Deployment Variability in Delta-Oriented Models [*]

Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa

Department of Informatics, University of Oslo, Norway
`{einarj,rudi,sltarifa}@ifi.uio.no`

Abstract

Software engineering today increasingly emphasizes variability by developing families of software products together to satisfy a range of application contexts or user requirements, where the different products vary in the features they support. In feature modeling, much focus has been on selecting functional features for different products and on the software quality attributes for features and products. However, the quality of the selected functional features of a product also depend on how the product is deployed.

ABS is a modeling language which supports variability in the formal modeling of software by using feature selection to transform a delta-oriented base model into a concrete product model. Recently, ABS has been extended for the timed modeling of deployment architectures, based on a separation of concerns between execution cost and the server capacity in the model. This allows the effect of deployment choices for a product to be observed on its quality of service. This paper combines deployment models with the variability concepts of ABS, in order to model deployment choices as features when designing a family of products.

1 Introduction

Variability is prevalent in modern software systems in order to satisfy a range of application contexts or user requirements [34]. A software product line (SPL) realizes this variability by providing a family of product variants, see, e.g., [29]. A specific product is obtained by selecting features from a feature model, which typically focuses on the functional features of different products and on the software quality attributes of different features and products. Many variants of feature-based variability modeling exist; for a survey on different feature model languages [36].

To express variability in the design of systems, features typically take the form of architectural models, behavioral models, and test suites [35]. Architectural variability focuses on the presence of component variants, and can be described using, e.g., the Variability Modeling Language [27], UML stereotypes [13], or (hierarchical) component models such as Koala [37]. Delta modeling [8, 30, 31] is a variability modeling concept in which a set of system deltas specify modifications to a core product in order to obtain other products. Δ-MontiArch applies delta modeling to architectural description languages [14]; a delta can add or remove components, ports, and connections between components.

Complementing architectural models, which are concerned with describing the *logical* organization of a system in terms of components and their connections, we are interested in the *physical* organization of software units on physical (or virtual) machines; we call this physical organization the *deployment architecture*. Our approach to describing deployment architectures is based on a separation of concerns between the *application model*, which requires resources, and the *deployment scenario*, which reflects the virtualized computing environment which provides heterogeneous amounts of resources. For example, the functional features which can be

[*]Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu`).

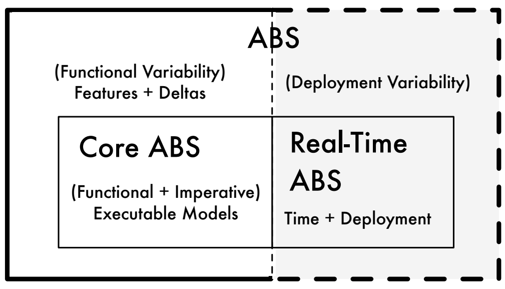

Figure 1: ABS language extension.

selected for the different products in a cellphone SPL, depend on the physical capacity of the different cellphones; e.g., a cell phone with limited processing capacity may require a simpler camera application than a very powerful cell phone. In the virtualized setting, an application model may be analyzed with respect to deployments on virtual machines with varying features: the amount of allocated computing or memory resources, the choice of application-level scheduling policies for client requests, or the distribution over different virtual machines with fixed bandwidth constraints.

In this paper, we are interested in how deployment variability can be integrated in SPL models. By introducing deployment variability at the design stage of SPL engineering, the different targeted deployment architectures of the SPL may be taken into account for different feature configurations early in the design of the SPL. Our starting point for this work is the recently developed **a**bstract **b**ehavioral **s**pecification language ABS, which adds support for variability to models in the kernel modeling language Core ABS [18]. ABS is *object-oriented* to stay close to high-level programming languages and to be easily usable as well as accessible to software developers, it is *executable* to support full code generation and (timed) validation of models, and it has a *formal semantics* which enables the static analysis of models; e.g., the worst-case resource consumption can be derived for a model. ABS is particularly suitable for our objective because (1) ABS supports SPL modeling based on deltas [7, 9], and (2) ABS supports the modeling of deployment decisions based on the modeling concept of *deployment components* [21] in the real-time extension Real-Time ABS [5]. Real-Time ABS allows resources and their dynamic management to be leveraged to the abstraction level of software models. Figure 1 shows how functional variability modeling in ABS extends Core ABS, and how time and deployment models in Real-Time ABS extend Core ABS. Although these extensions coexist for the same modeling language, these two aspects of ABS have so far never been combined. The purpose of this paper is to combine these two extensions in order to model deployment variability, corresponding to the dotted area in Figure 1.

