Resource-Aware Configuration in Software Product Lines

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

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

Abstract

Deriving concrete products from a product-line infrastructure requires resolving the variability captured in the product-line, based on a company's 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 should be considered. This article envisages several scenarios for *resource-aware configuration* which feature different performance and efficiency trade-offs. Resource-aware configuration aims at providing awareness of the performance quality throughout the configuration process. The common idea in all scenarios is the use of resource estimates obtained by an off-the-shelf resource analyzer as a heuristic for choosing among different candidate configurations. We report on a prototype implementation of the most practical scenario for resource-aware configuration and apply it on an industrial case study.

Keywords: Software Product Line Engineering, Product Configuration, Resource Analysis, Quality Metrics

1. Introduction

One increasing trend in the software engineering market is the need to develop multiple, similar software products instead of just a single individual product. *Software Product Line Engineering* (SPLE) [22] offers a solution 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 products.

Deriving concrete products from a product-line infrastructure requires resolving the variability captured in the product-line according to a company's

[☆] Corresponding author: damiano@fi.upm.es
Email addresses: elvira@sip.ucm.es (Elvira Albert), taslim.trif@iese.fraunhofer.de (Taslim Arif), karina.villela@iese.fraunhofer.de (Karina Villela), damiano@fi.upm.es (Damiano Zanardini)

market strategy or the requirements from specific customers. *Feature models* [19, 9] have been the main approach for capturing the commonality and variability in product-line. The configuration process usually consists in selecting those features that are applicable to the product, so that this product can be assembled from the product-line assets. Then, one of the most difficult tasks is the translation of market or customer requirements and goals into a 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 constraints. 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.

This article is concerned with the configuration problem from the perspective of product *performance*, a.k.a. product *resource consumption*, a frequently desired quality for software products. *Resource consumption* refers to a range of performance metrics and our whole approach is generic w.r.t. the metric used. In our implementation and case study, the quality metrics we use to estimate the degree of performance of a product are the amount of allocated memory and the number of executed instructions. We present three scenarios for *resource-aware configuration* of software product-lines. The common idea in all scenarios is the use of resource-consumption estimates as a *heuristic* for guiding the selection of features. The crux is the use of an automated resource analyzer (e.g., [14, 15, 3]) that provides estimates of the resource consumption and helps to select features in a (partially-built) product. The estimates are used to guide the configuration process towards more efficient products, while keeping intact all the *key features* of the product (i.e., the essential features from the user′s point of view), and also adhering to economic cost constraints and to the dependencies and constraints specified in the feature model.

Soltani et al. [29] and Siegmund et al. [25] have already identified performance as a business concern in their configuration approach. However, as far as we know, there are no other approaches to resource-aware configuration that use automated resource analysis to assist the configuration process.

1.1. Summary of Contributions

Our main contribution is on the notion of resource-aware configuration that relies on the rigorous formal technique of automated resource analysis to assist the configuration process. This overall contribution breaks down in a series of scenarios that realize such notion of resource-aware configuration, in a prototypical implementation of the most practical scenario, and in a preliminary experimental evaluation. The main contributions are summarized as follows:

- We propose a scenario for *Product-Level Analysis*, in which we estimate the resource consumption of (selected) products after product configu-

ration, i.e., resource analysis is run *a posteriori* for (selected) product configurations.

- We then discuss a scenario for *Partial-Product-Level Analysis*, in which the resource analyzer is invoked to analyze *partial products* obtained after the selection of features along the configuration process. Thus, there is an interwoven interaction between the configurator and the analyzer.

- The most practical scenario is *Feature-Level Analysis*, where the resource analyzer is run *a priori* to estimate the impact that each feature may have on the resource consumption of products. This is achieved by starting from a *minimal product* that includes the minimal number of features to get a valid configuration, and then analyzing the features under study one by one.

- We report on a prototype implementation of a feature-level analysis which uses the COSTABS resource analyzer [3] to infer resource estimates and annotates the features with performance levels which are then used by the configurator to suggest a valid product configuration that best fits the quality constraints provided by the user's input. The whole resource-aware configuration process is fully automatic.

- We have applied our implementation on an industrial case study that provides search and merchandising services. While product-level analysis of our case study requires the analysis of 768 products to obtain performance annotations, we will see that feature-level analysis only needs to generate and analyze 13 products. Our experiments show that it is feasible to infer performance annotations for all optional features in an efficient way. The annotations are then used by the product configurator to configure a product that meets the user's performance constraints.

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 are necessary later to present our resource-aware scenarios. In particular, we define a function *Configurator* which is instrumental later.

Section 4 overviews the main notions of static resource analysis. The analysis is viewed as a black box and we focus on describing the different parameters that the analysis often has, as well as the output of the analysis process. This background knowledge will then be useful to understand the scenarios.

Section 5 introduces the three scenarios for resource-aware configuration described in the previous section and points out the advantages and disadvantages of each of them. We will see that the different scenarios present a number of trade-offs: namely we have that product-level analysis is sound but inefficient, partial-level analysis is unfeasible using our current technology and feature-level analysis is practical but less precise.

Section 6 reports on a prototype implementation of the feature-level analysis scenario and applies it on an industrial case study. Finally, Section 7 reviews related work and Section 8 concludes.

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 [22]. 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 languages of the HATS[1] framework [7]. 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 above HATS project website).

2.2. Feature Models

A *feature model* [19, 9] represents a hierarchy of features, which are properties of domain concepts relevant to some domain stakeholder and used to discriminate between concept instances. Table 1 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

[1] HATS project website: `http://www.hats-project.eu/`

"XOR" (alternative) relation can hold between features in the same group [23]. In an "OR" group, it is also possible to set a minimum and maximum number (n_1 and n_2 in Table 1) of features that have to be present in any product. Moreover, a feature can be *mandatory*, if it is common to all possible instances of the concept, or *non-mandatory*, if it is marked as optional (**opt** in Table 1) or belongs to an alternative ("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.

