
Received 2 August 2014
Received in revised form 16 July 2015
Accepted 26 August 2015
Available online 6 September 2015

ABSTRACT

Deriving concrete products from a product-line infrastructure requires resolving the variability captured in the product line, based on the company market strategy or requirements from specific customers. Selecting the most appropriate set of features for a product is a complex task, especially if quality requirements have to be considered. *Resource–usage–aware configuration* aims at providing awareness of resource–usage properties of artifacts throughout the configuration process. This article envisages several strategies for resource–usage–aware configuration which feature different performance and efficiency trade-offs. The common idea in all strategies is the use of resource–usage estimates obtained by an off-the-shelf *static resource–usage analyzer* as a heuristic for choosing among different candidate configurations. We report on a prototype implementation of the most practical strategies for resource–usage–aware configuration and apply it on an industrial case study.

1. Introduction

One increasing trend in the market of Software Engineering is the need to develop multiple, similar software products instead of just a single individual product. *Software Product-Line Engineering* (SPLE) [41] offers a solution to this trend based on explicitly modeling what is common and what differs among product variants, and on building a reuse infrastructure, a so-called *product-line infrastructure*, that can be instantiated and possibly extended to build the desired similar software artifacts (the products).

Deriving concrete products from a product-line infrastructure requires resolving the variability captured in the product line according to a company's market strategy or the requirements from specific customers. *Feature models* [35,21] have been the main approach for capturing the commonality and variability in product lines. The process of *product configuration* usually consists in selecting those features that are applicable to the desired product, so that this product can be assembled from the product-line assets. One of the most difficult tasks is the translation of market or customer requirements and goals into the concrete set of features that best match them. Several aspects affect feature selection for a certain product: dependencies and constraints among features, the desired degree of *product quality*, and economic cost. Moreover, different stakeholders are capable of selecting external (visible to the customers and/or marketing people) and internal features (necessary to realize external features, but not visible). In product lines with a large number of features, which are very common in practice, feature selection becomes an increasingly difficult task, and may result in *invalid*, *inappropriate* or *inefficient* configurations.

* Corresponding author.
E-mail address: damiano@fi.upm.es (D. Zanardini).

http://dx.doi.org/10.1016/j.jlamp.2015.08.003
2352-2208/

Table 1
Support for feature selection.

Main characteristic	Support type	NF concerns	Underlying technology
Multi-level staged [22]	Interactive	Security	Specialized FMs
Probabilistic [23]	Interactive	No	Conditional probabilities and legal Joint Probability Distributions
Dynamic [39]	Automatic	No	Binding analysis and reconfiguration strategy
Multi-step [57,58]	Automatic	Cost	Constraint Satisfaction Problem
Polynomial-time [56]	Automatic	Yes	Multi-dimensional Multi-choice Knapsack Problem
Fast selection time [28]	Automatic	Yes	Genetic algorithm (repair operator and penalty function)
Business concern annotation [51]	Automatic	Yes	Hierarchical Task Network
Multi-view [31,1,32]	Interactive	No	Workflow management tool
Feature-wise and variant-wise properties [47]	Mostly automatic	Yes	Constraint Satisfaction Problem
Domain experts' judgment [59]	Interactive	Yes	Analytic Hierarchical Process

Several authors have contributed to the research on feature selection (Table 1). We have analyzed the proposed approaches in terms of the type of support (either interactive or automatic), the non-functional concerns that are taken into consideration, and the underlying problem-solving technology.

Concerning *Support Type*, interactive product configuration uses the rules provided by the feature model to propagate configuration choices made by the user [23], whereas automatic product configuration provides a set of configurations that satisfy the rules and the user's requirements and constraints. The selection of features in our resource–usage–aware configurator is mainly automatized. However, the user has a central role providing not only information on concerns (e.g., memory consumption) and constraints (e.g., that the cost has to be lower than x), but also on the key features of the product. *Key features* are those features which are required by the customer as a crucial part of the desired products, similarly to user-selected features included in the input partial configurations of the tool implemented by Sincero et al. [48]. If their presence does not infringe any rule, then the configurator will not propose deselecting them in any of the provided solutions. On the one hand, this information is essential for the efficiency and effectiveness of a product configurator. On the other hand, it provides an interesting balance between automatic and interactive configurations. Tun et al. [53] proposed an approach to systematically relate requirements to features that uses three separate feature models (requirements, world context and specifications) and respective links between them. Our approach addresses this issue by asking the user for key features and quality concerns (requirements and world context), and proposing configurations (specifications) that include such key features and optimize the quality concerns.

As regards *Non-Functional Concerns*, several approaches take into consideration cost constraints, but only few of them consider quality concerns [22,56,51,28,47] as we do. There are two crucial aspects in this context: (1) quality-aware configurations require modeling quality variability; and (2) it is necessary to provide support or guidance on how to obtain quality indicators. Etxeberria et al. [24] presented a survey on existing approaches for specifying variability in quality attributes. The six approaches (Goal-based model [26], F-SIG [33], COVAMOF [14], Extended Feature Model [12], Definition Hierarchy [38], and Bayesian Belief Network [60]) are compared according to the requirements defined by the authors for a quality-variability modeling approach. Our resource–usage–aware configurator adopts the Extended Feature Model approach, because this approach does not require the learning of additional/new notations by practitioners, which will promote the adoption of our approach in practice.

Regarding the third aspect used to compare the approaches in Table 1, namely, the *Underlying Technology*, our case study was built upon the CSP (Constraint-Satisfaction-Problem) solver called *Choco Java*, because: (1) the mapping of the product-configuration problem into CSP [58] is intuitive; and (2) there are translators from CSP into *Satisfiability Modulo Theories* (SMT), which can be adopted to address quality issues of the underlying technology, if required. However, any other underlying technology capable of dealing with quality annotation of features (e.g., the one used by Soltani et al. [51]) could have been used, which includes visualization and exploration techniques such as the ones proposed in [40].

This paper focuses on obtaining quality indicators of performance for features and/or product configurations that can be used to guide product configuration. *Performance* (a.k.a. *resource consumption* or *resource usage*) is a frequently desired quality for software artifacts. In our implementation and case study, the quality metrics we use to estimate the degree of performance of a product are either the amount of allocated memory (*memory consumption*) or the number of *executed instructions*. It is important to point out from the beginning that, unlike related work, the presented techniques rely on *static* resource–usage analysis, i.e., quality indicators are obtained without actually executing the code and refer to all possible inputs (not just to a few specific workloads, as in existing approaches).

We discuss and compare four strategies for *resource–usage–aware configuration* of software product lines. Many ideas behind such strategies are well-known; one of them is actually infeasible and is only presented in order to start the discussion. However, all these strategies are applied with static analysis in mind, which is something not discussed in existing works. The common idea in all strategies is the use of resource–usage estimates as a *heuristic* for guiding the automatic selection of features. The crux is the use of an automated static resource–usage analyzer (e.g., [27,30,6]) providing estimates

of the resource consumption of software artifacts. Such estimates are used to guide the configuration process towards more efficient products, while keeping all key features (as they are essential from the user's point of view), and also adhering to user constraints and to the dependencies and constraints specified in the feature model.

1.1. Summary of contributions

Our main contribution is the notion of resource–usage–aware configuration that relies on the rigorous formal technique of automated static resource–usage analysis in order to assist configuration. This overall contribution breaks down in different strategies that realize such a notion of resource–usage–aware configuration, a prototypical implementation of the most practical strategy, and a preliminary experimental evaluation.

- We first discuss a strategy for *Product-Based Analysis*, in which the resource consumption of (selected) products is estimated after product configuration, i.e., resource–usage analysis is run *a posteriori* on (selected) product configurations. This strategy (generate all products and analyze each of them) is well-known in principle; obviously, it is not scalable at all, and cannot be used in practice. However, it is useful as a starting point.
- We then discuss a strategy for *Partial-Product Analysis*, in which the resource–usage analyzer is invoked on *partial products* obtained after the selection of features along the configuration process. In this context there is an interwoven interaction between configuration and resource–usage analysis. This strategy is new; however, it is not scalable unless some clever way to analyze incomplete code, perform incremental static analysis and prune the tree of partial configurations is found.
- The third proposed strategy is *Feature-Based Analysis*, where the resource–usage analyzer is run *a priori* to estimate the impact that each feature may have on the resource consumption of products. This is achieved by (1) generating *minimal products*, one for each feature, which include that same feature and the minimal set of features needed to get a valid configuration; and (2) analyzing them one by one taking into account the portion of code which is affected by the selection of the feature under study. This strategy is similar to Siegmund et al. [45–47] as regards the definition of minimal products. However, there are a number of relevant differences: (1) it is applied to static-analysis results instead of the execution on a specific workload, which makes things much more delicate; (2) possibly (Section 4.2), only a small part of the code is analyzed; and (3) the way minimal products are studied is not the same: no comparison between a product with a given feature and the same product without that same feature is made; instead, resource–usage annotations are built (see below) from statically analyzing relevant portions of the code.
- In *Feature-Based Analysis with Interactions*, we enrich the previous feature-based analysis (which considers the resource contributed by each feature in isolation) with resource–usage information gathered from the interaction of features. In order to detect which features may interact, we can perform a pre-process to identify interactions that affect performance.
- We report on a prototype implementation of a feature-based analysis (third strategy) which uses the SACO static resource–usage analyzer [6,5] to infer resource–usage estimates, and annotates features with *resource–usage annotations* which are then used by the configurator to suggest a valid product configuration that best fits the quality constraints provided by the user. The whole process of resource–usage–aware configuration is fully automatic.
- We have applied our implementation to an industrial case study that provides search and merchandising services. While product-based analysis of our case study requires the analysis of 768 products to obtain resource–usage annotations, we will see that feature-based analysis only needs to generate and analyze 13 products. Our experiments show that it is feasible to infer resource–usage annotations for all optional features in a fairly efficient way. Annotations are then used by the product configurator to configure a product that meets the user's constraints on performance.

Let us emphasize that, from the point of view of the static analysis component, most of the general principles behind the four strategies date back to early work on static analysis (see [19] and its references). In particular, the first strategy for product-based analysis corresponds to the original analysis (or whole-program analysis) [18]. The second one is based on principles of incremental static analysis [29,9]. The last two ones are related to modular static analysis [17].

As already pointed out, there is related work in the area of obtaining indicators of performance for features and/or product configurations that can be used to guide product configuration [37,48,45–47]. However, as far as we know, there are no other approaches to resource–usage–aware configuration that use automated static resource–usage analysis to assist configuration. For example, Siegmund et al. [45–47] do not use static analysis; instead, the execution of software artifacts for specific workloads is performed. All in all, what is unique in our work is that we base our strategies in a very complex property obtained by the rigorous formal technique of automated resource–usage analysis. The complexity (and uncomputability) of the property under study is clear since the inference of resource guarantees implies proving termination, which is considered as a difficult property by itself to infer. What is discussed in each strategy, and constitutes part of our original contribution, is how the resource–usage guarantees are obtained, and what are the advantages and disadvantages of each setting.

1.2. Organization of the article

The rest of the article is organized as follows. Section 2 provides an overview of the SPLE paradigm in the context of a case study used in this article for discussion. Section 3 outlines some essential notions of product configuration which later are needed to present our resource–usage–aware strategies.

Section 4 overviews the main notions of static resource–usage analysis. In the article, we use an out-of-the-box resource–usage analyzer that does not need to be changed to our needs. The section thus focuses on describing the different parameters that the analysis often has, as well as the output of the analysis process. This background knowledge will be useful to understand the strategies.

Section 5 introduces the four strategies for resource–usage–aware configuration, and points out the advantages and disadvantages of each of them.

Section 6 reports on a prototype implementation of the feature-based analysis (the third strategy) introduced in Section 5, whereas Section 7 describes our experiments on an industrial case study. Section 8 discusses threats to internal and external validity of our approach.

Finally, Section 9 reviews related work and Section 10 concludes the article.

The present work continues the line of work started by mostly the same authors in a previous publication [55].

2. SPLE on a case study

SPLE is a software-development paradigm characterized by two main processes: (1) *Family Engineering*, where product-line assets that are part of the product-line infrastructure are created; and (2) *Application Engineering*, where these assets are reused to create specific products according to customer requirements [41]. The process of Application Engineering becomes a *Product-Derivation* process when it is mainly concerned with the configuration of a product and its automatic derivation from the product-line assets.

For the sake of concreteness, this section presents excerpts of product-line assets from an industrial case study that has been developed using the ABS tool suite.[1] However, the ideas developed in this article are also applicable to other feature-oriented SPLE formalisms.

2.1. Case study

The *Fredhopper Access Server* (FAS) is a distributed and concurrent system that provides search and merchandising services to e-Commerce companies. Briefly, FAS provides to its clients structured search capabilities within the client data. FAS is structured as a set of live and staging environments. A live environment processes queries from client web applications via web services, with the aim of providing a constant query capacity to client-side web applications. A staging environment receives data updates in XML format, indexes the XML, and distributes the resulting indices across all live environments according to the replication protocol implemented by the *Replication System*. The Replication System consists of a *SyncServer* at the staging environment, and one *SyncClient* for each live environment. The *SyncServer* determines the schedule of replication, as well as its contents, while every *SyncClient* receives data and configuration updates. There are several variants of the Replication System that were developed as a software product line; one of them is used as a running example in this article (the source code of the case study can be found in the ABS website).