This paper proposes a way to introduce deployment variability into models of SPLs. The main contributions of the paper are:

- we integrate delta models with deployment components in the ABS modeling language;
- our integration allows orthogonality between functional variability and deployment variability;
- the integration is illustrated by variability patterns for MapReduce [11], a programming model for highly parallelizable programs; and
- the integration allows ABS tools to be used to analyze functional features with respect to a deployment scenario during the early design stage of an SPLs.

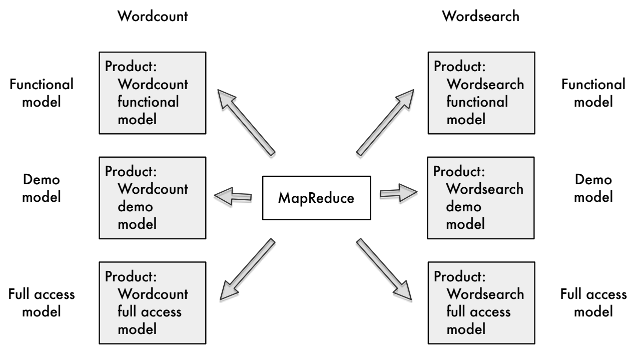
Wordcount
Wordsearch
Functional model
Product: Wordcount functional model
Product: Wordsearch functional model
Functional model
Demo model
Product: Wordcount demo model
MapReduce
Product: Wordsearch demo model
Demo model
Full access model
Product: Wordcount full access model
Product: Wordsearch full access model
Full access model

Figure 2: A family of products sharing an underlying MapReduce structure.

Paper overview The paper is organized as follows. Section 2 motivates our work by introducing a running example of deployment variability. Section 3 introduces the basic modeling level of the abstract behavioral specification languages ABS. Section 4 introduces delta-oriented variability and its realization in the ABS modeling language. Section 5 explains the approach to deployment modeling taken in ABS. Section 6 combines delta-oriented variability with deployment modeling, and discusses how to extend a feature model with deployment variability. Section 7 presents the model and simulations of our case study. Section 8 discusses related work and Section 9 concludes the paper.

2 Motivating Example

MapReduce [11] is a programming pattern for processing large data sets in two stages; first the *Map* stage separates highly parallelizable jobs on distinct parts of the data to produce intermediate results, then the *Reduce* stage merges the intermediate data into a final result. The initial and intermediate data are on the form of key/value pairs, and the final result is a list of values per key. MapReduce as such does not specify the computations done by the two stages or the distribution of workloads across machines, making it a good abstract base model for software product lines.

We use MapReduce to model a product line consisting of a range of services which inspect a set of documents. The individual products may implement, among others, Wordcount, which counts the number of occurrences of words in the given documents, and Wordsearch, which searches for documents in which a given word occurs. For simplicity, we will assume that a service only provides one of the Wordcount or the Wordsearch feature. The services are implemented on a cluster of computers, using MapReduce.

For attracting clients to the word count and word search services, there are freely available demo versions of the services, which offer the same functionality as the full versions, albeit with a lower quality of service. In the implementation of the services, the demo versions will run on a few machines, whereas the full versions have access to the full power of the cluster. In our model, we therefore have three versions of each service: the purely functional model, the model with full access to the cluster, and a model with restricted access to the cluster. We will use this product line, depicted in Figure 2, as a running example in the paper.

3 Behavioral Modeling in ABS

The kernel of the abstract behavioral specification language ABS is a formally defined object-oriented language called Core ABS [18], targeting the executable design of distributed systems. ABS is based on concurrent object groups (COGs), akin to concurrent objects [6,19], Actors [1], and Erlang processes [4]. COGs in Core ABS support interleaved concurrency based on guarded commands. This allows active and reactive behavior to be easily combined by means of a cooperative scheduling of processes, which stem from method calls. Core ABS models have a functional and an imperative layer, combined with a Java-like syntax. Real-Time ABS extends Core ABS models with *implicit* time; the execution time is not specified directly in terms of durations (as in, e.g., UPPAAL [26]), but rather *observed* by measurements of the executing model. For the formal definition of the syntax, type system, and semantics of Core ABS, we refer the reader to [18]. For the formal definition of Real-Time ABS, we similarly refer the reader to [5].