Table 1: Main μTVL Constructs

General Construct	**μTVL Construct**
Mandatory feature	no **opt** before feature identifier
Optional feature	**opt** before feature identifier
AND relation	**group allof**
OR relation	**group** $[n_1..*]$, **group** $[n_1..n_2]$
XOR relation	**group oneof**
Cross-tree relations	**require**, **exclude**, **ifout** and logical operators !, ||, &&, $\rightarrow$ and $\leftrightarrow$

The *Micro Textual Variability Language* (μTVL) [7] is a text-based feature modeling language that extends a subset of TVL [8]. Table 1 shows its main constructs. A feature model is represented textually as a tree of nested features, each with a collection of boolean or integer attributes. Additional cross-tree relations can also be expressed.

```
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 our case study. 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 code level how each feature contributes to the behavior of the final product. Several approaches have been used to this end, such as *aspect-oriented* [20], *feature-oriented* [6], and *delta-oriented* programming [24].

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 to apply is based on the selection of desired features for the final product. The *Delta Modelling Language* (DML) [7] is used to define deltas, and provides constructs for modifying code, such as **adds**, **removes** or **modifies**, which can refer either to classes, interfaces, or methods. 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 [18], a language which has been recently developed to program distributed concurrent systems. The sequential part of the language is similar to Java, but it also includes a function sublanguage to define data types.

```
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

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) [7] 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 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). 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.

```
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

2.5. Product Specification

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 attributes of features which need it. This provides traceability and supports automatic derivation of products from the product-line infrastructure. Listing 4 shows two products from the replication system product-line using the *Product Selection Language* (PSL) [7].

```
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

2.6. Product Generation

Given a feature model FM, a core model CM, a set of deltas D, a product-line configuration LC, 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 in FM; (2) use LC 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 the product configuration process, but just to fix the basic notions needed to accurately describe the scenarios 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.

A set of *key features* $K \equiv k_1, \ldots, k_s$ might be optionally provided. Key features are those features whose selection the user of the configurator is confident about; therefore, they should be part of all solutions provided by the configurator, as long as they respect the feature model. The solution provided by the configurator is a set of candidate configurations $Conf \equiv C_1 \ldots C_n$, where each C_i is defined as a set of features $\{f_1, \ldots, f_m\}$ (optionally providing initial values for attributes). All configurations include the set of mandatory features and the set of key features $k_1, \ldots, k_s$ selected by the user, and must be valid w.r.t. the feature model.

Example 3.1. *For instance two candidate configurations for our case study are those in Listing 4 which, in the above notation, are:* $C_1 = \{$`ReplicationSystem`, `Installation`, `Resources`, `JobProcessing`, `ReplicationItem`, `Dir`, `Load`, `Schedule`, `Site`, `Seq`$\}$ *and* $C_2 = \{$`ReplicationSystem`, `Installation`, `Resources`, `JobProcessing`, `ReplicationItem`, `Dir`, `Load`, `Schedule`, `Site`, `Seq`, `File`, `Journal`, `ClientNr{c=2,j=5}`, `Update{u=3}`, `Search{d=10,l=20}`, `Business{d=10,l=20}`$\}$. *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 for instance the configuration* C_1 *above, we generate a product from it by following the CL specification in Listing 3 and applying the deltas in Listing 2 to the core module. The result is a program written in the ABS language (same language as in the deltas in Listing 2).*

3.2. Configuration Trees

In order to define the resource-aware configuration scenarios, it is useful to view the configuration process as the construction of a *decision tree*, referred to as the *configuration tree*, whose nodes are *partial configurations* $C \equiv \{f_1, \ldots, f_n\}$, i.e., the set of features selected so far while traversing the tree. A branch $C \rightsquigarrow C'$ in the tree adds one or more features to C such that the child C' is $C \cup \{f_{n+1}, \ldots, f_m\}$. The reason why several features might be added in one step are the constraints in the feature model; e.g., it might be the case that one cannot select f_{n+1} without selecting $f_{n+2}, \ldots, f_m$. 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.

Example 3.3. *For instance, the configuration tree for the case study has a branch* `ReplicationSystem` $\rightsquigarrow$ `Installation` $\rightsquigarrow$ `Resources`$\rightsquigarrow$ `JobProcessing`$\rightsquigarrow$ `ReplicationItem`$\rightsquigarrow$`Dir`$\rightsquigarrow$ `Load`$\rightsquigarrow$ `Schedule`$\rightsquigarrow$ `Site`*. We assume that each node in the branch contains the new feature indicated in the node plus all features in the previous nodes of the same branch. From the last node, we will have as children configurations C_1 and C_2 of Example 3.1. For C_1, we add one further step* $\rightsquigarrow$ `Seq` *and the resulting configuration becomes a leave of the tree. For C_2, we have to add all remaining features.*

Given the product-line infrastructure PL and the set of key features K, we rely on a generic configurator invoked as $\mathit{Configurator}(PL, K)$ that computes a decision tree τ as described above. In the following, *Conf* denotes the leaves of the tree (i.e., the set of valid configurations), and *PartConf* denotes the intermediate nodes (i.e., the set of partial configurations).

Example 3.4. *The configuration tree for our case study has 768 leaves, so one could generate 768 candidate products. Two of those leaves are C_1 and C_2.*

4. Static Resource Analysis

In our *resource-aware configuration* approach, resource-consumption estimates are computed by an off-the-shelf resource analyzer and used to select the most promising configuration candidate(s). We rely on the *generic* resource-analysis tool *Analyzer*, which, given a fragment of code P, an entry method m_0, and a resource metric of interest R, is invoked as $\mathit{Analyzer}(P, m_0, R)$ and analyzes the resource consumption of m_0, as well as of those n methods transitively invoked from it, w.r.t. R. As a result, it returns a set of $n + 1$ pairs (m_i, u_i), where m_i is a method name and u_i is an *upper bound* to its resource consumption. The upper bound is a *sound* worst-case approximation of the actual cost such that it is ensured that none execution of method m_i (for any input data) can exceed the inferred bound u_i.