2.2. Feature models

A *feature model* [35,21,42] represents a hierarchy of features, which are properties of domain concepts relevant to some domain stakeholder and used to discriminate between concept instances. Table 2 summarizes the general concepts in feature models. The hierarchy of features is organized as a tree: it starts from a `root` feature, which has a group of sub-features. An "AND", "OR", or "XOR" (alternative, exclusive "OR") relation can hold between features in the same group.[2] In an "OR" group, it is also possible to set a minimum and maximum number (n_1 and n_2 in Table 2, where $*$ means "unlimited") of features that have to be present in any product. A feature can be either *mandatory*, if it is common to all possible instances, or *non-mandatory*, if it is marked as optional (`opt` in Table 2) or belongs to an "OR" or a "XOR" group. In addition to the hierarchical relations, *cross-tree relations* control the selection of non-mandatory features: If a feature f_1 is selected and there is a relation "f_1 requires f_2", then f_2 has to be selected too. In contrast, if f_1 is selected and there is a relation "f_1 excludes f_2", then f_2 has to be deselected.

The *Micro Textual Variability Language* (μTVL) [15] is a text-based feature modeling language that extends a subset of TVL [16]. Table 2 shows its main constructs. A feature model is represented textually as a tree[3] of nested features, each with an optional collection of boolean or integer attributes. Additional cross-tree relations can also be expressed.

[1] ABS (Abstract Behavioral Specification) website: http://www.abs-models.com.

[2] In other approaches, an "AND" group can be simply a set of mandatory of optional (but not alternative) features with the same parent node in the feature tree.

[3] Actually, a feature model can be represented in μTVL as a *forest* where the roots of all trees are considered to be related by "AND".

Table 2
Main μTVL constructs.

Relation	μTVL construct
AND	**group allof**
OR	**group** $[n_1..*]$, **group** $[n_1..n_2]$
XOR	**group oneof**
Cross-tree	**require**, **exclude**, **ifout** and logical operators !, ||, &&, $\rightarrow$ and $\leftrightarrow$

```
root ReplicationSystem {
 group allof {
  Installation { group oneof { Site, Cloud } },
  Resources {
   group allof {
    opt Client{ Int c in [1..30]; Site -> c<=10; },
    opt Server{ Int c in [1..30]; Site -> c<=10; }
  } },
  JobProcessing {
   group oneof { Seq, Concur{require: Cloud;} } },
  ReplicationItem {
   group allof { Dir, opt File, opt Journal } },
  Load { group allof { /* more features */ }}
 } }
```

Listing 1. μTVL model of the Replication System.

Listing 1 shows an excerpt of the μTVL feature model for FAS. The Replication System has mandatory features (e.g., `ReplicationSystem`, `Installation`, `Dir`), plus a number of optional features. `Site` and `Cloud` are alternative features, as well as `Seq` and `Concur`. The selection of `Concur` requires that `Cloud` is also selected. Some features like `Client` have an integer parameter `c` whose value must be between 1 and 30; moreover, `c` cannot be greater than 10 whenever `Site` is selected.

2.3. Feature implementations

Feature implementations specify at the level of source code how each feature contributes to the behavior of the final product. Several approaches have been used to this end, such as *aspect-oriented* [36], *feature-oriented* [11], and *delta-oriented* programming [44]. This paper refers to delta-oriented programming, and the Replication System has also been implemented using this technique; however, most aspects of the discussion are amenable to other approaches. For example, the global approach to static analysis (Section 4.1) analyzes whole products, so that it does not depend on how they are generated. On the other hand, the local approach (Section 4.2) does depend on how variability is implemented, but the analysis of code in delta modules is not conceptually different from analyzing code in feature modules.

The features of the Replication System have been implemented using *delta-oriented* programming. The implementation of a software product line in delta-oriented programming is divided into a *core model* and a set of *delta modules* (or *deltas*). The (possibly empty) core model consists of the classes that implement a complete product of the product line, while deltas describe how to change the core model to obtain new products. The choice of which deltas have to be *activated* (i.e., applied to the source code) is based on the selection of desired features for the final product. The *Delta Modeling Language* (DML) [15] is used to define deltas, and provides constructs for specifying how the delta modifies the source code, such as `adds`, `removes` or `modifies`, which can refer to classes, interfaces, methods, etc. For instance, a delta can add a new class, providing its complete declaration, or modify one by specifying which methods or attributes have to be added, removed or modified. Listing 2 shows an excerpt from a delta module of the Replication System in which, among other things, a new class `ReplicationSystem` is added and the class `ReplicationSystemMain` is modified. The language in which the modules are programmed is ABS [34], a language which has been recently defined for developing distributed concurrent systems. The sequential part of the language is similar to Java, but it also includes a functional sub-language to define data types. The concurrent sub-language is based on the *actor* concurrency model [2].

2.4. Linking feature models to feature implementations

A feature-oriented product-line infrastructure is composed, at least, of a feature model and the code that implements the features in it. The *Product-Line Configuration Language* (CL) [15] links feature models specified in μTVL with deltas in order to provide a specification of the variability in a product line. A product-line configuration consists of a set of features

```
delta ReplicationSystemDelta;
  adds data JobType = Replication | Boot;
  adds type ClientId = Int;
  adds class ReplicationSystem(
     [Final] Int maxUpdates, ... ,
     SyncServer getSyncServer() { ... }
     SyncClient getSyncClient(ClientId id) { ... }
     Unit run() { ... }
  }
  modifies class ReplicationSystemMain {
     adds Unit run() {
        List<Schedule> schedules=this.getSchedules();
        Set<ClientId> cids = this.getCids();
        Int maxJobs = this.getMaxJobs();
        Int maxUpdates = this.getMaxUpdates();
        new cog ReplicationSystem(
          maxUpdates,schedules,maxJobs,cids); }
  }
```

Listing 2. A delta module of the Replication System.

```
productline PL;
  features ReplicationSystem, Resources, ... , Data;

  delta ReplicationSystemDelta when ReplicationSystem;
  delta ResourcesDelta
      after ReplicationSystemDelta when Resources;
  delta ClientDelta(Client.c)
      after ResourcesDelta when Client;
  delta DataClientNrDelta after ClientNrDelta,DataDelta
      when Data && ClientNr;
```

Listing 3. CL specification of the Replication System.

```
product DefaultProduct(
  ReplicationSystem, Installation, Resources,
  JobProcessing, ReplicationItem, Dir, Load, Schedule,
  // non-mandatory features
  Site, Seq);

product TwoClients(
  ReplicationSystem, Installation, Resources,
  JobProcessing, ReplicationItem, Dir, Load, Schedule,
  // non-mandatory features
  Site, Seq, File, Journal, ClientNr{c=2,j=5},
  Update{u=3}, Search{d=10,l=20}, Business{d=10,l=20});
```

Listing 4. Two products in the product line.

assumed to exist, and a set of *delta clauses*. Each delta clause specifies a delta and the conditions for its application, called *application conditions*. These conditions contain (1) propositional formulas over the set of known features and attributes (`when` clauses), and (2) a partial ordering on deltas (`after` clauses, shown in the listing below) which specifies the order in which deltas have to be applied when some feature is selected. When the condition holds for a given product, the delta is said to be *active*. The partial ordering indicates which deltas, when active, should be applied before the considered delta. Listing 3 provides an excerpt of the CL specification of our product line.

2.5. Product specifications

Product specifications are used to define the products of a product line by stating which features should be included in each of them and setting the feature attributes when needed. This provides traceability and supports automatic derivation of products from the product-line infrastructure. Listing 4 shows two products from the product line of the Replication System using the *Product Selection Language* (PSL) [15].

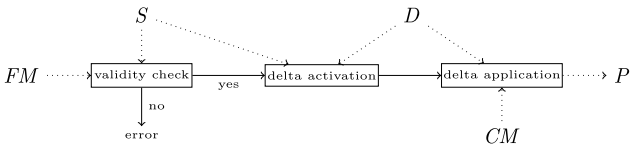


Fig. 1. The whole process of product generation.

2.6. Product generation

After having introduced all previous steps, we can define product generation (Fig. 1). Given a feature model *FM*, a core model *CM*, a set of deltas *D*, a product-line configuration *C*, and a product specification *S*, the following steps are systematically performed to build the final software product: (1) Check the product specification *S* against *FM* for validity in order to assure that the set of features in *S* obey the relations provided by *FM*; (2) use *C* to activate the deltas from *D* with valid application conditions according to *S*; (3) apply the active deltas to the core model *CM* in the prescribed order. Applying all active deltas yields the final product *P*.

3. Product configuration

It is not the purpose of this article to formalize product configuration; rather, this section just introduces the basic notions needed to accurately describe the strategies for resource–aware configuration in SPLE.

3.1. Definition

Given a product-line *PL* with a feature model *FM*, *product configuration* is the process of selecting those features that comply with *FM* and fulfill the stakeholders' requirements, which results in the product specification *S*.

The solution provided by the configurator is a set of candidate configurations $C_1 \dots C_n$, where each C_i is defined as a set of features $\{f_1, \dots, f_m\}$ (optionally providing initial values for attributes). All configurations include the set of mandatory features, and must be valid with respect to the feature model.

Example 3.1. Listing 4 shows two candidate configurations for FAS, which, in the above notation, are represented as:

$$C_1 = \left\{ \begin{array}{c} \texttt{ReplicationSystem, Installation, Resources, JobProcessing,} \\ \texttt{ReplicationItem, Dir, Load, Schedule, Site, Seq} \end{array} \right\}$$

$$C_2 = \left\{ \begin{array}{c} \texttt{ReplicationSystem, Installation, Resources, JobProcessing,} \\ \texttt{ReplicationItem, Dir, Load, Schedule, Site, Seq, File,} \\ \texttt{Journal, ClientNr\{c=2, j=5\}, Update\{u=3\},} \\ \texttt{Search\{d=10, l=20\}, Business\{d=10, l=20\}} \end{array} \right\}$$

Observe that in C_2 we provide initial values for the attributes of certain features.

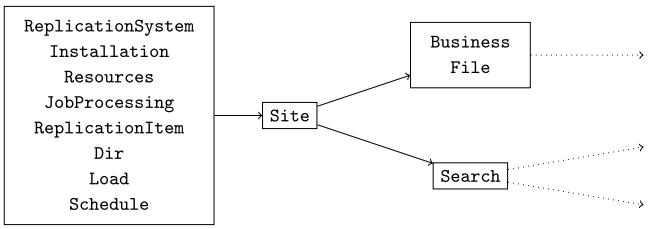
For each candidate configuration C_i, the unique associated product P_i denoted as $P(C_i)$ in the following can be automatically derived from the product-line infrastructure (Section 2.6) by taking *S* to be equal to C_i.

Example 3.2. Consider the configuration C_1 in Example 3.1; a product can be generated from it by following the CL specification in Listing 3 and applying to the core module the deltas in Listing 2. The result is a program written in the ABS language (i.e., the same language as the deltas in Listing 2) where, as it can be seen by inspecting the delta code, the class `ReplicationSystemMain` is modified by adding a method `Unit run()`.

3.2. Configuration trees

In order to define the strategies for resource–aware configuration, it is useful to view the configuration process as the construction of a *decision tree*, referred to as the *configuration tree* τ, whose nodes represent *partial configurations*. A partial configuration $C = \{f_1, \dots, f_n\}$ is a set of features corresponding to a valid product, or a subset of it. Each node is labeled by a partial configuration which is a superset of partial configurations labeling ancestor nodes. Thus, traversing a *path* $C \rightsquigarrow C'$ in the tree corresponds to adding features progressively, until a valid product is possibly obtained in C'. The edge from a node *C* to its direct child C' represents a minimal increase in the size of the feature set: more than one feature can be added in a single step because of cross-tree relations in the feature model: e.g., it might be the case that one cannot select f_{n+1} without selecting $f_{n+2}, \dots, f_m$. Nodes can have several children; e.g., both C' and C'' may be children of *C* when we choose among optional or alternative features; this may happen if C' has an optional feature that C'' does not add, or C' adds the feature f' to *C* while C'' adds the feature f'', and f' and f'' are alternative. The root of the tree is labeled by the set of mandatory features.

Example 3.3. The picture below shows a small portion of the configuration tree for the case study:



where, for each node, only new features (i.e., features which were not selected in ancestor nodes) are represented. Note that `File` is required by `Business`, so that there can be no node only containing `Business`. Note also that no feature set in this part of the tree is a valid product, because neither `Seq` nor `Concur` have been selected so far; however, all feature sets are subsets of valid configurations.

Given the product-line infrastructure *PL*, we rely on two tools coping with the variability in the feature model:

- A generic configurator invoked as *Configurator*(*PL*) is able to generate all valid configurations, relying on the feature model.
- A partial configurator *TreeBuilder*(*PL*) progressively computes a configuration tree τ as described above. The computation is progressive since, in general, there is no need to compute the whole τ. Instead, the tree can be built as specified in Section 5.2.

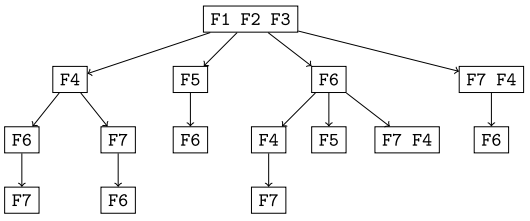
In the following, *ValidConf* denotes the set of valid configurations, and *PartConf* denotes the nodes of a configuration tree (i.e., the set of configurations, either partial or complete).