3.1 The Functional Layer

The purpose of the functional layer of Core ABS is to describe computation succinctly in a representation independent way. The modeler may abstract from the details of low-level imperative implementations of data structures while maintaining an overall object-oriented design close to the target system. The functional layer of Core ABS consists of algebraic data types such as the empty type Unit, booleans Bool, integers Int; parametric data types such as sets Set<A> and maps Map<A, B> (for type parameters A and B); and functions over values of these data types, with support for pattern matching.

Example 1. To illustrate the definition of data types and functions, *polymorphic sets* can be defined using a type variable A and two constructors EmptySet and Insert. To illustrate function definitions in Core ABS, we show some standard functions on sets: emptySet(xs) checks whether xs is an empty set; insertElement only inserts an element e if the set xs does not already contain e (in fact, our definition of the set does allow multiple occurrences); remove removes an element e from a set xs (assuming single occurrences in the set); and take returns some element of the set xs.

```
data Set<A> = EmptySet | Insert(A, Set<A>);

def Bool emptySet<A>(Set<A> xs) = (xs == EmptySet);

def Set<A> insertElement<A>(Set<A> xs, A e) = case contains(xs, e) {True  => xs;
                                                                    False => Insert(e, xs); };

def Set<A> remove<A>(Set<A> xs, A e) = case xs { EmptySet     => EmptySet ;
                                                 Insert(e, ss) => ss;
                                                 Insert(s, ss) => Insert(s,remove(ss,e)); };

def A take<A>(Set<A> ss) = case ss { Insert(e, _) => e; };
```

3.2 The Imperative Layer

The purpose of the imperative layer of Core ABS is to describe concurrency, communication, and synchronization. This is done at the level of objects, and defines interfaces, classes, and methods. Core ABS objects are *active* in the sense that their run method, if defined, gets called upon creation. Communication and synchronization are decoupled in Core ABS. Communication is based on asynchronous method calls, denoted by assignments f=o!m(e) where f is a future variable, o an object expression, and e are (data value or object) expressions. After calling f=o!m(e), the caller may proceed with its execution *without blocking* while m(e) executes. Two operations on future variables control synchronization in ABS. First, the statement **await** f? *suspends the active process* unless a return value from the call associated with f has arrived, allowing other processes in the same COG to execute. Second, the return value is retrieved by the expression f.**get**, which *blocks all execution in the COG* until the return value is available. Inside a COG, Core ABS also supports standard synchronous method calls o.m(e).

A COG can have at most one active process, executing in one of the objects of the COG. This active process can be unconditionally suspended by the statement **suspend**, adding this process to the queue of the COG, from which an enabled process is then selected for execution. The guards g in **await** g control suspension of the active process and consist of Boolean conditions conjoined with return tests f? on future variables f. Just like functional expressions, guards g are side-effect free. The remaining statements of Real-Time ABS are standard; e.g., sequential composition $s_1; s_2$, assignment x=rhs, and **skip**, **if**, **while**, and **return** constructs. Right hand side expressions rhs include the creation of an object group **new cog** C(e), object creation in the group of the creator **new** C(e), method calls, and future dereferencing f.**get**, in addition to the functional expressions e.

Example 2. To illustrate how the functional and imperative layers of Core ABS are typically combined, let Worker be an interface and workers a set of objects typed by Worker. Sets are defined in Example 1. The method getWorker will only create a new Worker (by instantiating the class Worker) if no Worker is available in the set workers, otherwise it will take one worker which it removes from the set. The method finished inserts w into the set if it is not already there.

```
Worker getWorker() {
  Worker w = null;
  if (emptySet(workers)) { w = new cog Worker(this); nWorkers = nWorkers + 1; }
  else { w = take(workers); workers = remove(workers, w); }
  return w;
}

Unit finished(Worker w) {
  workers = insertElement(workers, w);
}
```

4 Delta-Oriented Variability in ABS

This section describes how software product lines (SPLs) are modeled in ABS. Variability is used to specify that multiple similar models can be created by selecting different instances of features and applied them to a variable source code. ABS includes a delta-oriented framework for variability [7, 9]. Figure 3 depicts a delta-oriented variability model where a feature model F with orthogonal variability [16] is represented as two trees that hierarchically structure the set of features of this model; it is also possible to observe variability at the level of code where a common base model P can be modified by applying delta modifications from the delta model Δ. Sets of features from the feature model F are linked to sets of delta modifications from the delta model Δ, which apply to the common base model P to produce different product line configurations C, C' and C', and finally a specific product ρ is extracted from the product line configuration C.