Example 4.1. *Consider a fragment of method* `Unit transferItems(Set<File> fileset)` *showed in Listing 5 which is part of our case study. This method has been pointed out in [12] as a hot spot in the execution of the case study. This method traverses the set of files that receives as input parameter and on each element of the set, it performs a number of processing operations. 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 which are completely outside the focus of this work). The important point to notice is the external* **while** *loop which traverses the set of files (parameter* `fileset`*) using an iterator and, at each loop iteration, it invokes some auxiliary operations that will add their resource consumption.*

Let us analyze its resource consumption using the COSTABS tool [3], an implementation of a resource analyzer for ABS programs. We select the cost model that counts number of steps, since this is the metric which is most related to execution time. COSTABS returns the following asymptotic *upper bound: size(fileset)*size(rdir)^2 + size(fileset)^3*size(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 { .....
        // ommitted a fragment of the method
}
```

Listing 5: Excerpt of method transferItems of case study

The most relevant points of static resource analysis relevant to our study are: (1) Upper bounds are cost expressions that might include polynomial, logarithmic, exponential subexpressions (and any combination of them). (2) For simplicity, in the example we have showed the result of COSTABS in asymptotic form (or big O notation), i.e., we have removed all the constants as the expression was rather large. The result provided by the analyzer is a precise upper

bound that includes also 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 in most methods unless they have constant cost. (4) In order to compare the cost 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 [4]. We thus assume the existence of an operator "$<$" which allows us to compare two upper bounds. (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.

5. Scenarios for Resource-Aware Configuration

In the remainder of the section, we discuss four ways to carry out the interaction between *Analyzer* and *Configurator*, and point out advantages and drawbacks of each of them. In order to define such scenarios, we assume the existence of an *entry* method that corresponds to the *main* method from which the analysis starts. This is without loss of generality since, if there is no *entry* method, then the same approach can be applied to any other method or set of methods of interest.

5.1. Product-Level Analysis

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

1. Given the product-line infrastructure PL and the set of key features K, we first obtain the set of final (valid) configurations $Conf$ (Section 3);
2. for each $C_i \in Conf$, we generate a product $P_i \equiv P(C_i)$, and analyze it by running $Analyzer(P_i, entry, R)$ where R is the resource metric 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.

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) we have techniques for comparing upper bounds and choosing the minimum [4].

This scenario produces the most accurate results, though *soundness of the selected candidate* is not ensured. Note that the resource analysis guarantees that the obtained upper bounds u_i are sound, i.e., u_i correctly over-approximates the resource consumption of the product P_i for any possible input data. However, it is not guaranteed that 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 analysis of one product can be larger than the loss in the analysis of another. 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 scenario should produce very accurate results.

Disadvantages. The main drawback of this approach is its inefficiency. For a configuration tree with k leaves, we need to invoke the product generator and the 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-sized programs, their application to commercial products is still a research challenge. In conclusion, we argue that, although this scenario is feasible in theory, it is beyond the current state of the practice.

Example 5.1. *In the 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 prohibitive.*

5.2. Partial-Product-Level Analysis

It is quite natural to think of an *interleaved* cooperation between the resource analyzer and the configurator in such a way that *Configurator* 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 the *partial configurations* built throughout the configuration process. 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. Given the product line infrastructure PL and the set of key features K, we proceed to construct the *configuration tree* τ (Section 3). Partial-product-level analysis consists of the following steps:

1. For each partial configuration $C \in \mathit{PartConf}$ of τ, we generate a partial product $P(C)$;
2. we *incrementally* analyze the partial product executing $\mathit{Analyzer}(P(C), \mathit{entry}, R)$, reusing the results inferred for the partial products of ancestor nodes of the configuration tree;
3. we decide if the estimated resource (entry, u) for $P(C)$ is "acceptable"; otherwise, we prune this branch of τ (i.e., the children of C will not be considered).

In order to decide if the resource consumption is acceptable, the user can set up a threshold (or maximal amount of resources) *Limit* before starting the configuration process. Thus, in step 3, we simply check if $u > \mathit{Limit}$ to decide if the branch must be pruned. Another possibility is to compare the resource

consumption of all candidates by keeping the results for the best product constructed so far (namely, u_{min}), and prune a branch if a partial product already exceeds u_{min}.

Example 5.2. *Let us consider that the user imposes as threshold Limit = size(fileset)^2, i.e., the set of files that are to be transferred can be traversed at most a quadratic number of times. During the construction of the configuration tree, as soon as we choose a feature that triggers a delta that includes method* `transferItems` *(see Example 4.1), the threshold provided by the user is exceeded since the resource consumption of* `transferItems` *previously computed size(fileset)*size(rdir)^2 + size(fileset)^3*size(rdir) > size(fileset)^2. Thus, the current branch of the configuration tree is pruned and this feature is not selected.*

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 applying the deltas, *incremental resource analysis* [5] 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 in [5] that incremental resource analysis can save up to 50% of the analysis time when compared with non-incremental whole program analysis.

Disadvantages. A problem with this approach is that it is not feasible using current technology. First, generating partial products is not always feasible: a partial product is, in general, incorrect code since relevant parts of the code might be missing. The product generator aims at building a final product; using our current technology from the HATS project tool suite, 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 generating the product. Even if there were a product generator for partial products, this approach has some efficiency and soundness issues.

Concerning efficiency, *Analyzer* has to be invoked, in the worst case, on all intermediate nodes *PartConf* and leaves *Conf* of τ. Consider a complete binary tree with n leaves: the analyzer will have to be invoked up to $m \equiv 1{+}2{+}4{+}\ldots{+}n$ times, where each summand corresponds to the number of nodes in the corresponding level of the tree. Thus, for our case study $m > 768$.[2] In contrast, in product-level analysis (Section 5.1) it is invoked only n times, which is strictly smaller than m. The analyses can be performed incrementally and, according to [5], each partial analysis can achieve a gain of around 50% w.r.t. analyzing the full product every time. Therefore, according to the current state of the practice, the efficiency approach is comparable to product-level analysis.

[2] We do not have the concrete value of m for our case study because we use the tool that computes the configuration tree as a black box in our implementation.

On the other hand, soundness is potentially lost whenever a branch in the selection tree is pruned. This is because choosing the *local best* solution does not necessarily lead to the *global best* solution, since a feature added in a later selection might affect the resource consumption significantly. Let as illustrate this issue with a very simple example.

Example 5.3. *Consider the partial product resulting from applying the delta* `foo1` *to* `entry` *in Listing 6. If we now 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` *is* $(n_0 - x_0) * (x_0)$*, which is quadratic. This is because we invoke 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 the quadratic cost.*