Example 3.4. For the case study, *Configurator* finds 768 valid configurations, i.e., *ValidConf* contains 768 products. This number comes from observing that `Site` and `Cloud` are alternative, as well as `Seq` and `Concur`, but the latter requires `Cloud`, so that each product selects one of the following three features sets: {`Site`, `Seq`}, {`Cloud`, `Seq`}, and {`Cloud`, `Concur`}. Moreover, there are nine optional features, thus originating 512 possibilities for each of the three selections above. The final number n is 768 instead of $1536 = 3 \cdot 512$ because of constraint in the feature model: of the 16 combinations derived by the selection of `Business`, `Data`, `File`, and `Journal`, only 8 represent valid products, so that $n = 1536 \cdot 8/16 = 768$.

Example 3.5. The configuration tree for our case study has several millions of nodes.[4] To understand where the number of nodes comes from, consider a much smaller example of a feature tree:

```
root F1 {
  group allof {
    F2 { group oneof { F4, F5 } },
    F3 { group allof { opt F6, opt F7 { require: F4; } } } } }
```

In this case, there are 3 mandatory features (`F1`, `F2`, and `F3`), two optional features (`F6` and `F7`) with cross-tree relations, and an alternative group (`F4` and `F5`). The resulting configuration tree has 15 nodes:



[4] Exactly, 491 882 570. This number has been obtained by using a Prolog program which simulates the generation of all possible partial configurations, and counts them.

4. Static resource–usage analysis

In our approach to *resource–aware configuration*, resource–consumption estimates are computed by an off-the-shelf resource analyzer and used to select the most promising configuration candidate(s). Two approaches to static resource–usage analysis will be part of the following discussion.

- The first, *global* approach considers a product *as a whole* and tries to infer information about complete executions.
- The second, *local* approach only considers some relevant parts of the code (below, the footprint) in isolation, based on the features under study.

4.1. The global approach

The global analysis of a program relies on a *generic* resource–analysis tool *Analyzer*, which, given a code fragment P, an entry method *entry*, and a resource metric of interest R, is invoked as $Analyzer(P, entry, R)$ and analyzes the resource consumption of *entry*, as well as those n methods transitively invoked from it, w.r.t. R. As a result, it returns an *upper bound* u to the resource consumption of *entry*. The existence of a global *entry* method that corresponds to the *main* method is assumed, so that u describes the resource usage of executing a product as a whole. The upper bound is a *sound* worst-case approximation of the actual cost: it is guaranteed that no execution of *entry* (for any possible input data) can consume more than u resource units. Importantly, this is in contrast with the results obtained by dynamic performance analysis, which studies the resource consumed by a particular execution on a given set of input data. Thus, static resource analysis ensures sound bounds in a system when only one execution in a million can lead to a high resource consumption, while this anomalous case would be probably missed by dynamic analysis. To compute an upper bound for the entry method implies, in general, producing upper bounds for every method possibly invoked by *entry*; the following example refers to one of such intermediate upper bounds.

Example 4.1. Consider a fragment of method `Unit transferItems(Set<File> fileset)` showed in Listing 5, which is part of the FAS case study. This method has been pointed out in previous work [13] as a hot spot in the execution time of the case study. This method traverses the set of files that receives as input parameter, and performs a number of operations on each element of the set. It is not relevant for our purposes to understand the behavior of the method, which includes also primitives for concurrency (like future variables, *await* operations and asynchronous calls) that are completely outside the focus of this work. The important point is the external `while` loop which traverses the set of files (parameter `fileset`) using an iterator and, at each loop iteration, invokes some auxiliary operations that will consume additional resources.

Let us analyze its resource consumption using the SACO tool [6], an implementation of *Analyzer* for ABS programs. We select the cost model that counts the number of steps, since this is the metric which is most related to execution time. SACO returns the following *asymptotic* upper bound:

$$size(\texttt{fileset}) * size(\texttt{rdir})^2 + size(\texttt{fileset})^3 * size(\texttt{rdir})$$

which is a polynomial of degree 4 on the size of the argument `fileset` and the class field `rdir`.

```
Unit transferItems(Set<File> fileset) {
   while (hasNext(fileset)) {
     Pair<Set<File>,File> nf = next(fileset);
     fileset = fst(nf);
     File file = snd(nf);
     FileSize tsize = fileContent(file);
     Fut<Unit> rp = job!command(AppendSearchFile); await rp?;
     Fut<Maybe<FileSize>> fs = job!processFile(fst(file));
     await fs?;
     Maybe<FileSize> content = fs.get;
     FileSize size = 0;
     if (isJust(content)) {
          size = fromJust(content);
     }
     if (size > tsize) {
          rp = job!command(OverwriteFile);
          await rp?;
          rp = job!processContent(file);
          await rp?;
     } else { .....
       // omitted a fragment of the method
   }
```

Listing 5. Excerpt of method transferItems of case study.

The aspects of static resource analysis which are relevant to our study are the following:

1. Upper bounds are cost expressions that might include polynomial, logarithmic, exponential subexpressions (and any combination of them).
2. For simplicity, in the example the result of SACO was shown in asymptotic form (or "big-O" notation), i.e., all constants have been removed as the expression was rather large and difficult to read. In general, the result provided by the analyzer is a precise upper bound that also includes constants.
3. The upper bound is given in terms of the size of the input parameters (e.g., size(`fileset`)) and of the class fields (e.g., size(`rdir`)). This is the case for the upper bound of most methods whose resource usage is not constant.
4. In order to compare the resource usage of two fragments of code, we need to be able to compare upper bound expressions of the above form. This problem has been studied in previous work [7]; therefore, the existence of an operator "$<$" which allows comparing two upper bounds is assumed.
5. The upper bound is ensured to be *correct*, i.e., it is a safe approximation of the worst-case resource consumption of running the program for any possible input data.

4.2. The local approach

In some of the strategies described in Section 5, it makes sense not to consider a program as a single piece of code to be executed from its entry method. This happens because (1) the code is actually incomplete, as when not all features leading to a valid configuration have been selected yet (Section 5.2); or (2) the focus is on the specific impact of a single feature on resource–usage behavior (Section 5.3).

On one hand, if the program code is incomplete, then analyzing the entire code is, in general, not possible because of inconsistencies in the code itself. On the other hand, the strategy presented in Section 5.3 only focuses on the part of the code which is in the product *just because* a certain feature have been selected: the feature *footprint* (defined below). In both cases, methods (either all currently available methods, as in the case of incomplete code, or the footprint of some feature, as in the case of feature-focused analyses) can be extracted from the domain artifacts (the core code or the delta modules) and analyzed separately by a call $Analyzer(PL, m, R)$[5] to the analyzer for each method m under study. However, it is important to point out that, although each method m is analyzed separately, other methods invoked by m are also considered in the result.

Not surprisingly, the local approach to static resource–usage analysis is not able to provide a sound, global upper bound to resource consumption. Instead, the result of the analysis is a set of *resource–usage annotations*, one for each method under study. Such annotations have to be transformed and combined into a useful piece of information (see below). For example, the implementation described in Section 6 does the following:

- It statically analyzes all methods in the footprint of the feature under study, obtaining a resource–usage upper bound for each of them (taking other invoked methods into account).
- It transforms each upper bound into a numeric *resource–usage annotation*, based on asymptotic complexity: exponential upper bounds are assigned a higher number than linear or constant ones.
- It combines annotations by *arithmetic mean* into a measure of the resource–usage of the part of the code under study.

Computing footprints. The notion of the *footprint* of a feature f has to be defined: the goal is to identify the parts of the code whose presence in the final product depends on the selection of f. This is done by collecting all the deltas[6] $\delta_1..\delta_n$ which *could be active* when f is selected, and, for each δ_i, computing the set M_i of methods which will be in the final product due to the activation of δ_i. The definition of a product line (Section 2.4) specifies which deltas must or could be active when f is selected. Conservatively, all deltas whose associated delta clause has an application condition where f occurs are considered; this is an approximation, since it can be the case that a delta δ is only active when both f_1 and f_2 are selected, so that the selection of f_1 does not imply the activation of δ. For instance, the delta clause `delta DataClientNrDelta` in Listing 3 states that this delta is active and must be applied after the deltas `ClientNrDelta` and `DataDelta` (if they are also active) whenever the expression `Data && ClientNr` holds. Conservatively, this delta is considered when computing the footprint of `Data`, even though the effective activation of this delta also requires the selection of `ClientNr`. Once the set of deltas that can be active when f is selected has been obtained, an inspection of all delta declarations collects, for each δ_i, the set M_i of methods added or modified[7] by δ_i. The union of all the M_i is called the *footprint* of f, denoted by $Footprint(f)$, and can be computed statically.

[5] The use of PL instead of P as the first argument of the call means that the code of m and the methods it invokes is extracted from the domain artifacts, not an individual product.

[6] Note that, if feature-oriented programming were used instead of delta-oriented programming, then the footprint of a feature could be (more easily) extracted from the corresponding feature module.

[7] Methods that are removed by a delta are not included because there is no code to be analyzed anymore.

Example 4.2. When studying the feature `File` according to the strategy of Section 5.3, a product is generated, which includes 138 methods. However, the footprint of the feature `File` only contains nine methods, which are those modified or created by delta `FileDelta` which

- adds class `ReplicationFilePattern`, containing six methods;
- modifies one method in class `ReplicationSnapshotImpl`;
- modifies two methods in class `TesterImpl`.

If the footprints of all features from a certain set (e.g., all non-obligatory features, as described in Section 5.3) have to be analyzed, then it is possible, up to a certain extent, to eliminate redundancies by considering which deltas are involved in any footprint. Suppose a delta δ may be activated both by the selection of f_1 and by the selection of f_2, i.e., both f_1 and f_2 appear in its delta clauses. Suppose also that the method m and every method which is (directly or indirectly) invoked by m are modified or created by δ. In this case, the analysis result obtained for m during the analysis of the footprint of f_1 can be safely reused when analyzing the footprint of f_2, thus resulting in a reduction of the overall computational effort.

Generating and combining resource–usage annotations. As pointed out above, the last two steps of the local approach to static analysis generate and combine resource–usage annotations to obtain a measure of the resource usage of the code under study. The presented implementation annotates methods with resource–usage annotations representing *asymptotic al complexity* (in this paragraph, this choice will be called (1a)), and uses the *arithmetic mean* to combine annotations (in this paragraph, (2a)). However, the choice of the best way to generate annotations and the best combination function is highly application-dependent, and a thorough discussion is outside the scope of this paper.

With respect to the generation of resource–usage annotations from analysis results, the following alternative approaches can be considered:

(1b) If the expected *probability distribution* of the input to the method under study is known (whether exactly or not), then the upper bound computed by the static analyzer, which is a function of the input size, can be easily converted into the expected resource usage by replacing input variables by their expected input size. For example, if the upper bound is $2n^2$, then knowing that the average value for the input size is 100 may lead to the numerical resource–usage annotation $2 \times 100^2 = 20\,000$.
(1c) If the *maximum value* of the input size is known, then an annotation can be generated from the upper bound similarly to (1b).

As regards the combination of resource–usage annotations, the reason for using the arithmetic mean can be the interest on a measure which takes all methods into account. However, there are several reasonable alternatives:

(2b) If some more information is provided, then the *weighted mean* can be used; for example, the weight associated to a method may depend on how often it will be executed, according to the domain-dependent information that could be available at the moment.
(2c) If the goal is to avoid computations whose resource consumption is unacceptable, then the *maximum* can be used. For example, such a combination function could help to select code where no methods can have exponential complexity.
(2d) In presence of a hard limit to resource–usage (e.g., the amount of available memory), a *threshold* function can indicate if computations will exceed the limit: if m is the maximum amount of resources available, then the threshold t can be defined as $t(x) = 0$ if $x \leq m$, and $t(x) = 1$ otherwise. This combination function can only be used if (1c) is chosen as the method to generate annotations.

In the following, the operator $\oplus$ will denote the function which combines resource–usage annotations into a single one, corresponding to a measure of the resource usage of the code under study (whether a partial product or a footprint).

5. Strategies for resource–usage–aware configuration

This section discusses different ways to carry out the interaction between *Analyzer* and *Configurator*, and points out advantages and drawbacks of each of them.

Resource–Aware Configuration is a problem of optimization whose goal is to select the optimal product in terms of resource consumption. In general, we say that an algorithm is *optimal* if it always chooses the optimal product; needless to say, optimality cannot be obtained in general because exact resource consumption is not a computable property. On the other hand, an algorithm is said to be *optimal modulo Analyzer* whenever it *would* be optimal if *Analyzer* were perfect, i.e., if the results provided by the analyzer were always exact. Optimality modulo *Analyzer* amounts to say than the inevitable loss of precision only comes from resource–usage analysis. It is important to point out that the notions of optimality and optimality modulo *Analyzer* both rely on establishing how two products are to be compared w.r.t. their cost: given two products P' and P'', it can be case that the goal is to select the best for some specific input, or *on average*, or on the worst case, or w.r.t. any other reasonable requirement.

Example 5.1. Suppose that the program P' has an exact resource consumption of $n^2 - n + 10$ where n is the size of its only parameter, and that the static analyzer gives the upper bound $n^2 + 10$, which is a good approximation of the exact cost. On the other hand, suppose that the resource consumption of P'' is $n + n$, but the static analyzer outputs $n^2 + n$.

In this case, the best product for a specific input with size $n = 5$ is P'', and the result of the analyzer leads to choose P'' as the best program. On the other hand, for $n = 20$, P'' is still the best program because $n + n < n^2 - n + 10$ when $n = 20$, but P' is chosen. Finally, in order to compare P' and P'' on average, a probability distribution of the input has to be provided.

In the following, *obligatory* features are either (1) features which have to be selected according to the product-line definition (for example, the root feature, or children of a `group allof` declaration which are not marked as `opt`); or (2) *key features*, i.e., user-required features in the sense of Section 1.