Feature model A feature model in ABS is represented textually as a forest of nested features where each tree structures the hierarchical dependencies between related features, and each feature in a tree may have a collection of Boolean or integer attributes. The ABS feature model

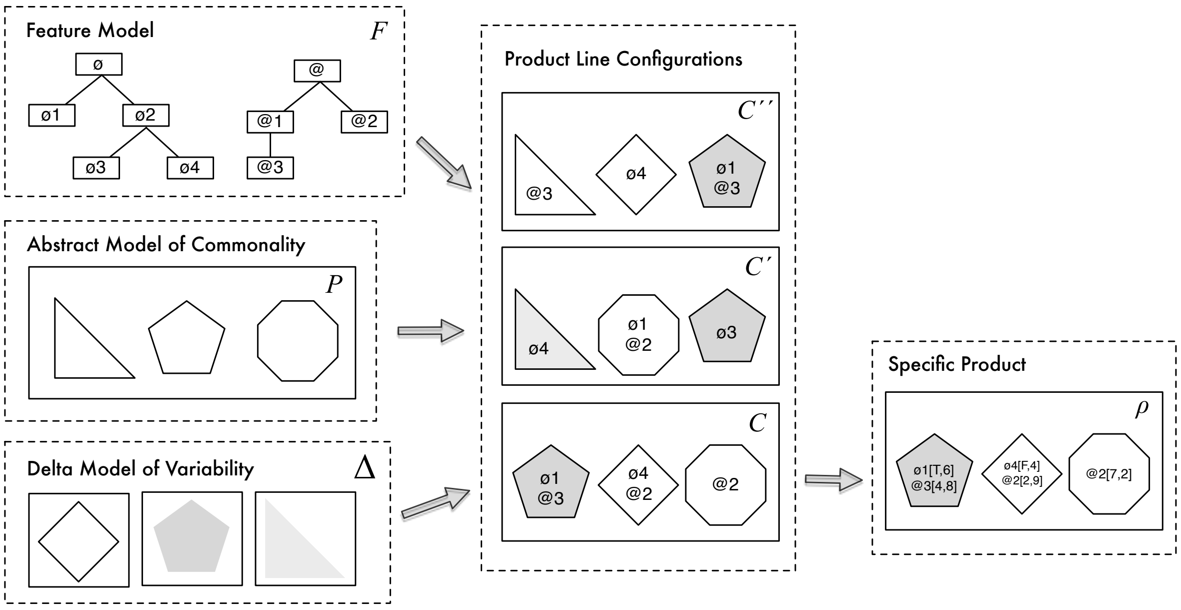

Figure 3: A graphical representation of a Delta-Oriented variability model.

can also express other cross-tree dependencies, such as mandatory and optional sub-features, and mutually exclusive features. The **group** keyword is used to specify the sub-features of a feature; the **oneof** keyword means that exactly one of the sub-features must be selected in the created product line, the range of values associated to an attribute specify the values in which an attribute can be instantiated when an specify product is generated. For the full details, we refer the reader to Clarke et al. [7, 9].

Example 3. In the functional feature model of the MapReduce example from Section 2, a tree with a root Calculations offers two alternative and mutually exclusive features that can be selected to express that a specific product supports counting words or searching for words.

```
root Calculations {
  group oneof {
    Wordcount,
    Wordsearch
  }
}
```

In addition ABS allows a feature model with multiple roots (hence, multiple trees) to describe orthogonal variability [16], which is useful for expressing unrelated functional and other features (e.g., features related to quality of service).

Delta model The concept of delta modeling was introduced by Schaefer et al. [31–33] as a modeling and programming language approach for software-based product lines. This approach aims at automatically generating software products for a given valid collection of features, providing flexible and modular techniques to build different products that share functionality or code. In delta-oriented programming, application conditions over the set of features and their attributes, are associated with units of program modifications called delta modules. These delta modules may add, remove, or otherwise modify code. The implementation of an SPL in delta-oriented programming is divided into a common core module and a set of delta modules. The core module consists of classes that implement a complete product of the SPL. Delta modules describe how to change the core module to obtain new products. The choice of which delta modules to apply is based on the selection of desired features for the final product.