```
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: Unsoundness of Partial-Product Analysis

The above example reveals that it is not sound 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 the deltas are applied in the inverse order). Therefore, sound results can only be obtained by building full configuration trees and analyzing the resulting complete products. We conclude that this scenario can be of interest only if: (1) we have tools to build partial products; (2) incremental analysis performs much more efficiently than whole-program analysis; and (3) we have a language in which the new increments (deltas) applied to a partial product cannot interfere with the existing code in the partial product. In the absence of such conditions, product-level analysis seems to be a better choice.

5.3. Feature-Level Analysis

With the aim of devising a more practical scenario, 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. We denote by P_0 a product with the minimal number of features needed to get a valid

configuration[3]. Then, the *minimal product* for f (denoted $P_{min}(f)$) is obtained by applying feature f to P_0. We note that, due to requirements among features, adding f may result in adding other features, as the following definition states.

Definition 5.4 (minimal product). *A minimal product for an optional feature f consists of:*

(a) all mandatory features;

(b) the feature f under consideration;

(c) for every group of features to be included, n_1 features chosen randomly (preserving the validity of the configuration), where n_1 is the minimum number of features to be selected;

(d) any feature required by this selection according to cross-tree relations.

Minimal products for features in alternative groups are obtained in a similar manner: they have to include

(a) all mandatory features;

(b) from the alternative group of the feature f under consideration, n_1 features chosen randomly (preserving the validity of the configuration), but including f itself, where n_1 is the minimum number of features to be selected;

(c) for every other alternative group, n_1 features chosen randomly (preserving the validity of the configuration);

(d) any feature required by this selection according to cross-tree relations.

As discussed in Section 2.2, non-mandatory features are (1) those marked with an **opt** modifier, or (2) those appearing in an alternative group (**group oneof** construct). Concerning disjunctive groups (**group** `[`$n_1 \ldots n_2$`]` or **group** `[`$n_1 \ldots$`*]`), every product must include n_1 to n_2 features belonging to the same group, where the `*` symbol stands for n_2 when the group has n_2 features; the construct **group oneof** is a special case of disjunctive group where $n_1 = n_2 = 1$.

The intuitive idea of the feature-level scenario is that the resource consumption due to feature f, denoted U_f, can be approximated as $U_f = R_f - R_0$, where R_f is the resource consumption obtained by analyzing $P_{min}(f)$ and R_0 is the resource consumption obtained by analyzing P_0 for the considered *entry*. The generation and analysis of $P_{min}(f)$ needs to be performed only for non-mandatory features, since the analysis of $P_{min}(f)$ aims at deciding whether selecting f is good from the point of view of performance. This makes no sense for mandatory features because they will be selected in any case. Given a product-line with non-mandatory features $f_1..f_n$, the feature-level analysis is performed as follows:

[3]It should be noted that P_0 can be non-unique because we may want to randomly re-select every time alternative features.

1. generate the minimal product P_0, invoke once $Analyzer(P_0, entry, R)$, and get the resource estimate $(entry, R_0)$;
2. generate the minimal products $P_{min}(f_i)$ for $1 \leq i \leq n$;
3. analyze all products by running $Analyzer(P_{min}(f_i), entry, R)$, get resource estimates $(entry, u_i)$ and obtain the resource consumption due to feature i as $U_i = u_i - R_0$;
4. given a candidate configuration $C \equiv \{f_{i_1}, \ldots, f_{i_s}\} \in Conf$, its resource consumption is $U \equiv U_{i_1} \oplus \ldots \oplus U_{i_s}$ (using some specific operator $\oplus$), where U_{i_j} is the resource consumption obtained for f_{i_j} in step 3.

Note that if the minimal product P_0 does not contain an entry method, then $R_0 = 0$. Similarly, if the selection of one feature does not modify the cost, then u_i will be the same as R_0 and the resource consumption associated to the feature will be 0. In the above steps, we use a generic operator $\oplus$ to combine the resource consumption of a set of features. One could think of using + and simply accumulate the resource consumption contributed by all features (as we do in the implementation), or using max to be able to discard products including features with very high resource consumption. Given k candidate configurations $C_1 \ldots C_k$, the best candidate is the one whose resource consumption is minimal.

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 case study, 13 instead of 768), thus making this approach much more feasible than the first two ones; (2) the whole analysis process can take place *before* the configuration begins and thus there is no need to design a complex interaction between *Analyzer* and $Configurator$, this has an important advantage over the partial-product analysis.