5.1. Product-based analysis

In the first strategy, *Analyzer* obtains the resource estimates directly from the final products. The process consists of three steps:

1. Given the product-line infrastructure *PL*, we first obtain the set of final (valid) configurations *ValidConf* (Section 3);
2. for each $C_i \in \mathit{ValidConf}$ containing all obligatory features, we generate a product $P_i \equiv P(C_i)$, and analyze it by running $\mathit{Analyzer}(P_i, \mathit{entry}, R)$ where R is the resource of interest;
3. the *best candidate* is the product P whose resource consumption u, corresponding to the pair (entry, u), is the minimum among all products.

This approach is conceptually simple and keeps *Analyzer* and *Configurator* completely separate.

Advantages. The main advantage of this approach is that it can be potentially implemented using existing technology since (1) there are tools that behave like *Configurator*; (2) there exist product generators for valid configurations; (3) a static analyzer *Analyzer* for final products can be used; (4) techniques for comparing upper bounds and choosing the minimum are available [7]; and (5) there is little need to design complex interactions between these components.

This strategy is the most direct, since it solves the problem of choosing between products by actually generating and studying complete products. It is optimal modulo *Analyzer* since the process of picking the best product is feasible if cost information is exact, provided the comparison criterion (worst-case, best-case, average, input-specific, etc.) is correctly specified. However, it is not strictly optimal because the best a sound resource–usage analysis can do is giving upper bounds u_i which correctly over-approximate the resource consumption of the product P_i for any possible input data. Therefore, it is not guaranteed that the chosen P_m is the best candidate, since the static analyzer performs several approximations in order to obtain a sound result, and the loss of information in the resource–usage analysis of a product can be larger than the loss in the analysis of another one. One can easily provide examples for which this leads to selecting a "best" candidate that is actually not the best. Thus, the analysis is used as a heuristic for guiding the selection rather than as a guarantee. Still, this strategy produces accurate results.

Disadvantages. The main drawback of this approach is its inefficiency. For a product line with k valid configurations, we need to invoke the product generator and the resource–usage analyzer k times. Each analysis is performed on a full product, which can be a large and complex piece of software. The results from analyzing one product cannot be reused when analyzing the next one, as there is no knowledge on which parts of the product are the same as those of previous products. Unfortunately, static analysis tools for a property as complex as resource usage are not yet developed at an industrial level: while they can handle medium-size programs, their application to commercial products is still a research challenge. In conclusion, we argue that, although this strategy is feasible in theory, it is beyond the current state of the practice.

Example 5.2. In the FAS case study, this approach involves generating 768 different products, analyzing each of them, and choosing the one that shows the best performance behavior. Most products are, in terms of lines of code, even bigger than the code implementing the whole product-line, thus making the analysis of each single product very expensive. In general, the number of products to be generated, together with their sizes (for our case study, each product has more than 2.000 lines of code), can make this task prohibitively expensive.

5.2. Partial-product analysis

It is quite natural to think of an *interleaved* cooperation between the resource analyzer and the configurator. In fact, it would be useful for the configurator to get information about resource consumption *as long as* the configuration is built (i.e., as long as features are selected), in order to give up adding features whenever the current feature set has a high probability to be inefficient. This strategy can be obtained by interleaving the work of the configurator *TreeBuilder* (Section 3.2) and

```
int entry(int x,int n){
  while (x<n) x=incr(x); return x+n; }

delta foo1{ modifies int incr(int x) { return x++; } }

delta foo2{ modifies int incr(int x) {
   int x0=x; while (x>0) {f++; x--;}; return x0++; } }
```

Listing 6. Partial-Product Analysis.

Analyzer, in such a way that *TreeBuilder* invokes *Analyzer* along the configuration process to be aware of the resource consumption associated with *partial* (i.e., in the process of being computed) configurations. This approach requires being able to estimate the resource consumption of *partial products* associated with *partial configurations*.

Given a product-line infrastructure *PL*, the *configuration tree* τ (Section 3) is partially built. Whenever a new node of τ (i.e., a partial configuration) is generated, the information about the resource consumption will allow the configurator to decide if it is worth continuing the construction of such a configuration, or if it is better to reject that path of the configuration tree. Partial-product-level analysis consists of the following steps:

1. Starting from the root node of τ a partial configuration $C \in PartConf$ is computed, and a partial product $P(C)$ is generated; whenever some C is computed, all its ancestor nodes in τ have been computed before.
2. $P(c)$ is *incrementally* analyzed by executing $Analyzer(P(C), R)$, reusing as much as possible the results inferred for the partial products corresponding to ancestor nodes.
3. It is decided whether the estimated resource consumption u for $P(C)$ is "acceptable"; otherwise, the current branch of τ is pruned (i.e., the children of C will not be considered).

In order to decide if the resource consumption along a path is acceptable, the user can set a threshold (or maximal amount of resources) *Limit* before starting configuration. Thus, in step 3, the simple check "$u > Limit$?" decides if the current branch of the tree must be pruned or not. Another possibility is to keep the results for the best product constructed so far (namely, u_{min}), and prune a branch if the resource consumption of a partial product already exceeds u_{min}.

Example 5.3. Consider the FAS case study, and suppose that the user imposes the expression $size(\texttt{fileset})^2$ as the threshold *Limit*; i.e., the set of files that are to be transferred can be traversed at most a number of times which is quadratic on its size. During the construction of the configuration tree, as soon as a partial configuration C selects a feature that triggers a delta including method `transferItems` (see Example 4.1), the threshold provided by the user is exceeded since the resource consumption of `transferItems` (namely, $size(\texttt{fileset}) * size(\texttt{rdir})^2 + size(\texttt{fileset})^3 * size(\texttt{rdir})$) is larger than $size(\texttt{fileset})^2$. Thus, the branch of the configuration tree starting at C will be pruned.

It must be pointed out that, in general, pruning a branch of the configuration tree could lead to rule out the best (complete) configuration, which could possibly correspond to a node in the pruned branch. Supposed the branch rooted at C is pruned because some feature in C contributes to get a resource consumption beyond the limit. It can be the case that a feature $f \notin C$ is added at a children node, which greatly lowers the global resource consumption. Indeed, this can happen, so that this method is not optimal, not even modulo *Analyzer*. However, this is not a common situation since it is unlikely that the impact on resource usage of two features which can be selected in the same product is opposite.

Example 5.4. Consider the partial product resulting from applying the delta `foo1` to `entry` in Listing 6. If we measure the number of instructions executed by `entry`, the resource analysis infers a linear cost, namely, $n_0 - x_0$ instructions, where n_0 and x_0 refer to the initial values of n and x, respectively. Observe that the `while` loop in the `entry` method is executed $n - x$ times, explaining the linear cost obtained. On the other hand, consider the partial product that results from applying `foo2` to the previously constructed (and analyzed) product: the resource consumption of `entry` becomes $(n_0 - x_0) * (x_0)$, which is quadratic. This is because we invoke the method `incr` inside the `while` loop of `entry` and each of the invocations executes the `while` loop of the new implementation of `entry`. The latter `while` loop performs `x` iterations, which, multiplied by the number of iterations of the `while` loop of `entry`, leads to a quadratic cost. This example reveals that it might not be accurate to prune τ, as a future modification might affect the resource consumption of a previously analyzed (partial) product. The consumption can be increased (as in the example) but also reduced (e.g., if deltas are applied in the inverse order). Therefore, optimal (modulo *Analyzer*) results can only be obtained by building full configuration trees and analyzing all the resulting complete products, which actually boils down to product-based analysis.

There is another issue about the way code can be analyzed. The above steps include a call $Analyzer(P(C), R)$ to the analyzer, but such a call is, in general, not possible by using an off-the-shelf static analyzer which aims at finding a global upper bound to the resource consumption (Section 4.1) starting from the entry method. The reason is that $P(C)$ will be usually incorrect code since the configuration C it comes from is not valid (there may be features missing, which usually

boils down to have calls to undefined methods, references to undefined fields, etc.). Instead, the local approach to static analysis (Section 4.2) can be chosen: the analyzer will study all methods in $P(C)$ whose code is syntactically correct. The generation of resource–usage annotations and their combination via some operator $\oplus$ into a measure of the global resource usage of the partial product would follow the discussion of Section 4.2.

Advantages. The main advantage of this approach comes from pruning the configuration tree and avoiding building products whose resource consumption exceeds the provided threshold. Furthermore, as (partial) products are built incrementally by activating the corresponding deltas, *incremental resource–usage analysis* [9] can be used, so that information gathered in the analysis of previous partial products is reused whenever it is valid. In a different context and for a different language, it is proven [9] that incremental resource analysis can save up to 50% of the analysis time when compared to non-incremental program analysis. In this context, incremental resource–usage analysis can be used in order to save computational effort whenever adding a feature to a partial product only affects a limited number of methods, so that previously-obtained results for unaffected code can be reused.

Disadvantages. A problem with this approach is that generating partial products is not always feasible using the current technology from the ABS tool suite since a partial product is, in general, incorrect code (some relevant parts of the code might be missing). Indeed, the product generator aims at building a final product, and, as soon as the generator finds a method that is not defined, the whole process fails. As a consequence, most nodes in *PartConf* cannot be evaluated since there is no tool for actually generating the product.

Concerning efficiency, *Analyzer* has to be invoked, in the worst case, on all nodes *PartConf* of τ, and Section 3.2 indicated that the number of nodes can be huge even for small feature trees. Therefore, this method is not practical unless the resource of interest guarantees that most part of the configuration tree will be pruned, or a better way to traverse τ is found.

On the other hand, as pointed out before, optimality is potentially lost whenever a branch in the selection tree is pruned. This is because choosing the *locally best* solution does not necessarily lead to the *globally best* solution, since a feature added in a later selection might affect the resource consumption significantly.

5.3. Feature-based analysis

With the aim of devising a more practical strategy, we consider a third possibility: assessing the resource consumption due to each feature f in the product-line by generating and analyzing a product containing only f plus the minimal number of features needed to get a valid configuration. The *minimal product* for f (denoted $P_\mu(f)$) is a valid product containing f such that removing any other feature from it leads to an invalid configuration. In general, such a product is not unique because different features could be selected from a group.

Definition 5.5 *(Minimal product).* A minimal product for a feature f is defined as follows. Note that, according to the syntax and semantics of μTVL [15], `group allof`, `group oneof` and `group [`n_1`..*]` declarations can be rewritten as, respectively, `group [`n`..`n`]`, `group [1..1]` and `group [`n_1`..`n`]`, where n is the number of children features, so that the general form `group [`n_1`..`n_2`]` is the only one to be considered. Then, the following rules are followed until the set of selected features stabilizes:

(a) $P_\mu(f)$ contains the root feature;
(b) $P_\mu(f)$ contains the feature f under consideration, and all its ancestors in the tree (that is, the parent of f, the parent of the parent, etc.);
(c) Given the declaration `group [`$n_1 \ldots n_2$`]` where f is one of the children, after $n_1 - 1$ children different from f have been chosen randomly, those of them which are not marked by an `opt` modifier are included in $P_\mu(f)$;
(d) Given any other declaration `group [`$n_1 \ldots n_2$`]` where the parent feature is in $P_\mu(f)$: after n_1 children have been chosen randomly, those of them which are not marked by an `opt` modifier are included in $P_\mu(f)$;
(e) Any feature required by this selection according to cross-tree relations is also included in $P_\mu(f)$.

This definition of minimal products is similar to $a \times min(a)$ products in related work [45]. However, their use is quite different, as the present work does no comparison between $a \times min(a)$ and $min(a)$. Instead, static analysis of $P_\mu(f)$ following the local approach is performed.

The generation and analysis of $P_\mu(f)$ needs to be performed only for features which are not obligatory, since the analysis of $P_\mu(f)$ aims at deciding whether selecting f is good from the point of view of resource usage, and this makes no sense for obligatory features because they will be selected in any case. Given a product-line with non-obligatory features $f_1..f_n$, the feature-based analysis is performed as follows: for each feature f_i

1. The minimal product $P_\mu(f_i)$ is generated;
2. The footprint of f_i is computed;

```
adds Unit foo() {
  while (myfield>0) {
    x=new Ob();
    m();
    myfield=myfield-2;
  }
}
```

Listing 7. Code corresponding to f_A.

3. For every method m in the footprint of f_i, a call $Analyzer(PL, m, R)$ is executed according to Section 4.2, where R is the resource of interest;
4. Given all the analysis results (upper bounds), resource–usage annotation are generated and combined via some $\oplus$ into a *per-feature resource–usage annotation* for f_i (Section 4.2).

Each per-feature resource–usage annotation is a representation of how well each feature is expected to behave from the resource–usage point of view. Once per-feature resource–usage annotations have been computed for every non-obligatory feature, this information is passed to the configurator in order to choose the best configuration according to resource–usage concerns.

Advantages. This methodology has a number of practical advantages with respect to the others: (1) the number of minimal products to be analyzed is much smaller than the number of products (in the FAS case study, only 13 compared to 768), thus making this strategy much more feasible than the first two ones; (2) every time a minimal product is analyzed, only a limited part of the code has to be inspected: the footprint; and (3) the whole analysis process can take place *before* the configuration begins, so that there is no need to design a complex interaction between *Analyzer* and *Configurator* (the represents an important advantage over the partial-product analysis, mainly).

Disadvantages. Clearly, this approach is not optimal because the local approach to static analysis is not. Moreover, *interactions* between different features in a product are not considered. However, as we noted before, the other two strategies are not optimal either, though the loss of precision in the product-based strategy should be smaller. How close the results of this strategy are to optimality also depends on the resource of interest, and the way resource–usage annotation are generated and combined.