Technically, delta modules have a unique identifier, a list of parameters, and a body containing a sequence of class and interface modifiers. Such a modification can add a class or interface declaration, modify an existing class or interface, or remove a class or interface. The modifications can occur within a class or interface body, and modifier code can refer to the original method by using the **original**() keyword. Delta modules in ABS can be parametrized by attribute values to enable the application of a single delta in more than one context.

Product line configuration The product line configuration links feature models with delta modules to provide a complete specification of the variability in an ABS product line. A product line configuration consists of the set of features of the product line and a set of delta clauses. Each delta clause names a delta module and specifies the conditions required for its application, called application conditions. A partial ordering on delta modules constrains the order in which delta modules can be applied to the core module.

Specific product A product selection clause generates a specific product from an ABS product line. It states which features are to be included in the product and specifies concrete values for their attributes. A product selection is checked against the feature model for validity. The product selection clause is used by the product line configuration to guide the application of the delta modules during the generation of the final product.

Generated final product Given a Core ABS program P, a set of delta modules Δ, a product line configuration C, a feature model F, and a product selection ρ (as depicted in Figure 3), the final product, which will be a Core ABS program, is derived as follows: First check that the product selection ρ satisfies the constraints imposed by the feature model F; then select the delta modules from Δ with a valid application condition with respect to ρ; and finally apply the delta modules to the core program P in some order respecting the partial order described in C, replacing delta parameters in the code with the literal values supplied by the feature.

5 Deployment Modeling in ABS

Real-Time ABS extends Core ABS with primitives to describe deployment architectures. The purpose of describing deployment in a modeling language is to differentiate execution time based on where the execution takes place. With implicit time, no assumptions about execution times are hard-coded into the models. Instead, we want to model how the execution time of a statement varies with the available *capacity* of the chosen deployment architecture and on *synchronization* with (slower) objects. For example, the response time to a request in a distributed system depends not only on the size of the job requested, but also on the amount of available resources and on the usage policy for these resources, which are scattered around the deployment architecture of the distributed system. The execution time of a method call depends on how quickly the call is effectuated by the server object; in fact, *similar calls to the same method do not always take the same amount of time.*

Deployment architectures describe how distributed systems are mapped on physical and/or virtual media with many locations. The planning and validation of a deployment architecture to optimize performance implies determining the amount of necessary resources at the different locations as well as an optimal usage of these resources, such that the system fulfills its performance requirements. Real-Time ABS lifts deployment architectures to the abstraction level of the modeling language, where the physical or virtual media are represented as *deployment components* [20]. In a Real-Time ABS model, different deployment components may have different bounds on the locally available resources.

Real-Time ABS introduces a separation of concerns between the resource cost of performing a computation and the resource capacity of a given deployment component. This separation of concerns between resource *cost* and resource *capacity* aids to model and validate different deployment scenarios at an early stage during the software development process. The focus in this article is on CPU resources in virtualized media; we use resource cost annotations to express resource consumption during computation.

5.1 Deployment Components

A *deployment component* in Real-Time ABS is part of the deployment architecture of the model, on which a number of COGs are deployed. Each deployment component has an execution capacity, which is specified as the amount of resources which is available per accounting period; for simplicity, this accounting period is fixed in the semantics of Real-Time ABS and corresponds to the interval between integer values in the dense time domain of the language. The main block of a model executes in a root COG located on a default deployment component environment, with unrestricted processing capacity. To capture different deployment architectures, a model may create other deployment components with different resource capacities. When COGs are created, they are by default allocated to the same deployment component as their creator, but they may also be allocated to a different deployment component. Thus, in a model without explicit deployment components all objects execute in the default environment, which places no restrictions on the processing capacity of the model.

Deployment components are first-class citizens of Real-Time ABS. Syntactically, deployment components in Real-Time ABS are manipulated in a way similar to objects. Variables which refer to deployment components are typed by an interface DC. They may be passed around as arguments to method calls and they support a number of methods for load monitoring and load balancing purposes (further details may be found in [20]). Deployment components are dynamically created as instances of class DeploymentComponent, which implements DC. Deployment components may be created dynamically, depending on control flow, or statically in the main block of the model. Deployment components are created by the expression **new cog** DeploymentComponent(d,c). Here, the parameter c of type Resource specifies the initial CPU capacity of the deployment component. The parameter d of type String is a descriptor mainly used for monitoring purposes; i.e., it provides a user-defined name for the deployment component which facilitates querying the run-time state but that has no semantic effect. COGs are deployed on deployment components upon creation. By default a COG is deployed on the same deployment component as its creator. However, a different deployment component may be selected by means of an optional *deployment annotation* [DC: e] to the object creation statement, where e is an expression of type DC. Note that deployment annotations can only occur associated with the creation of COGs, not with objects.