Disadvantages. Clearly, this approach is not sound because it does not consider the interaction of different non-mandatory features in a product (i.e., products that are not minimal in the sense described above). However, as we have noted before, the other two scenarios are not sound either, though the loss of soundness in the first scenario should be rather negligible. In order to consider the interaction among features, one could adopt intermediate solutions (e.g., those proposed in [25] for a different measurement-based approach) and generate minimal products also for combinations of interacting features, instead of considering them in isolation.

Example 5.5. *In the* `ReplicationSystem` *example, there are nine optional features and two alternative (***group oneof***) groups, each containing two alternative features; thus,* 13 *minimal products are generated and analyzed. This is a great improvement over the 768 products to be analyzed in the first approach. However, the feature-level 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.*

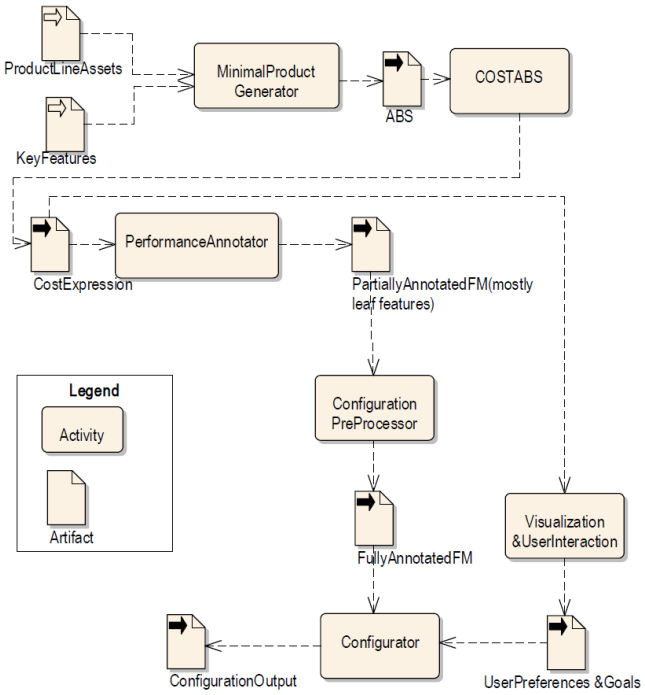

Figure 1: The Resource-Aware Configurator

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

6. Implementation and Preliminary Experimental Evaluation of a Feature-Level Resource-Aware Configurator

This section describes our implementation of the feature-level scenario and applies it to the case study. We have developed a prototype resource-aware product configurator that implements the feature-level scenario described in Section 5.3. Figure 1 provides an overview of the workflow in our tool.

The first phase consists in generating *performance annotations* by carrying out the following steps:

(1a) Given a product-line infrastructure *PL* and a selection of key features, the component *MinimalProductGenerator* generates the minimal products for each non-mandatory feature;

(1b) the off-the-shelf resource analyzer COSTABS [3] is used to analyze the minimal products; and

(1c) the component *PerformanceAnnotator* takes the cost expression returned by COSTABS and uses it to annotate the feature model with performance annotations.

The final output of this phase is an *PartiallyAnnotatedFeatureModel*.

The second phase is the *product configuration* itself, in which:

(2a) the *PartiallyAnnotatedFeatureModel* is preprocessed to derive annotations for upper-level features from the annotations provided by the *PerformanceAnnotator*;

(2b) the component *Visualization&UserIntercation* asks the user to provide his/her quality constraints and concerns;

(2c) the configurator suggests a small set of valid product configurations that best fit the objective function representing the user's input; and

(2d) the user selects one of those configurations.

The final output is a PSL specification from which the product can be generated.

We describe below the main decisions made during the implementation of the above components.

6.1. Performance Annotation

The implementation of the performance annotator comprises steps (1a), (1b) and (1c) described in Section 5.3. The main decisions that have been taken in the implementation are: (1) considering the resource consumption of those methods potentially modified by the selection of a feature (named *footprint* of the feature), instead of looking at the entry method only, as described below; and (2) providing the configurator with *performance annotations* that are natural numbers instead of the cost expressions inferred by the resource analysis.

Generation of minimal products. First, a minimal product $P_{min}(f)$ is computed and generated for every non-mandatory feature f. Computing a minimal product involves reading the μTVL definition of the feature tree and identifying the set of non-mandatory features. Afterwords, the features that will be included in a minimal product $P_{min}(f)$ are selected following the definition of minimal product given in Section 5.3. Given the feature specification for $P_{min}(f)$, the actual product can be generated by existing tools available in the HATS project (see Section 1).

Example 6.1. *In the case study, the minimal product for* P_{min}(`Cloud`) *for* `Cloud` *includes all mandatory features,* `Cloud` *itself, and either* `Seq` *or* `Concur` *(chosen randomly). This configuration is the minimal valid configuration containing* `Cloud`*. The minimal product for* `Concur` *involves selecting all mandatory features,* `Concur` *itself, and* `Cloud`*, which is required by* `Concur`*. Finally,* P_{min}(`Client`) *consists of all mandatory features,* `Client`*, one between* `Site` *and* `Cloud`*, and one*

between `Seq` *or* `Concur` *(when* `Concur` *is selected,* `Cloud` *has also to be chosen). In all these cases, no optional features are selected unless they are strictly required.*

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 tool) generated in the tests, either minimal or not, have roughly the same size. Only considering the core-ABS code (i.e., excluding delta code, product declarations, etc.) gives a size of around 2.000 lines for all generated products (this will be discussed later in Table 2), 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, COSTABS analyzes each of them. The resource of interest is a parameter of the analyzer, which, in the current implementation of our solution, is either *response time*[4] or *memory consumption*.

When analyzing a piece of code, a typical choice in static analysis, is to analyze the entry method (e.g., the `main` method in a Java program). This usually involves analyzing all or most of the code, since such a method will probably invoke many other methods during its execution.