Example 5.6. In the `ReplicationSystem` example, there are 13 non-obligatory features[8] (9 are marked as `opt` and 4 are children of two `group oneof` declarations), so that only 13 minimal products have to be locally analyzed. This is a great improvement over the 768 products to be analyzed in the product-based strategy. However, the feature-based approach does not allow appreciating how different features behave when coexisting in a product: for instance, `Search` and `Business` can both be selected in a given product, but no minimal product contains both.

All in all, what the feature-based methodology can provide is a heuristic that describes the performance behavior of each feature and helps the process of configuration in the challenging task of choosing one configuration which, in addition to be valid, is efficient w.r.t. some resource–usage metric.

5.4. Feature-based analysis with interactions

The main disadvantage of the previous strategy is that it ignores interactions among features. As pointed out in the literature [45], the combined presence of several features might influence performance.

Example 5.7. For instance, consider a core code with a method `Unit m() { }` whose body is empty. Consider two code fragments that belong to delta modules activated by features f_A and f_B, respectively (Listings 7 and 8), and let memory consumption be the resource of interest. If f_A is considered in isolation, the memory consumed by the above fragment of code is $size(\texttt{Ob}) * nat(\texttt{myfield})/2$, where *nat* returns either 0 if `myfield` is negative as the loop will not be executed, or the positive value of `myfield`. Observe that, at each execution, `myfield` is decremented by two, so that the number of loop iterations is $nat(\texttt{myfield})/2$. However, if we consider the interaction of both features, then the number of iterations is not the same. In particular, the call to m executes an increment of `myfield`. In this case, we have that the worst memory consumption of `foo` is $size(\texttt{Ob}) * nat(\texttt{myfield})$, which doubles the previous amount.

[8] Under the assumption that no key features are required.

```
modifies Unit m () {
  myfield=myfield+1;
}
```

Listing 8. Code corresponding to f_B.

The resource usage of two interacting features f_A and f_B, denoted by $u_{f_A \times f_B}$, can be estimated similarly to the previous feature-based strategy. In particular, minimal products $P_\mu(f_A \times f_B)$ are generated, i.e., the minimal products including both f_A and f_B, and the resource consumption of the resulting product is estimated by local static analysis, obtaining $u_{f_A \times f_B}$. In the previous example, $u_{f_A \times f_B}$ returns an upper bound $size(\texttt{Ob}) * nat(\texttt{myfield})$ for `foo`, while u_{f_A} would be $size(\texttt{Ob}) * nat(\texttt{myfield})/2$.

A strategy that considers the combination of at most k interacting features, where k is a fixed parameter, can be proposed. Consider three features f_A, f_B, and f_C, and $k = 3$: a feature-based analysis with interactions between up to k features estimates the performance of (1) three minimal products $P_\mu(f_A)$, $P_\mu(f_B)$ and $P_\mu(f_C)$; (2) three minimal products containing two interacting features, namely, $P_\mu(f_A \times f_B)$, $P_\mu(f_A \times f_C)$, and $P_\mu(f_B \times f_C)$; and (3) one for the three interacting features, namely, $P_\mu(f_A \times f_B \times f_C)$. If $k = 2$, then the last product would not be considered. The definition of minimal products for more than one feature is a straightforward extension of Definition 5.5, and the footprint of some number of features considered together is simply the union of the footprints of each feature.

As in the previous feature-based approach, *Configurator* will receive resource–usage information which guides the process of finding the best configuration. The approach which does not consider feature interactions generates a per-feature resource–usage annotation. However, considering interactions between up to k features implies that each non-obligatory feature is involved in more than one minimal product, so that several resource–usage annotation can correspond to it. The best way to go seems to provide the configurator with all the information about minimal products. Whenever *Configurator* has to choose between two configurations, the following methodology can be used. For every configuration C:

- Let F be the set of non-obligatory features included in C, and let k be the maximum number of features involved in studied interactions.
- Let $a_{f_1,\ldots,f_m}$ be the resource–usage annotation corresponding to the minimal product $P_\mu(f_1 \times .. \times f_m)$.
- The *per-configuration resource–usage annotation* for C will be $a_{F_1} \oplus a_{F_2} \oplus .. \oplus a_{F_n}$ where $\{F_1, .., F_n\}$ is a partition of F, and it is k-maximal in the sense that each element of the partition has cardinality at most k, and the numbers of elements in the partition is minimal (i.e., F is split into a minimal number of non-overlapping subsets).

This means that a configuration is analyzed by studying the minimal products which consider the most complex feature interactions (up to k). The configuration with a smaller global resource–usage annotation is preferred. In total, the number of products generated will be $z = \sum_{i=1}^{k} \binom{n}{i}$.

One interesting aspect of this strategy is that we can know a priori if two features are not interacting. For the sake of resource analysis, two features might interact only if one feature modifies a method used in the other feature. We can perform a simple pre-processing to discard the lack of interaction among features. Techniques introduced in the literature [45] can also be used to rule out feature interactions.

Advantages. This strategy is obviously more accurate than the pure feature-based analysis without interactions. Besides, the pre-processing mentioned above and the fact that we can discard spurious interactions could make it both practical and accurate.

Disadvantages. If the constant k is big or even equal to the number of features, and there is no pre-processing to discard spurious interactions, then the number of products to be studied can be the same order of magnitude as the product-based strategy, thus making the approach prohibitively expensive. Moreover, in order to design this strategy state-of-the-art configurators are not sufficient, as resource–usage annotations refer, in general, to sets of features, and have to be combined into a single per-configuration annotation.

6. Implementation of a feature-based resource–usage–aware configurator

We developed a prototype *resource–usage–aware product configurator* that implements the feature-based strategy described in Section 5.3. Fig. 2 provides an overview of its work-flow. The first phase is the *Generation of Resource–Usage Annotations*, and consists of the following steps:

(1a) Given a product-line infrastructure *PL*, the component *MinimalProductGenerator* generates the minimal products for all non-obligatory features.
(1b) The off-the-shelf SACO static resource–usage analyzer [6] is used to analyze each minimal product $P_\mu(f)$ by following the local approach (Section 4.2) on the footprint of f.

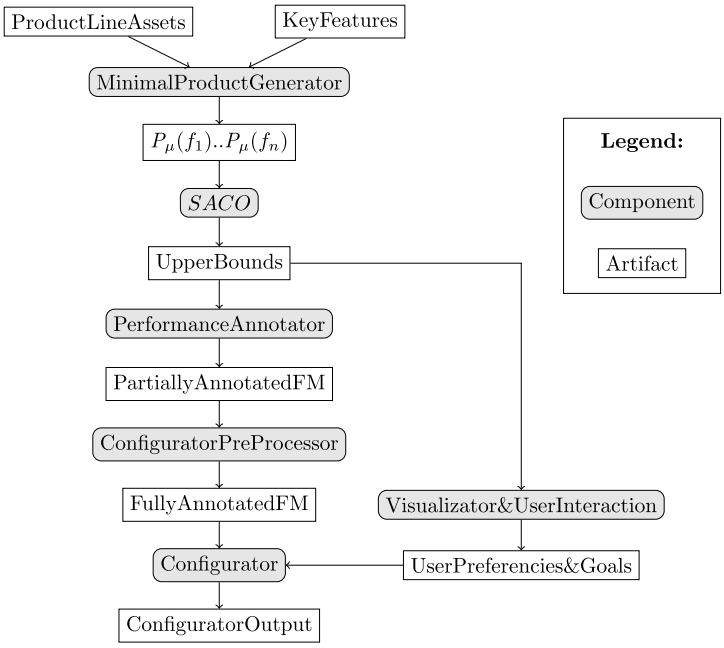


Fig. 2. The resource–usage–aware configurator.

(1c) The component *PerformanceAnnotator* takes the upper bounds on resource usage returned by SACO for every method in the footprint of f and (1) transforms each of them into a resource–usage annotation; (2) combines all the annotations into a per-feature resource–usage annotation for f; and (3) annotates all non-obligatory features in the feature tree with such annotations.

The final output of this phase is a *PartiallyAnnotatedFeatureModel*. The second phase is the *Product Configuration* itself, in which:

(2a) The *PartiallyAnnotatedFeatureModel* is pre-processed to derive annotations for upper-level features from the annotations provided by the *PerformanceAnnotator* (we note that most non-obligatory features are leaves of the feature tree).
(2b) The component *Visualization&UserInteraction* asks the user to provide his/her quality constraints and concerns.
(2c) The configurator suggests a small set of valid configurations that best fit the objective function representing the user's input.
(2d) The user selects one of those configurations (this step is not needed if step (2c) already gives a single configuration).

The final output is a PSL specification from which the product can be generated by tools available in the ABS tool-suite (Section 1).

The main decisions made during the implementation of the above components are described below.

6.1. Generation of resource–usage annotations

The main decision in this phase is which functions will be used in order to generate and combine resource–usage annotations (Section 4.2). As pointed out before, such choice is application-dependent, and the present work only suggest some possibilities.

Generation of minimal products. A minimal product $P_\mu(f)$ is computed and generated for every non-obligatory feature f. Computing a minimal product involves reading the μTVL definition of the feature tree and identifying the set of non-obligatory features. Afterwards, the features that will be included in a minimal product $P_\mu(f)$ are selected following Definition 5.5. Given the feature specification for $P_\mu(f)$, the actual product can be generated by existing tools available in the ABS tool-suite.

Example 6.1. In the FAS case study, the minimal product $P_\mu(\texttt{Cloud})$ for `Cloud` includes all obligatory features, `Cloud` itself, and either `Seq` or `Concur` (chosen randomly). This configuration is a minimal valid configuration containing `Cloud`. The minimal product for `Concur` involves selecting all obligatory features, `Concur` itself, and `Cloud`, which is required by `Concur`. Finally, $P_\mu(\texttt{Client})$ consists of all obligatory features, `Client`, one between `Site` and `Cloud`, and one between `Seq` or `Concur` (when `Concur` is selected, `Cloud` has also to be chosen).

In this case study, the total size of the code (including deltas, the product-line declaration, etc.) is around 4.000 lines. All the products (i.e., the code produced by the product generator) generated in the tests, either minimal or not, have roughly the same size. Only considering the actual code (i.e., excluding delta modules, product declarations, etc.) gives a size of around 2.000 lines for all generated products (this will be discussed later in Table 3), including the final best product selected by the configurator (2.025 lines of code). This means that generating minimal products does not need to give any significant advantage in terms of code size with respect to generate "complete" products. As pointed out above, the advantage is that minimal products are only a small part of the set of valid products.

Static analysis. Once minimal products have been generated, SACO analyzes each of them. The resource of interest is a parameter of the analyzer, which, in the current implementation of our solution, supports either *number of instructions* (at the level of source code) or *memory consumption* (i.e., the amount of memory locations used at runtime). The SACO analyzer also allows measuring data-transmission sizes, the number of requests to servers, and other interesting resource–usage metrics.

The analyzer takes the *local* approach of Section 4.2: for every minimal product $P_\mu(f)$, only methods belonging to the footprint of f are considered, and each of them is analyzed separately. The result is an upper bound u to resource usage for every analyzed method, or "unknown" if the analysis was not successful.

As resource–usage is a very complex (and uncomputable) property of software artifacts, it is unavoidable that, in some cases, the static analyzer cannot give any useful information about a piece of code, and has to *fail*, i.e., return the "unknown" value. In this respect, it is important to point out that the local approach on the footprint of some feature f makes it easier (with respect to the global approach) to analyze the code, since methods are analyzed separately and (provided a suitable function is used in the next step) useful results can be obtained even if the analyzer fails sometimes.

Resource–usage annotations. While upper bounds output by SACO in the previous step provide a precise estimate of the resource consumption, manipulating them in the subsequent configuration phase is rather complex. In a product line with a large number of products and core assets, managing such expressions grows increasingly difficult, and results become hard to interpret, especially from the point of view of the user. For instance, deciding if an upper bound u is smaller than another one u' requires the use of specific techniques [7]. The result of the comparison is often not simply a Boolean answer, but rather constraints on the input values under which the comparison can be proved: u could be smaller for some specific values of the input, and larger for others.

The implementation generates resource–usage annotations from upper bounds, in the form of numeric constants giving information of the asymptotic complexity of methods. In order to carry out this mapping, we first transform an upper bound u into *asymptotic* form ("big-O" notation). This transformation can always be applied, and can be done locally and efficiently [3]. The next step is to map the asymptotic upper bound to a resource–usage annotation, according to the following table:

Complexity of the upper bound	Annotation
constant	0
logarithmic, sub-linear or linear	100
polynomial (up to degree 3)	200
high-degree polynomial or exponential	300
unknown (the analyzer could not get an upper bound)	400

Example 6.2. Consider the upper bound obtained from analyzing the method `transferItems` in Example 4.1 which is already in asymptotic form. According to the above choice, the expression (polynomial of degree four) will be mapped into the annotation 300.

The next step is to combine (see Section 5.3) per-method annotations a_m into a per-feature annotation a_f describing the resource usage of f. In the implementation, we are currently using the arithmetic mean of the resource–usage annotations of all methods in the footprint of f.