The available resource capacity of a deployment component determines the amount of computation which may occur in the objects deployed on that deployment component. Objects allocated to the deployment component compete for the shared resources in order to execute, and they may execute until the deployment component runs out of resources or they are otherwise blocked. For the case of CPU resources, the resources of the deployment component define its capacity inside an accounting period, after which the resources are renewed.

5.2 Resource Consumption

The resource consumption of executing statements in the Real-Time ABS model is determined by a default cost value which can be set as a compiler option (e.g., −defaultcost=10); the default is for statements to have no execution cost. However, this default cost does not discriminate between statements, so a more refined cost model will often be desirable. For example, in a realistic model the assignment x=e should have a significantly higher cost for a complex expression e than for a constant. For this reason, more fine-grained costs can be inserted into Real-Time ABS models by means of adding a *cost annotation* [Cost: e] to any statement.

It is the responsibility of the modeler to specify appropriate resource costs. A behavioral model with default costs may be gradually transformed to provide more realistic resource-sensitive behavior by inserting such cost annotations. The manual estimation of resource cost is time consuming and error-prone. Therefore, it is desirable to have tool support for this activity. COSTABS [2] is an automated static analysis tool that is able to compute a worst-case approximation of the resource consumption of the non-virtualized programs, based on static analysis techniques.

However, the modeler may also want to capture *normative* constraints on resource consumption, such as resource limitations, at an abstract level; these can be made explicit in the model during the very early stages of the system design. To this end, cost annotations may be used by the modeler to abstractly represent the cost of some computation which is not fully specified.

6 Deployment Variability in ABS

This section lifts deployment to the level of feature models for SPLs. As deployment decisions may be introduced in the SPL in a number of ways, the main guideline for our approach is to keep a reasonable orthogonality between the functional variability and the deployment variability in the SPL model. Moreover, the separation of concerns between cost and capacity in the deployment models of ABS will be reflected in the feature models. Thus, deployment variability introduces two new variation points in the feature models of ABS:

- **Resource cost variability**: These features determine the choice of cost model for different parts of the model's logical artifacts, which determines how the resource cost is estimated during execution in the model; and
- **Deployment architecture variability**: These features determine how the logical artifacts of the model are mapped to a specific deployment architecture, which determines the execution capacity of the different locations on which the logical artefacts execute.

The resulting variability space is depicted in Figure 4. To keep the separation of concerns between the cost and capacity of resources, the features expressing resource cost and the features expressing deployment architectures are kept in different trees in the feature model expressed in ABS.

Resource cost model variability The basic feature in this tree is the *no cost* feature, typically selected for the functional analysis of the SPL model. In many cases, it is relevant to introduce fixed costs for selected jobs, similar to costs in a basic queuing network or simulation model (see, e.g., [17]). ABS allows data-sensitive costs to be expressed for selected jobs. The modeler may be interested in either measured, real cost for selected jobs, or in worst-case approximations (which may depend on data flow as well as control flow), both of which can be expressed via cost annotations.

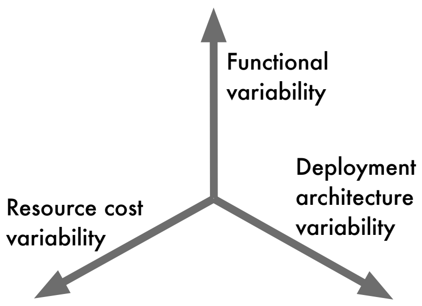
Functional variability
Deployment architecture variability
Resource cost variability

Figure 4: The SPL variability space with deployment variability.

Deployment architecture variability The basic feature in this tree is the *undeployed* feature which does not impose any capacity restrictions on the execution. This feature is typically selected for the functional analysis of the SPL model. In more refined variants, selected parts of the logical architecture can be deployed on deployment components with restricted capacity. For physical deployment, the deployment architecture is typically configured before the product execution starts. For virtualized deployment, the deployment architecture may evolve over time, modeling the startup and shutdown of virtual machines in a cloud computing scenario.