On the other hand, for a given minimal product $P_{min}(f)$, we are interested in reasoning only on the resource consumption of methods that are specific to the feature f under consideration. In general, the selection of f only affects a limited portion of the code, and the analysis benefits from identifying this portion. The definition of the product-line (Section 2.4) specifies which deltas must or could be active when f is selected. Conservatively, we take all deltas whose associated delta clause has an application condition where f occurs; this is an approximation since, it can be the case that a delta 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 the delta. 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 analyzing the minimal product for `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 the delta declarations collects all the methods added or modified by such deltas[5]. This set is called the *footprint* of f, and can be computed statically. For every method in *Footprint*(f), COSTABS is run and a cost

[4]It must be clear that response time does not refer to actual execution time in milliseconds; rather, it represents the number of executed instructions.

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

expression is obtained. Thus, the output of this step is a set of upper bounds (m_i, u_i) for each $m_i \in \mathit{Footprint}(f)$.

Example 6.2. *A number of 138 methods are defined in the minimal product* P_{min}(`File`)*. 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`*.*

Performance annotations. While the cost expressions u_i that COSTABS outputs 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 the results become hard to interpret, especially from the point of view of the user. For instance, deciding if a cost expression e is smaller than another one e' requires the use of specific techniques [4]. The result of the comparison is often not simply a Boolean answer, but rather constraints on the variables of e and e' under which the comparison can be proved.

As a practical solution, we consider transforming cost expressions into *performance annotations*, which indicate the resource consumption of executing a method by means of an integer number: the higher the number, the higher the consumption. In other words, (m_i, u_i) is transformed into (m_i, a_i), where a_i is the annotation. In order to carry out this mapping, we first transform u_i into *asymptotic* form (big O notation). This transformation can always be applied, and can be done locally and efficiently [2]. The next step is to map the asymptotic bound to a performance annotation. This mapping can be done using different heuristics, and the effectiveness of the transformation can only be assessed experimentally. For example, the current implementation makes the following choice:

```
  0 : constant cost
100 : logarithmic, sublinear or linear cost
200 : polynomial cost (up to degree 3)
300 : high-degree polynomial or exponential cost
400 : unknown (the analyzer could not get an upper bound)
```

Example 6.3. *Consider the cost expression that has been obtained from the analysis of 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 (by $\oplus$, see Section 5.3) per-method annotations a_i into a per-feature annotation a_f describing the performance of f. A reasonable

heuristic is required here, whose discussion is beyond the scope of this article. Simple heuristic functions such as addition, average or max can be easily implemented and evaluated experimentally. In the implementation, we are currently using the average of the performance levels of all methods in the footprint of the feature.

Example 6.4. *We have analyzed all methods in the footprint of* `File` *(Example 6.2) w.r.t. the cost metrics "response time". Once COSTABS generates performance annotations for all of them, the overall annotation* a_{File} *has been obtained as the average and the result is 133. This number is obtained by analyzing all the nine methods in the footprint: 5 of them have constant cost (performance level 0), 2 of them give a low-degree polynomial upper bound (performance level 200) while the last two cannot be analyzed (performance level 400). The final performance level is given to Configurator as an extension using the following syntax:*

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

Listing 7: Performance Annotation by COSTABS

The first instruction declares `im_memoryConsumption` *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 performance annotation is 133 whenever the feature is selected. Three lines are output for every different cost model considered in the analysis.*

6.2. Product Configurator

The configuration preprocessor combines the annotations a_f provided in the previous phase. Listing 8 shows an excerpt with the definition of preprocessing options for memory consumption. It states that the annotation of a higher-level feature is done using the `compositionOperator`, which is "+" in this case.

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

Listing 8: Preprocessing for Configuration

The resulting annotation will be a formula:

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

which represents the sum of the memory consumption of all children present in the configuration; if $childFeature_i$ is not selected in the configuration, $childFeature_i.im\_memoryConsumption = 0$. 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 root feature's attributes. 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, $im\_memoryConsumption$ can be more important than $im\_responseTime$ to the user. 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 one 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 quality metrics or it can be related with multiple metrics. Once the objective function and the quality constraints are elicited, the configurator finds suitable product configurations for the user.

Example 6.5. *In our example feature model, we have the feature* `Client` *and the user wants his product to be launched in a very thin client with respect to memory. She can set constraint on that feature specifying how much memory consumption she can tolerate. The constraint is specified as follows:*

$$Client.im\_memoryConsumption \leq 2$$

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 visualizing the results and eliciting the user's quality and functional requirements, the open source tool *FeatureIDE* is used.

6.3. Preliminary Experiments on the Case Study

We have implemented the generator of performance annotations as an extension of the COSTABS analyzer: it takes as input the ABS files containing all the code, and outputs a μTVL file with the performance annotations described in Example 6.4. In the case study, the code is contained in two files: `ReplicationSystem.abs` contains the module declaration, the core code, the delta definitions, the product-line configuration, and some product specifications; `Features.abs` contains the feature model. The total size of both files is 3.548 lines of code.