Example 6.3. We analyzed all methods in the footprint of `File` (Example 4.2) w.r.t. the cost metrics "number of instructions". Once SACO generates resource–usage annotations for all of them, the overall annotation $a_{\texttt{File}}$ has been obtained as the average, and the result is 133. This number is obtained by analyzing each of the nine methods in the footprint: 5 of them have constant cost (the resource–usage annotation is 0), 2 of them give a low-degree polynomial upper bound (annotation 200) while the last two cannot be analyzed (annotation 400). The final resource–usage annotation is given to *Configurator* using the syntax, see Listing 9. The first instruction declares `im_numberOfInstructions` as an integer that can take values 0 or 133. The second line states that, if the feature is not selected (`ifout`), then it must take value 0. Finally, according to the last line, the resource–usage annotation is 133 whenever the feature is selected. Three lines are output for every different cost model considered in the analysis. This information is what is actually sent to the configurator in order to carry out the following phase.

```
  extension File {
    Int im_numberOfInstructions in {0,133};
    ifout: im_numberOfInstructions == 0;
    im_numberOfInstructions == 133;
  }
```

Listing 9. Performance Annotation by SACO.

```
<preprocessingMetric>
  <metricId>im_memoryConsumption</metricId>
  <compositionOperator>+</compositionOperator>
  <parentAnnotatedWithChildConsideration>No</parentAnn...tion>
  <valueRange>calculate</valueRange>
  <defaultRange>0..15</defaultRange>
</preprocessingMetric>
```

Listing 10. Preprocessing for Configuration.

6.2. Product configuration

The configuration preprocessor combines the annotations a_f provided in the previous phase. Listing 10 shows an excerpt with the definition of preprocessing options for memory consumption which is yielded in XML. It states that the annotation of a higher-level feature is done using the `compositionOperator`, which is "+" in this case.

The resulting annotation will be a formula:

$$\sum_{i=0}^{n} childFeature_i.\texttt{im\_memoryConsumption}$$

which represents the sum of the memory consumption of all children present in the configuration; if $childFeature_i$ is not selected in the configuration, then $childFeature_i$.`im_memoryConsumption` is 0. The appropriate composition function can be easily defined for each specific metric and application.

After obtaining the fully-annotated feature model, any objective function can be defined on the attributes of the root feature. The *quality concerns* provided by the user are translated into an appropriate objective function. In the objective function, the priorities of different metrics can be reflected. For example, the property `im_memoryConsumption` can be more important to the user than `im_numberOfInstructions`. The user can directly quantify how important they are absolutely or several standard approaches for eliciting prioritization can be used. The *Analytical Hierarchical Process* (AHP) is a popular approach for finding priorities from relative importance of different criteria.

In addition to the objective function, the user can also set quality constraints by providing a threshold that cannot be exceeded. Quality constraints can be related to one or more quality metrics. Once the objective function and the quality constraints are elicited, the configurator finds suitable product configurations for the user.

Example 6.4. In our example feature model, we have the feature `Client` and the user wants her product to be launched in a very thin client with respect to memory. She can set constraints on that feature specifying how much memory consumption she can tolerate, e.g., some value t. The constraint is specified as follows:

$$Client.\texttt{im\_memoryConsumption} \leq t$$

In order to find valid solutions for the configuration problem, our configurator uses the Java-based CSP solver called *Choco Java*, which converts the feature model and the objective function into a *Constraint Satisfaction Problem* (CSP), and asks the CSP solver to solve it. For eliciting the user's quality and functional requirements and for visualizing the suggested product configurations, the open source tool *FeatureIDE* was extended [55].

7. Experiments on case studies

We have implemented the generator of resource–usage annotations as an extension of the SACO analyzer: it takes as input the ABS files containing all the code, and outputs a μTVL file with the resource–usage annotations described in Example 6.3. In the FAS case study, the total size is 3.548 lines of code.

This prototypical implementation follows the third strategy (*Feature-based analysis*, Section 5.3) with the *local* approach to the analysis of each minimal product. Interactions between features (fourth strategy, Section 5.4) are not taken into account because the implementation is still at an early stage. However, we believe that the results are already relevant since the analysis of the footprint of a minimal product is interesting on its own, and paves the way for future developments. Indeed, to get meaningful results from the analysis of a portion of a product code (the footprint) is not always possible,

Table 3
Experimental evaluation on the case study.

f	$t(P_\mu(f))$	$F(P_\mu(f))$	$M(P_\mu(f))$	$L(P_\mu(f))$	$FP(f)$	$t(a_f)$	a_f
Feature	Time for min. prod.	Features in min. prod.	Methods in min. prod.	Lines in min. prod.	Meths in footprint	Time for annot.	Global annot.
`Client`	1643 ms	11	132	2030	3	1783 ms	0
`Server`	1655 ms	11	132	1977	1	1694 ms	0
`File`	1669 ms	11	140	2053	9	15 675 ms	133
`Journal`	1647 ms	11	139	2069	7	4309 ms	86
`Update`	1659 ms	11	131	1972	1	1686 ms	0
`ClientNr`	1645 ms	11	131	2024	2	1746 ms	0
`Search`	1652 ms	11	132	2030	2	2042 ms	200
`Business`	1641 ms	12	142	2063	2	2035 ms	200
`Data`	1641 ms	13	150	2160	3	2112 ms	133
`Seq`	1633 ms	10	130	1972	21	9952 ms	76
`Concur`	1658 ms	10	133	2025	24	14 874 ms	163
`Site`	1657 ms	10	132	1972	0	1658 ms	0
`Cloud`	1645 ms	10	132	1972	0	1645 ms	0

and depends on the property under study and many other aspects; e.g., when actual execution of the code is needed, as in dynamic analysis, it is not easy to identify how each part of the code affects the whole computation. In any case, our attempt demonstrates, at least, that this way to analyze products has potential and can be applied to a hard task like statically computing upper bounds to resource consumption.

The resource under consideration in the experiments is the *number of instructions*. As already mentioned, in the feature model there are 8 obligatory features and 13 non-obligatory ones; consequently, 13 minimal products have to be generated. In order not to decrease the variability of the product line, no features have been selected as key features. For each non-obligatory feature f, the footprint is computed, and the core part of SACO (i.e., the static analyzer properly said) is called once for each method in the footprint. Experiments have been carried out on a MacBook Pro laptop with a 2.4 GHz Intel Core i5 processor and 4 Gb of memory, running Mac OS 10.7.5. The execution has been repeated 5 times, and reported times (expressed in milliseconds) were computed as the average of all the executions. Table 3 summarizes the experiments: column "f" is the name of the feature under study; "$t(P_\mu(f))$" is the time needed to generate the corresponding minimal product by using existing tools; "$F(P_\mu(f))$", "$M(P_\mu(f))$" and "$L(P_\mu(f))$" are, respectively, the number of features, of methods, and of lines of code (considering only the core ABS code) in the minimal product; "$FP(f)$" is the size (number of methods) of the corresponding footprint; "$t(a_f)$" is the time needed to obtain the global resource–usage annotation of the minimal product; finally, "a_f" is the per-feature resource–usage annotation (according to Section 6.1, it ranges from 0 to 400; the lower, the more efficient).

We can observe that all minimal products are very similar in size, since most code is shared by all of them, and that the time needed to generate them (most of which is taken by the execution of the ABS tools for generating products) is also similar. The most significant difference lies in the size of their footprint: this is consistent with the intuition that the difference between two features is related to the portion of code they directly affect. Note also that some features have no methods in their footprint; this means that, actually, they are "dummy" features which do not modify the code, so that they are given a default resource–usage annotation 0. As regards the efficiency of the analysis process, there is a common pre-processing task which is the same for every feature, and takes around 1350 milliseconds. Column "$t(a_f)$" shows the total time taken by SACO; there is a 10-seconds timeout on each call to the analyzer, which is only reached once when analyzing a method in the footprint of `File`. It must be pointed out that all the minimal products have been analyzed separately, while the implementation could have been optimized by reusing several parts of the computation; for example, most of the work done by SACO on a method can be reused for other methods in the same footprint, and part of the work on a product can be reused for other products. To improve the efficiency following these and other directions is part of future work.

By using the annotations shown in Table 3, the resource–usage–aware configurator suggested 64 possible configurations. The overall resource–usage annotation for each of these configurations is 76, obtained according to the configuration pre-processing described in Listing 10. Out of the possible configurations, if we consider only the configurations that have the minimum number of features constituting a valid configuration, we get the following two configurations:

P1: { `ReplicationSystem`, `Installation`, `Resources`, `JobProcessing`, `ReplicationItem`, `Dir`, `Load`, `Schedule`, `Site`, `Seq` }

and

P2: { `ReplicationSystem`, `Installation`, `Resources`, `JobProcessing`, `ReplicationItem`, `Dir`, `Load`, `Schedule`, `Cloud`, `Seq` }

In the absence of resource–usage annotations, the configurator could suggest another minimal configuration:

```
// in class ReplicationFilePattern; resource usage = 12 instructions
FileEntry getContents() { return internal.getContents(); }

// in class BasicReplicationItemImpl; resource usage = 9 instructions
FileEntry getContents() { return dirContent(snapshot); }

// resource usage = 6 instructions
def FileEntry dirContent(Directory f) = entries(snd(f));
```

Fig. 3. The code of `ReplicationFilePattern.getContents` and the methods and functions called by it.

P3: { `ReplicationSystem`, `Installation`, `Resources`, `JobProcessing`, `ReplicationItem`, `Dir`, `Load`, `Schedule`, `Cloud`, `Concur` }

The overall resource–usage annotation of P3 is 133, which is worse than the overall resource–usage annotations of P1 and P2, despite the fact that all of them are minimal valid configurations. As expected, using the proposed resource–usage annotations, we obtain configurations that have better overall performance.

7.1. Validation

We argue that our experiments, even if still at a very preliminary stage, constitute a proof of concept that resource–usage–aware configuration is feasible. However, it still remains to see how close our resource–usage annotations are to the actual resource consumption of the products. This requires profiling tools (which are currently at the development stage) to be applied to the generated products. Moreover, it requires defining and evaluating heuristics for the different operators we have used in the feature-based strategy.

In the absence of such profiling tools, the results obtained from the static analyzer can be compared with upper bounds computed "by hand". We focus on the feature `File`, whose corresponding footprint consists of 9 methods added or modified by the delta `FileDelta`. The global resource–usage annotation for this feature is 133, and comes from the fact that, according to SACO, 5 methods have constant resource usage (annotation 0), 2 are quadratic (annotation 200), and 2 could not be successfully analyzed (annotation 400). All 5 supposedly constant methods have actually a constant resource usage. For example, the execution of the method `getContents` (shown in Fig. 3 together with the methods and functions it calls[9]) declared in the class `ReplicationFilePattern` (created by `FileDelta`) actually takes 12 steps (instructions), as inferred by SACO: in fact, this method calls `BasicReplicationItemImpl.getContents()`, whose execution takes 9 steps, and the 3 remaining steps are just to set up the call and return the result. The same happens with the call to `dirContent` from `BasicReplicationItemImpl.getContents()`.

Not surprisingly, there are cases where the result inferred by SACO is sub-optimal. For example, the static analyzer is not able to infer an upper bound for the method `compareDirWithPattern` (whose code is shown for clarity in Fig. 4), added by `FileDelta` to the modified class `TesterImpl`. This method simply calls `TesterImpl.compareEntrySets(this,eids,aids,ee,ae)`, for which SACO actually infers a quadratic upper bound `eids*ee + eids*ae +29/2*eids − ee − ae −9/2`. The reason why the upper bound for `compareEntrySets` does not lead to an upper bound for `compareDirWithPattern` is that the input for `compareEntrySets` is computed by applying some functions, and the analyzer cannot find an upper bound for them. Concretely, the function `getFileIdFromDir`, invoked twice to produce the first and second actual parameter of `compareEntrySets` (namely, formal parameters `eids` and `aids`), has polynomial resource usage, but the analyzer is not able to produce this result because of the nature of the recursion used in it. In fact, the recursive calls in `getFileIdFromEntries` are combined by `union`, which is quadratic and whose resource usage depends on both its parameters. Moreover, there is no way to establish the size of `entries(c)` from the size of `fe`. This makes the analysis of this piece of code something which is beyond the capabilities of SACO and, as far as we know, any state-of-the-art static resource–usage analyzer. Section 8 contains a more detailed discussion of the limitations of static analysis when it comes to resource usage and related properties like termination.

In the experiments, we used the *arithmetic mean* on resource–usage annotations for all the methods in the footprint as the final annotation of a feature, and the *sum* of the resource–usage annotations of all features as the resource–usage annotation of a product. Obviously, other choices could have been taken. Future work includes proposing new heuristics that allow having annotations which are closer to the actual resource usage, and undertaking a thorough experimental evaluation.