For both of these variation points, a refined feature model will typically allow to select or deselect resource-sensitive features for different parts of the model, expressed by the feature hierarchy and by cross-tree dependencies.

Example 4. We extend the feature model of the running example with a tree for resource costs, with root Resources, and another tree for deployment architecture, with root Deployments. Observe that this will increase the number of products in the SPL. The Resources root has the basic cost model NoCost, the model FixedCost to express that the CPU cost of executing a piece of code from the specific product (i.e., the cost of searching a word will depend on the specific searching algorithm) is data-independent and can be specified in the attribute cost. Furthermore, the feature WorstcaseCost selects the worst-case cost model in terms of the size of the input files, and MeasuredCost monitors the execution and reports the actual incurred cost during execution of the model. The Deployments root has three alternative features related to the number of available machines in the physical deployment architecture and the capacity of each of them specified by the attribute capacity.

```
root Resources {
  group oneof {
    NoCost,
    FixedCost { Int cost in [ 0 .. 10000 ] ; },
    WorstcaseCost,
    MeasuredCost
  }
}
root Deployments {
  group oneof {
    NoDeploymentScenario,
    UnlimitedMachines { Int capacity in [ 0 .. 10000 ] ; },
    LimitedMachines { Int capacity in [ 0 .. 10000 ] ;
      Int machinelimit in [ 0 .. 100 ] ; }
  }
}
```

```
// These definitions to be changed in delta modifications
type InKeyType = String; // filename
type InValueType = List<String>; // file contents
type OutKeyType = String; // word
type OutValueType = Int; // count

interface MapReduce {
  // invoked by client
  List<Pair<OutKeyType, List<OutValueType>>> mapReduce(
      List<Pair<InKeyType, InValueType>> documents);
  // invoked by workers
  Unit finished(Worker w);
}

interface IMap {
  // invoked by MapReduce component
  List<Pair<OutKeyType, OutValueType>> invokeMap(InKeyType key, InValueType value);
}

interface IReduce {
  // invoked by MapReduce component
  List<OutValueType> invokeReduce(OutKeyType key, List<OutValueType> value);
}

interface Worker extends IMap, IReduce { }
```

Figure 5: Interfaces of the base model of the MapReduce example in ABS.

7 Example: Variability in an SPL based on MapReduce

This section describes the implementation of a generic MapReduce framework in ABS and its adaptation to different products in the SPL described in Section 2. It will become apparent that a product that is implemented according to best practices for object-oriented software (i.e., decomposing functionality, methods implementing one task only, and the careful definition of datatypes) also makes the product well-suited as a base product for a software product line.

7.1 Commonalities in the ABS Base Product

Figure 5 shows the interfaces for the main MapReduce object and for the Worker objects which will carry out the computations in parallel. The computation is started by calling the mapReduce method with a list of *(key, value)* pairs. The main object will then create a number of worker objects, call invokeMap on these objects, gather and collate the results of the mapping phase, and then call invokeReduce on the workers and return the final results.

The base product in our example implements a word count function; it takes a list of files and their contents and returns a list of words and their total number of occurrences across all files. It does not implement resource or cost management. The MapReduce object reuses Worker objects from a pool and creates new workers when the pool is empty, as shown in Figure 6. Workers add themselves back to the pool by calling finished.

Figure 7 shows part of the worker implementation of the base product (i.e., a Wordcount product without any cost model). The invokeReduce method sets up the result, calls a private method reduce which emits intermediate results using the method emitReduceResult. The reduce method in Figure 7 is equivalent to the one shown in the original MapReduce paper [11]. The mapping functions of the worker objects are implemented in the same way.

7.2 Variability in the ABS Product Line

To change the functional feature of the model from computing word counts to computing word search, some parts of the model need to be altered via delta application. The same applies when

```
class MapReduce implements MapReduce {
  Int nWorkers = 0;
  List<Pair<OutKeyType, List<OutValueType>>>
      mapReduce(List<Pair<InKeyType, InValueType>> items) {
    ...
    while (~isEmpty(items)) {
      ...
      Worker w = this.getWorker();
      ...
    }
    ...
  }

  Worker getWorker() {
    Worker w = null;
    if (emptySet(workers)) {
      w = new cog Worker(this);
      nWorkers = nWorkers + 1;
    } else {
      w = take(workers);
      workers = remove(workers, w);
    }
    return w;
  }
  ...
}
```

Figure 6: Fragments of code in the MapReduce class from the base model of the MapReduce example in ABS.

varying the deployment and cost model, as explained in Section 6. These variation points in the SPL turn out to be orthogonal and can be modified independently of each other.