The resource under consideration is response time. As mentioned before, in the feature model there are 8 mandatory features and 13 optional or alternative features; consequently, 13 minimal products have to be generated. For each of them, the footprint is computed, and the core part of COSTABS (i.e., the

f	$t(P_{min}(f))$	$F(P_{min}(f))$	$M(P_{min}(f))$	$L(P_{min}(f))$	$FP(f)$	$t(a_f)$	a_f
`Client`	1643	11	132	2030	3	1783	0
`Server`	1655	11	132	1977	1	1694	0
`File`	1669	11	140	2053	9	15675	133
`Journal`	1647	11	139	2069	7	4309	86
`Update`	1659	11	131	1972	1	1686	0
`ClientNr`	1645	11	131	2024	2	1746	0
`Search`	1652	11	132	2030	2	2042	200
`Business`	1641	12	142	2063	2	2035	200
`Data`	1641	13	150	2160	3	2112	133
`Seq`	1633	10	130	1972	21	9952	76
`Concur`	1658	10	133	2025	24	14874	163
`Site`	1657	10	132	1972	0	1658	0
`Cloud`	1645	10	132	1972	0	1645	0

Table 2: Experimental evaluation on the case study

static analyzer properly said) is called once for each method in the footprint. Experiments on performance annotation have been carried out on a MacBook Pro with a 2.4 GHz Intel Core i5 processor and 4Gb of memory, running Mac OS 10.7.5. The execution has been repeated 5 times, and reported times (expressed in milliseconds) are computed as the average of all the executions. Table 2 summarizes the experiments: column "f" is the name of the feature under study; "$t(P_{min}(f))$" is the time needed to generate the corresponding minimal product by using existing tools; "$F(P_{min}(f))$", "$M(P_{min}(f))$" and "$L(P_{min}(f))$" 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 performance annotation of the minimal product; finally, "a_f" is the global performance annotation (according to Section 6.1, it ranges from 0 to 400; the lower, the more efficient).

Let us draw some conclusions from the table. 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 HATS tools for product generation) is also similar for all of them. 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 performance 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 COSTABS; this time is obtained by putting a 10-seconds timeout on each call to the analyzer

(such timeout is only reached once when analyzing a method in *Footprint*(`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 COSTABS 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 the future work.

By using the performance annotations shown in the table, the resource-aware configurator suggested the following configuration:

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

In absence of performance annotations, the configurator had suggested another configuration:

{`ReplicationSystem`, `Installation`, `Resources`, `JobProcessing`, `ReplicationItem`, `Dir`, `Load`, `Schedule`, `Site`, `Concur` }

whose overall performance annotation (according to the preprocessing for configuration described in Listing 8, which simply sums the performance annotations of all features) is worse than the one chosen by considering performance issues. As expected, using the annotations, we obtained a configuration that has a better overall performance level.

We argue that our experiments, even if still at a very preliminary stage, constitute a proof of concept that resource-aware configuration is feasible. However, it still remains to see how close our performance 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-level scenario. In the experiments, we used the average performance annotation for all the methods in the footprint as the final annotation of a feature, and the addition of the performance annotations of all features as the performance annotation of a product. Obviously, other choices could have been taken. Future work includes proposing new heuristics that allow us to have annotations which are closer to the actual resource consumption, and undertaking a thorough experimental evaluation.

7. Related Work

Finding a selection of the features that satisfies the requirements of a concrete product and adheres to the feature model rules involves reasoning over a complex set of constraints to meet a final goal. Interactive product configuration support uses the feature model rules to propagate configuration choices made by the user [11], whereas automatic product configuration support provides a set of configurations that satisfy the feature model rules and the user's requirements and constraints.

Table 3: Support for Feature Selection

Main Characteristic	**Support Type**	**NF Concerns**	**Underlying Technology**
Multi-level staged [10]	Interactive	Security	Specialized FMs
Probabilistic [11]	Interactive	No	Conditional probabilities and legal Joint Probability Distributions
Dynamic [21]	Automatic	No	Binding analysis and reconfiguration strategy
Multi-step [32]	Automatic	Cost	Constraint Satisfaction Problem
Polynomial-time [31]	Automatic	Yes	Multi-dimensional Multi-choice Knapsack Problem
Business concern annotation [29]	Automatic	Yes	Hierarchical Task Network
Multi-view [1]	Interactive	No	Workflow management tool
Performance measurement [25]	Automatic	Resource	SAT Solvers

Table 3 summarizes a literature review on product configuration support. Concerning *Support Type*, the selection of features in our resource-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. If those features do not infringe upon any rule, 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.

Several efforts have been made around research on product configuration. Quality-aware configurations require modeling variability in quality attributes. Sinnema et al. [28] described COVAMOF, a modeling technique for variability. Etxeberria et al. [13] presented a survey on existing approaches including COVAMOF for specifying variability in quality attributes, the requirements for a quality variability modeling technique and comparison of the techniques. Hubaux et al. [17] presented the multidimensional separation of concerns in feature-based configuration, i.e., generating concern-specific configuration views. They also presented in [16] the feature configuration workflow, with scheduling parts of the feature diagram as part of the configuration process. Tun et al. [30] showed a systematic approach to relate requirements to features.

As regards *Non-Functional Concerns*, several approaches take into consideration cost constraints, but only few of them consider quality concerns such as

performance and security. Some notable exceptions are [27, 25, 26]. The most related work to ours is [25] which is also concerned with deriving optimal products with respect to non-functional requirements by showing customers which features must be selected. Similar to our feature-level analysis, they propose an approach to predict product's non-functional properties by aggregating the influence of each selected feature on a non-functional property to predict a product's properties. They also try to generate and measure a small set of products and, by comparing measurements, they approximate each feature's influence on the non-functional property in question. A fundamental difference is that our approach to estimate the performance of features is based on a formal method (i.e., static resource analysis), while [25] performs measurements. 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 which are safe for any possible input data value. In addition to the advantage of having a result which is valid for any input data, an important consequence of this different choice is that we can analyze partial products and focus on the performance behavior of fragments of the product (e.g., the footprint) while, because they perform measurements, they need to analyze it globally. This gives us further flexibility. The interaction of features is not taken account in our analysis explicitly, though there is some implicit interaction which is taken into account with the mandatory features (since they are in the minimal product). The techniques proposed in [25] to take into account such interaction can be also used in our approach without requiring any conceptual change to our scenarios.