Finally, it remains to discuss why the configurations P1 and P2 are actually better than P3. The main difference lies in selecting `Seq` instead of `Concur`. It would not be realistic to claim that P1 and P2 are better than P3 because executing them would take less time. This is way beyond the scope of this paper since there are many issues which, ideally, should be addressed before making the claim: e.g.: how the statistical distribution of inputs is (i.e., how well a program performs when all possible inputs are taken into account, each one with its related probability); or: how and whether a smaller

[9] `entries` and `snd` are built-ins whose inferred resource usage is 2.

```
Unit compareDirWithPattern(String pattern,Directory e,Directory a) {
  this.compareEntrySets(
    filters(pattern,getFileIdFromDir(e)),
    filters(pattern,getFileIdFromDir(a)),
    qualifyFileEntry(entries(snd(e)),fst(e)),
    qualifyFileEntry(entries(snd(a)),fst(a))
  );
}
def Set<FileId> getFileIdFromDir(Directory d) =
  case snd(d) {
    Entries(e) =>
      case fst(d) == rootId() {
         True => getFileIdFromEntries1(e);
         False => getFileIdFromEntries(fst(d),e);
      };
  };
def Set<FileId> getFileIdFromEntries1(FileEntry fe) =
  case fe {
    EmptyMap => EmptySet;
    InsertAssoc(Pair(i,c),fs) =>
      case isFile(c) {
         True => Insert(i,getFileIdFromEntries1(fs));
         False => union(getFileIdFromEntries(i,entries(c)),
                  getFileIdFromEntries1(fs));
     };
  };
def Set<FileId> getFileIdFromEntries(FileId id, FileEntry fe) =
  case fe {
    EmptyMap => EmptySet;
    InsertAssoc(Pair(i,c),fs) =>
      case isFile(c) {
         True => Insert(makePath(id,i),getFileIdFromEntries(id,fs));
         False =>
         union(getFileIdFromEntries(makePath(id,i),entries(c)),
            getFileIdFromEntries(id,fs));
       };
  };
```

Fig. 4. The code of `compareDirWithPattern` and related functions.

number of executed instructions affects the "real" execution time (in milliseconds) of a program. Even if the focus were limited to the platform-independent notion of *number of instructions* (i.e., without studying how the real execution time is affected), and the worst case (instead of the average case) were only considered, the limits of static resource–usage analysis would not allow to compute an upper bound for such a complex piece of code as the case study.

However, the local approach used in the experiments is still relevant to the problem of finding the best configuration: in fact, `Seq` was found to be better than `Concur` in terms of its global resource–usage annotation because it contains a smaller number of methods which could be problematic in terms of performance. In particular, SACO is able to infer a constant or linear upper bound for 20 out of 21 methods in the footprint corresponding to `Seq`, whereas there are 5 methods in the footprint of *Concur* for which an upper bound could be obtained. Although this is not a guarantee that the performance of P1 and P2 will be better than P3, at least it indicates that executing P3 is more likely to fall into the execution of non-terminating or very expensive methods. We believe that (1) this is consistent with the usual philosophy underlying static analysis, where the generality of the results is at least as important as the precision on specific cases; and (2) it paves the way for improvements which could be obtained thanks to advances in the static resource–usage techniques and tools.

8. Threats to validity

This section discusses threats to the validity of our approach. We first revise the internal components of our approach which can compromise the precision of the method. Then, we discuss the generalization of our approach to be used in combination with other static and dynamic analyzers.

8.1. Internal validity

As pointed out before, resource–usage–aware configuration is a problem of optimization whose goal is to select the product which is optimal in terms of resource usage. It is clear that optimality cannot be obtained in general because the

exact resource consumption is not computable. Still, we want to discuss the internal sources of imprecision that are due to the methods we use to select the best product, namely, the loss of precision due to resource–usage analysis.

There is an inevitable loss of precision due to the approximations that must be performed in order to obtain an upper bound on resource usage. Such source of precision loss have been described above (in Section 4 and elsewhere). Here, we simply list the rest of such approximations, and point to related work where technical descriptions can be found:

1. *Non-linear ranking functions* [43]: most existing resource–usage analyzers can bound the number of iterations of loops when there is a *linear ranking function* that approximates such bounds; however, when this is not the case, one needs to use more advanced techniques which are frequently not incorporated into state-of-the-art analyzers. As an example, SACO can only find linear ranking functions.
2. *Field-sensitive analysis* [8]: there might be also a big loss of precision when the resource consumption of a fragment of code depends on the size of data that are *not local* to the methods, e.g., it is stored in class fields. This is a challenging issue because one needs to be sure that these data are not accessed (and modified) outside the fragment of code under study (in transitive calls). If certain condition about *aliasing* between variables are met, then such global data can be converted into local data, and the resource–usage analysis can obtain bounds that depend on it. Unfortunately, this is not always the case, and the result is the need for approximations which unavoidably lose precision.
3. *Size-abstractions* [20]: When the resource usage depends on the size of data allocated in complex data structures (as in the case of a loop traversing a tree), it is necessary to use *size abstractions* which accurately capture how the size of the data structure decreases in the computation. For example, *path-length* [52] is a practical measure used in object-oriented programming. However, there is an inevitable loss of precision due to its use (since it only says that the path-length of the data structure decreases, but it does not say by how much it decreases).
4. *Concurrency* [4]: For *thread-level-concurrency*, we are not aware of any resource analyzer that can handle thread interleavings. For *actor-based concurrency*, there have been recent proposals [10,4] to leverage the methods used in the analysis of sequential code to the concurrent paradigm. The loss of precision in the analysis of concurrent programs occurs when tasks interleave, and we have to assume that global data might have been modified at these interleaving points. SACO works for actor-based concurrency, and is able to give accurate upper bounds for a wide class of programs.

8.2. External validity

Generalization to other static analyzers. We have illustrated the four strategies proposed in the article using the SACO resource–usage analysis tool. However, our method can be used in combination with any other resource–usage analyzer. For instance, in principle, CoFloCo [25], SPEED [27] or Loopus [49] could be used as well. On the other hand, the advantages and disadvantages of each strategy entirely depend on the property that we are considering (in our case, resource consumption). Other properties that can be inferred by static analyzers might lead to different assessment of the strategies. For instance, if one simply wants to measure the number of lines of each product, then the first strategy would be the ideal one, because even if the number of products can be huge, the number of lines can be counted easily and efficiently. Therefore, we do not claim that resource–aware configuration is an idea which can be easily transposed to other properties than resource usage.

Dynamic analyzers. In contrast to other approaches [45], our discussion of the four strategies is based on the fact that the property is obtained by rigorous *static analysis* and by code inspection only, i.e., programs under study are not actually run and *all possible inputs* are taken into account. Otherwise, the second strategy would not make sense as one cannot run in general a partial application. However, in principle, it is possible to analyze a partially-built program (though the state of implementation of resource analyzers is behind this). Also, the evaluation of a feature-based strategy would be not always possible, since some features may need a context in order to execute, i.e., the application must be entirely built and executed from a `main`, while static analyzers can in general analyze the code of features separately from their execution context.

9. Related work

Very few authors have addressed the problem of obtaining quality indicators for features and/or product configurations that can be used to guide the product configuration process.

In the classification and survey of analysis strategies for software product lines presented by Thum et al. [54], analysis strategies applied to software product lines are classified in *Product-Based (unoptimized and optimized) Analyses*, *Family-Based Analyses*, *Feature-Based Analyses*, and combined analysis strategies. The authors focus on analyses that operate statically, and the types of software analysis taken into account are *Type Checking*, *Static Analysis*, *Model Checking*, and *Theorem Proving*. None of the static analysis strategies referred in that survey – thirteen strategies mostly published during the last four years – aim at supporting resource–usage–aware configuration.

Soares at al. [50] present a systematic review of the analysis of *non-functional properties* in software product lines. They focus on execution/runtime non-functional properties, visible and measurable at source code or during the product execution, such as reliability and performance. 36 primary studies are classified in *Quality Prediction*, *Quality Estimation*, and *Feature Selection*. In the context of our work, the category of interest is *Quality Estimation*, despite some work reported in

Quality Prediction being of interest too. Performance appeared as one of the most commonly addressed runtime properties in this survey, which helped us to check the completeness of the related work discussed in this section.

Kolesnikov et al. [37] propose the use of *quality predictors* (e.g., a predictor for high memory consumption) based on measures of internal product attributes to guide a sampling process that determines the products that fall into the category denoted by the given predictor (e.g., products with high memory consumption). In their approach, predictors are established from the relationship between internal and external product attributes and a small training set of products; a sampling framework based on cooperative game theory generates a collection of feature sets that belongs to the quality category of interest; and products containing one or more of these feature sets will belong to the same quality category. The general idea is the same as ours: to only use statically available information from the feature model and the source code. However, our approach is more suitable for resource–usage aware product configuration, because it finds out the configurations including the desired key features and optimizing resource consumption, instead of finding out products that fall into a certain category, but might not have any relation to the desired product in terms of key features. On the other hand, their approach can be used for different quality predictors, whereas the approach described in the present paper is specific to resource usage.

Similarly to our approach, Sincero et al. [48] address the problem of finding out product configurations that include the desired key features and optimize non-functional properties. In their approach, the concept of *Partial Configuration* is used to enable the user to select features in a feature model that must be present in the product configuration (corresponding to our key features), and to mark features as *open* (might or might not be present) and *blocked* (will not be present). The approach relies on a testing infrastructure in which products are generated and tested, incrementally feeding a data base of non-functional properties for valid product configurations. Processing mechanisms are in charge of reasoning about the influence of each feature or combination of features on the quantification of a certain non-functional property. In their case study, they tested all valid product configurations derived from the partial configuration and used analysis of covariance as processing mechanism. In this approach, the number of valid product configurations derived from the partial configuration and, thus, to be tested might still be high. The authors mention that an approximation of the response for not-tested configurations can be calculated.

Siegmund et al. [46,45] present an approach for estimating non-functional properties of products in an SPL by aggregating the non-functional properties of selected features. Based on the feature documentation, a small but suitable set of products are compiled and measured, and the values of non-functional properties per feature are approximated from deltas between two products that differ only in the presence or absence of this feature. This approach takes into consideration feature interaction, by having a model that defines known feature interactions and measures their influence. The influence of a feature interaction is estimated by adding a single product that contains the interacting features to the set of products, and by computing the delta between non-functional properties of the actually measured product and predicted non-functional properties of the same product. The authors show that for a product line with n features, already $n + 1$ measurements can lead to acceptable predictions of footprint.[10] With regard to measurement of feature interactions, the initial approach [46] took into consideration the mapping between features and implementation units, source code, and domain expert knowledge to identify more complex feature interactions. An alternative in case of lack of domain knowledge was to simply assume the existence of a feature interaction between each pair of features (*pair-wise measurement*), which substantially increases the number of products to be measured. The approach described in [45] is an evolution of the one described in [46], in which the authors reduce the effort for pair-wise measurement and propose three heuristics for detecting the relevant performance feature interactions: (1) pair-wise (or first-order) interactions are the most common form of performance-relevant feature interactions; (2) second-order feature interactions can be predicted by analyzing already detected pair-wise interactions; and (3) there are few features (called *hot-spot* features) that interact with many features. Their general approach is implemented in a tool called SPL Conqueror [47]; however, performance is treated as a variant-wise quantifiable property in [47], which has several implications: (1) it is not used to support the selection of product configurations that best fits the user requirements, but used in a second stage for defining which product configuration is optimal taking performance into consideration; (2) the product configurations must be generated and measured; (3) as a result of the consequent need to reduce the number of product configurations to be generated and measured, features that have a negative effect on a property that is of interest to a customer are from the beginning excluded from further considerations. In our approach, performance is treated as a feature-wise quantifiable property, so that it can be considered in the objective function in a standard way, which allows supporting the selection of feature configurations with the definition of constraints, if desired, as well as optimization taking into consideration other non-functional properties instead of excluding features from the beginning from further consideration.

In summary, similarly to our feature-based analysis, these related papers [48,46,45] propose an approach to predict non-functional properties by aggregating the influence of each selected feature on a non-functional property. A fundamental difference is that the estimate of the performance of the features is based on formal methods (i.e., static resource–usage analysis) in our approach, while their works perform measurements dynamically. They generate and measure a small set of products and, by comparing measurements, they approximate the influence of each feature on the non-functional property in question. The differences between static and dynamic analysis are well-known: while measurements consist in executing

[10] Here, *footprint* means the size of the compiled program, and is completely different from our notion of *feature footprint*, introduced in Section 4.2.

the application and monitoring the measure of interest during runtime, static analysis infers the properties by examining the code only and without executing the program. The two approaches are complementary: rigorous resource–usage guarantees (upper bounds) can only be found by static analysis; however, due to loss of precision in the analysis, the guarantees can be too pessimistic, and measurement-based techniques can give more accurate estimates. Our analysis of the four strategies is based on the fact that the property is obtained through rigorous analysis by inspecting the code only. Otherwise, the second strategy would not make sense as one cannot run a partial application. Also, the evaluation of a feature-based strategy would be not always possible, since some features may need a context in order to execute (i.e., the application must be run from a `main`, and it is not possible to evaluate the feature for different input values).

It is well-known that the use of formal methods has some advantages. In our case, we rely on a static analysis which infers approximations that are safe for any possible input data value. In addition, an important consequence of this choice of static analysis is that we can analyze partial products or focus on the performance behavior of fragments of the product (e.g., the footprint), while they need to analyze performance globally, as they perform measurements. This gives us flexibility.

Finally, there are authors who propose the use of domain expert judgment to assign qualitative or quantitative values to the interdependency between functional features and quality attributes (e.g. [51,59]). The approach by Zhang et al. [59] uses *Analytical Hierarchy Process* (AHP), a well-known pair-wise comparison method used to calculate the relative ranking of different opinions, as underlying technology, whereas Soltani et al. [51] propose the use of *Stratified Analytical Hierarchy Process* (S-AHP), because it significantly reduces the number of needed pairwise comparisons. Both approaches depend on the availability of domain experts, who must engage themselves in a time-consuming and error-prone activity.

10. Conclusion and future work

This article introduces a notion of *resource–usage–aware configuration* based on static analysis, which strives for finding a selection of features with good behavior from the point of view of resource usage, and complying with the quality constraints provided by the user. We have envisaged several strategies for resource–usage–aware configuration, and described a prototype implementation of the most practical strategy. Our implementation shows that it is feasible to use an off-the-self static analyzer to obtain resource–usage indicators that can be used to annotate feature models. Using the annotated feature model, the configurator is able to suggest a small set of valid product configurations that best fit the objective function representing the user input.