Regarding *Underlying Technology*, we have adopted the CSP solver called *Choco Java* because: (1) the mapping of the product-configuration problem into CSP [32] is intuitive; and (2) if required, there are translators from CSP into *Satisfiability Modulo Theories* (SMT), which can be adopted to deal with performance issues.

8. Conclusions and Future Work

In this article, we have introduced the notion of *resource-aware configuration* which strives for finding a selection of features which leads to a product that has an efficient performance and that complies with the quality constraints provided by the user. We have envisaged several scenarios for resource-aware configuration and described a prototype implementation of the most practical scenario. Our implementation shows that it is feasible to use an off-the-self analyzer to obtain performance indicators that can be used to annotate feature models. Using the annotated feature models, the configurator is able to suggest a small set of valid product configurations that best fit the objective function representing the user's input.

Our implementation and its application to the case study constitutes a proof of concept for resource-aware configuration. However, a thorough experimental evaluation is required to assess the accuracy of the envisaged scenarios 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 cost expressions into performance 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.

[1] Ebrahim Abbasi, Arnaud Hubaux, and Patrick Heymans. A toolset for feature-based configuration workflows. In *Proceedings of SPLC'2011*, pages 65–69. IEEE, August 2011.

[2] E. Albert, P. Arenas, D. Alonso, S. Genaim, and G. Puebla. Asymptotic Resource Usage Bounds. *Proceedings of APLAS'2009*, volume 5904 of *Lecture Notes in Computer Science*, pages 294–310. Springer, December 2009.

[3] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. COSTABS: A Cost and Termination Analyzer for ABS. In *Proceedings of PEPM'2012*, pages 151–154. ACM Press, 2012.

[4] E. Albert, P. Arenas, S. Genaim, I. Herraiz, and G. Puebla. Comparing cost functions in resource analysis. In *Proceedings of FOPARA'2009*, volume 6234 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2010.

[5] E. Albert, J. Correas, G. Puebla, and G. Román-Díez. Incremental Resource Usage Analysis. In *Proceedings of PEPM'2012*, pages 25–34. ACM Press, January 2012.

[6] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6), 2004.

[7] Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. Variability modelling in the ABS language. In *Proceedings of FMCO'2010*, volume 6957. Springer-Verlag, 2011.

[8] A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, November 2010.

[9] K. Czarnecki and U.W. Eisenecker. *Generative Programming*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[10] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process Improvement and Practice*, 10(2), 2005.

[11] K. Czarnecki, S. She, and A. Wasowski. Sample spaces and feature models: There and back again. In *Proceedings of SPLC'2008*, pages 22–31. IEEE, Setember 2008.

[12] F. S. de Boer, R. Hähnle, E. Broch Johnsen, R. Schlatte, and P. Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study. In *Proceedings of ESOCC*, volume 7592 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2012.

[13] Leire Etxeberria, Goiuria Sagardui Mendieta, and Lorea Belategi. Modelling variation in quality attributes. In *Proceedings of VaMoS'2007*, pages 51–59, 2007.

[14] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proceedings of POPL'2009*, pages 127–139. ACM, 2009.

[15] J. Hoffmann and M. Hofmann. Amortized Resource Analysis with Polynomial Potential. In *Proceedings of ESOP'2010*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010.

[16] A. Hubaux, A. Classen, and P. Heymans. Formal modelling of feature configuration workflows. In *Proceedings of SPLC'2009*, pages 221–230, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[17] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, Dirk Deridder, and Ebrahim Abbasi. Supporting multiple perspectives in feature-based configuration. *Software and Systems Modeling*, pages 1–23, November 2011.

[18] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A Core Language for Abstract Behavioral Specification. In *Proceedings of FMCO'2010, Revised Papers*, volume 6957 of *LNCS*, pages 142–164. Springer, 2012.

[19] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, SEI, 1990.

[20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP'1997*, volume 1241 of *Lecture Notes in Computer Science*, 1997.

[21] J. Lee and K. C. Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In *Proceedings of SPLC'2006*, pages 131–140. IEEE, August 2006.

[22] K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, Heidelberg, 2005.

[23] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending feature diagrams with uml multiplicities. In *Proceedings of IDPT'2002*, pages 23–27, June 2002.

[24] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proceedings of SPLC'2010*, September 2010.

[25] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. S. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of ICSE*, pages 167–177, 2012.

[26] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable Prediction of Non-functional Properties in Software Product Lines. In *Proceedings of SPLC'2011*, pages 160–169, 2011.

[27] J. Sincero, W. Schröder-Preikschat, and O. Spinczyk. Approaching Non-functional Properties of Software Product Lines: Learning from Products. In *Proceedings of APSEC*, pages 147–155. IEEE Computer Society, 2010.

[28] Marco Sinnema, Sybren Deelstra, Jos Nijhuis, and Jan Bosch. COVAMOF: A Framework for Modeling Variability in Software Product Families. In *Proceedings of SPLC'2004*, volume 3154, pages 197–213. Springer, 2004.

[29] Samaneh Soltani, Mohsen Asadi, Marek Hatala, Dragan Gasevic, and Ebrahim Bagheri. Automated planning for feature model configuration based on stakeholders' business concerns. In *Proceedings of ASE'2011*, pages 536–539. IEEE, November 2011.

[30] T. Than Tun, Q. Boucher, A. Classen, A. Hubaux, and P. Heymans. Relating requirements and feature configurations: a systematic approach. In *Proceedings of SPLC'2009*, pages 201–210, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[31] Jules White, Brian Dougherty, and Doulas Schmidt. Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Systems and Software*, pages 1268–1284, August 2009.

[32] Jules White, Brian Dougherty, Doulas Schmidt, and David Benavides. Automated reasoning for multi-step feature model configuration problems. In *Proceedings of SPLC'2009*, pages 11–20. Carnegie Mellon University, August 2009.