The main difference with respect to related work is the use of static analysis. Most approaches in the literature execute the generated products for some specific workload, while the present approach aims at obtaining upper bounds to the resource usage, which is a different (and much more difficult) problem. To transform the resulting upper bounds into a useful piece of information for the process of configuration is also a non-trivial task. Moreover, the way products are analyzed in the feature-based strategy is also new and interesting.

Our implementation and its application to case studies constitutes a proof of concept for resource–usage–aware configuration. However, a thorough experimental evaluation is required to assess the accuracy of the envisaged strategies and, in particular, to define appropriate heuristics that lead to efficient products. In future work, we plan to define and evaluate different heuristics to combine the contribution of each method to the resource consumption of the feature, and also more refined heuristics to map resource–usage upper bounds into annotations. Also, we currently do not have tools to profile the generated products and see the actual resource consumption for a wide range of input data. This is also subject of ongoing work.

As a final remark, it must be pointed out that scalability is hard to obtain in this framework because of the limitations of static analysis when a hard-to-compute property like resource usage is studied. This is a drawback of static analysis more than our work, and our proposals would certainly benefit from any advance in that research field.

Acknowledgements

This work was partially funded by the European Union projects FP7-ICT-231620 (HATS: Highly Adaptable and Trustworthy Software Using Formal Models – http://www.hats-project.eu/) and FP7-ICT-610582 (ENVISAGE: Engineering Virtualized Services – http://www.envisage-project.eu), by the Spanish project TIN2012-38137, and by the CM (Comunidad de Madrid) project S2013/ICE-3006.

References

[1] E. Abbasi, A. Hubaux, P. Heymans, A toolset for feature-based configuration workflows, in: Proceedings of SPLC, 2011, pp. 65–69.
[2] G. Agha, C.J. Callsen, Actorspace: an open distributed programming paradigm, in: Proceedings of PPoPP, 1993, pp. 23–32.
[3] E. Albert, D. Alonso, P. Arenas, S. Genaim, G. Puebla, Asymptotic resource usage bounds, in: Proceedings of APLAS, in: Lect. Notes Comput. Sci., vol. 5904, Springer-Verlag, 2009, pp. 294–310.
[4] E. Albert, P. Arenas, J. Correas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, G. Román-Díez, Object-sensitive cost analysis for concurrent objects, Software Testing, Verification and Reliability 25 (3) (2015) 218–271.
[5] E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martín-Martín, G. Puebla, G. Román-Díez, SACO: static analyzer for concurrent objects, in: Proceedings of TACAS, in: Lect. Notes Comput. Sci., vol. 8413, Springer-Verlag, 2014, pp. 562–567.
[6] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, COSTABS: a cost and termination analyzer for ABS, in: Proceedings of PEPM, ACM Press, 2012, pp. 151–154.

[7] E. Albert, P. Arenas, S. Genaim, I. Herraiz, G. Puebla, Comparing cost functions in resource analysis, in: Proceedings of FOPARA: Revised Selected Papers, in: Lect. Notes Comput. Sci., vol. 6324, Springer-Verlag, 2009, pp. 1–17.
[8] E. Albert, P. Arenas, S. Genaim, G. Puebla, G. Román-Díez, Conditional termination of loops over heap-allocated data, Sci. Comput. Program. 92 (2014) 2–24.
[9] E. Albert, J. Correas, G. Puebla, G. Román-Díez, Incremental resource usage analysis, in: Proceedings of PEPM, ACM Press, 2012, pp. 25–34.
[10] E. Albert, A. Flores-Montoya, S. Genaim, E. Martin-Martin, Termination and cost analysis of loops with concurrent interleavings, in: Proceedings of ATVA, in: Lect. Notes Comput. Sci., vol. 8172, Springer-Verlag, 2013, pp. 349–364.
[11] D. Batory, J. Sarvela, A. Rauschmayer, Scaling step-wise refinement, IEEE Trans. Softw. Eng. 30 (6) (2004) 355–371.
[12] D. Benavides, P. Trinidad, A. Ruiz-Cortez, Automated reasoning on feature models, in: Proceedings of CAiSE, Springer-Verlag, 2005, pp. 491–503.
[13] F.S.D. Boer, R. Hähnle, E.B. Johnsen, R. Schlatte, P.Y.H. Wong, Formal modeling of resource management for cloud architectures: an industrial case study, in: Proceedings of ESOCC, in: Lect. Notes Comput. Sci., vol. 7592, Springer-Verlag, 2012, pp. 91–106.
[14] J. Bosch, S. Deelstra, M. Sinnema, COVAMOF in Systems and Software Variability Management – Concepts, Tools and Experiences, Springer-Verlag, 2013, pp. 141–150.
[15] D. Clarke, R. Muschevici, J. Proença, I. Schaefer, R. Schlatte, Variability modelling in the ABS language, in: Proceedings of FMCO 2010, in: Lect. Notes Comput. Sci., vol. 6957, Springer-Verlag, 2011, pp. 204–224.
[16] A. Classen, Q. Boucher, P. Heymans, A text-based approach to feature modelling: syntax and semantics of TVL, Sci. Comput. Program. (2010) 1130–1143.
[17] M. Codish, S. Debray, R. Giacobazzi, Compositional analysis of modular logic programs, in: Proceedings of POPL, ACM Press, 1993, pp. 451–464.
[18] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proceedings of POPL, ACM Press, 1977, pp. 238–252.
[19] P. Cousot, R. Cousot, Modular static program analysis, in: Proceedings of CC, in: Lect. Notes Comput. Sci., vol. 2304, Springer-Verlag, 2002, pp. 159–178.
[20] P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among variables of a program, in: Proceedings of POPL, ACM Press, 1978, pp. 84–96.
[21] K. Czarnecki, U. Eisenecker, Generative Programming, Addison–Wesley Longman Publishing Co., Inc., 2000.
[22] K. Czarnecki, S. Helsen, U.W. Eisenecker, Staged configuration through specialization and multi-level configuration of feature models, Softw. Process Improv. Pract. 10 (2) (2005) 143–169.
[23] K. Czarnecki, S. She, A. Wasowski, Sample spaces and feature models: there and back again, in: Proceedings of SPLC, 2008, pp. 22–31.
[24] L. Etxeberria, G.S. Mendieta, L. Belategi, Modelling variation in quality attributes, in: Proceedings of VaMoS, 2007, pp. 51–59.
[25] A. Flores-Montoya, R. Hähnle, Resource analysis of complex programs with cost equations, in: Proceedings of APLAS, in: Lect. Notes Comput. Sci., vol. 8858, Springer-Verlag, 2014, pp. 275–295.
[26] B. González-Baixauli, J.C.S. do Prado Leite, J. Mylopoulos, Visual variability analysis for goal models, in: Proceedings of RE, IEEE, 2004, pp. 198–207.
[27] S. Gulwani, K.K. Mehra, T.M. Chilimbi, SPEED: precise and efficient static estimation of program computational complexity, in: Proceedings of POPL, ACM Press, 2009, pp. 127–139.
[28] J. Guo, J. White, G. Wang, J. Li, Y. Wang, A genetic algorithm for optimized feature selection with resource constraints in software product lines, J. Syst. Softw. 84 (12) (2011) 2208–2221.
[29] M. Hermenegildo, G. Puebla, K. Marriott, P. Stuckey, Incremental analysis of constraint logic programs, ACM Trans. Program. Lang. Syst. 22 (2) (2000) 187–223.
[30] J. Hoffmann, M. Hofmann, Amortized resource analysis with polynomial potential, in: Proceedings of ESOP, in: Lect. Notes Comput. Sci., vol. 6012, Springer-Verlag, 2010, pp. 287–306.
[31] A. Hubaux, A. Classen, P. Heymans, Formal modelling of feature configuration workflows, in: Proceedings of SPLC, 2009, pp. 221–230.
[32] A. Hubaux, P. Heymans, P.-Y. Schobbens, D. Deridder, E. Abbasi, Supporting multiple perspectives in feature-based configuration, Softw. Syst. Model. (2011) 1–23.
[33] S. Jarzabek, B. Yang, S. Yoeun, Addressing quality attributes in domain analysis for product lines, IEEE Softw. 153 (2) (2006) 61–73.
[34] E.B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, M. Steffen, ABS: a core language for abstract behavioral specification, in: Proceedings of FMCO, in: Lect. Notes Comput. Sci., vol. 6957, Springer-Verlag, 2012, pp. 142–164.
[35] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, A.S. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Technical Report CMU/SEI-90-TR-21, SEI, 1990.
[36] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of ECOOP, in: Lect. Notes Comput. Sci., vol. 1241, 1997, pp. 220–242.
[37] S.S. Kolesnikov, S. Apel, N. Siegmund, Predicting quality attributes of software product lines using software and network measures and sampling, in: Proceedings of VaMoS, 2013, pp. 1/5–5/5.
[38] J. Kuusela, J. Savolainen, Requirements engineering for product families, in: Proceedings of ICSE, ACM Press, 2000, pp. 61–69.
[39] J. Lee, K.C. Kang, A feature-oriented approach to developing dynamically reconfigurable products in product line engineering, in: Proceedings of SPLC, 2006, pp. 131–140.
[40] A. Murashkin, M. Antkiewicz, D. Rayside, K. Czarnecki, Visualization and exploration of optimal variants in product line engineering, in: Proceedings of SPLC, ACM, 2013, pp. 111–115.
[41] K. Pohl, G. Böckle, F. Van Der Linden, Software Product Line Engineering: Foundations, Principles, and Techniques, Springer-Verlag, 2005.
[42] M. Riebisch, K. Böllert, D. Streitferdt, I. Philippow, Extending feature diagrams with UML multiplicities, in: Proceedings of IDPT, 2002, pp. 23–27.
[43] S. Sankaranarayanan, H. Sipma, Z. Manna, Non-linear loop invariant generation using Gröbner bases, in: Proceedings of POPL, ACM Press, 2004, pp. 318–329.
[44] I. Schaefer, L. Bettini, V. Bono, F. Damiani, N. Tanzarella, Delta-oriented programming of software product lines, in: Proceedings of SPLC, 2010, pp. 77–91.
[45] N. Siegmund, S.S. Kolesnikov, C. Kästner, S. Apel, D.S. Batory, M. Rosenmüller, G. Saake, Predicting performance via automated feature-interaction detection, in: Proceedings of ICSE, 2012, pp. 167–177.
[46] N. Siegmund, M. Rosenmüller, C. Kästner, P.G. Giarrusso, S. Apel, S.S. Kolesnikov, Scalable prediction of non-functional properties in software product lines, in: Proceedings of SPLC, 2011, pp. 160–169.
[47] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, G. Saake, SPL conqueror: toward optimization of nton-functional properties in software product lines, Softw. Qual. J. 20 (3–4) (2012) 487–517.
[48] J. Sincero, W. Schröder-Preikschat, O. Spinczyk, Approaching non-functional properties of software product lines: learning from products, in: Proceedings of APSEC, IEEE Computer Society, 2010, pp. 147–155.
[49] M. Sinn, F. Zuleger, H. Veith, A simple and scalable static analysis for bound analysis and amortized complexity analysis, in: Proceedings of CAV, in: Lect. Notes Comput. Sci., vol. 8559, Springer-Verlag, 2014, pp. 745–761.
[50] L. Soares, P. Potema, I. Machado, I. Crnkovic, E. Almeida, Analysis of non-functional properties in software product lines: a systematic review, in: Proceedings Euromicro DSD/SEAA, 2014, pp. 328–335.
[51] S. Soltani, M. Asadi, M. Hatala, D. Gasevic, E. Bagheri, Automated planning for feature model configuration based on Stakeholders' business concerns, in: Proceedings of ASE, IEEE, 2011, pp. 536–539.
[52] F. Spoto, F. Mesnard, É. Payet, A termination analyzer for java bytecode based on path-length, ACM Trans. Program. Lang. Syst. 32 (3) (2010).

[53] T. Than Tun, Q. Boucher, A. Classen, A. Hubaux, P. Heymans, Relating requirements and feature configurations: a systematic approach, in: Proceedings of SPLC, 2009, pp. 201–210.
[54] T. Thum, S. Apel, C. Kastner, I. Schaefer, G. Saake, A classification and survey of analysis strategies for software product lines, ACM Comput. Surv. 47 (1) (2014) 6:1–6:45.
[55] K. Villela, T. Arif, D. Zanardini, Towards product configuration taking into account quality concerns, in: Workshop on Formal Methods and Analysis in Software Product Line Engineering (FMSPLE), in: Proc. SPLC, vol. 2, ACM Press, 2012, pp. 82–90.
[56] J. White, B. Dougherty, D. Schmidt, Selecting highly optimal architectural feature sets with filtered Cartesian flattening, J. Syst. Softw. 82 (8) (2009) 1268–1284.
[57] J. White, B. Dougherty, D. Schmidt, D. Benavides, Automated reasoning for multi-step feature model configuration problems, in: Proceedings of SPLC, 2009, pp. 11–20.
[58] J. White, J.A. Galindo, T. Saxena, B. Dougherty, D. Benavides, D.C. Schmidt, Evolving feature model configurations in software product lines, J. Syst. Softw. 87 (2014) 119–136.
[59] G. Zhang, H. Ye, Y. Lin, Quality attribute modeling and quality aware product configuration in software product lines, Softw. Qual. J. 22 (3) (2014) 365–401.
[60] H. Zhang, S. Jarzabek, B. Yang, Quality prediction and assessment for product lines, in: Proceedings of CAiSE, Springer-Verlag, 2003, pp. 681–695.