In the example, the methods to be modified by deltas are not public; i.e., they are not part of the published interface of the classes comprising the base model. This appears to be a recurring pattern: public methods like invokeReduce of Figure 7 interact with the outside world, gather and decompose data for computation and returning. If the modeler factors out computation into private methods with only one single task to perform (like reduce in Figure 7), these methods can be cleanly replaced in deltas, without imposing constraints on the implementation. This suggests that clean object-oriented code will in general be likely to be amenable to delta-oriented modification.

Functional Variability Figure 8 shows a delta fragment that modifies the functionality of the base model. Since the types of the input data and result can change, the base model uses the type synonyms InKeyType, InValueType, OutKeyType, OutValueType, which can be altered in a delta. This means that the signatures of the MapReduce class do not need to be adapted. To change the computation itself, one modifies the methods map and reduce of the Worker class. The implementer of the new map and reduce methods can use emitMapResult and emitReduceResult as in the base model, and does not need to care about invocation or return value handling protocols.

Resource Cost Variability Costs are incurred during (and because of) computational activity, hence the cost model seems to be linked to the computation. However, in MapReduce one can associate cost with two events: *invoking* a mapping or reduction step, and *producing* an intermediate result. Both of these events occur outside the map and reduce methods that implement the computation, therefore the cost model can be modified independently.

Figure 9 shows a delta implementing the FixedCost feature, which assigns a cost given as a feature attribute to each computation of an intermediate result; the feature attribute is passed in as a delta parameter. In general, costs are introduced by modifying the methods onMapStart and

```
class Worker(MapReduce master) implements Worker {
  List<OutValueType> reduceResults = Nil;
  ...
  List<OutValueType> invokeReduce(OutKeyType key, List<OutValueType> value) {
    reduceResults = Nil;
    this.onReduceStart(key, value);
    this.reduce(key, value);
    master!finished(this);
    List<OutValueType> result = reduceResults;
    reduceResults = Nil;
    return result;
  }

  Unit reduce(OutKeyType key, List<OutValueType> value) {
    List<Int> numlist = value;
    Int result = 0;
    while (~(numlist == Nil)) {
      result = result + head(numlist);
      numlist = tail(numlist);
    }
    this.emitReduceResult(result);
  }

  Unit emitReduceResult(OutValueType value) {
    this.onReduceEmit(value);
    reduceResults = Cons(value, reduceResults);
  }

  Unit onReduceEmit(OutValueType value) { skip; }
}
```

Figure 7: Fragments of code in the Worker class from the base model of the MapReduce example in ABS: implementing the reduce part of the Wordcount example.

onReduceStart for assigning costs to starting a computation step, and by modifying onMapEmit and onReduceEmit for assigning costs to the production of a result.

Deployment Architecture Variability Deployment architecture, i.e., decisions on how many workers to create and how many resources to supply them with, is implemented in the method getWorker of the MapReduce class. Figure 6 in Section 7.1 shows an implementation that simply creates any worker objects needed, in an environment disregarding any resource constraints. To change this behavior, the modeler implements a delta that overrides getWorker (and also the method finished should the new getWorker method not use the resource pool of the base model); an example can be seen in Figure 10.

This delta changes the deployment model as follows:

- Workers execute in deployment components with a specified processing capacity capacity.
- The system will use a maximum number of workers maxWorkers.

The Product Line Configuration The feature model presented in Section 6 extends the SPL of Section 2 with resource cost variability, resulting in an SPL with 14 different products. Figure 11 shows part of the product line configuration for the SPL and Figure 12 shows the specification of some of the derivable products.

7.3 Results

In the deployment components of the deployment architecture features, capacity is defined by the amount of resource costs that can be processed per accounting period (in terms of the dense time semantics of execution in Real-Time ABS). When the base model is extended with features for

```
delta DOccurrences;
uses MapReduce;
modifies type OutValueType = String;
modifies class Worker {
  modifies Unit map(InKeyType key, InValueType value) {
    ...
  }
  modifies Unit reduce(OutKeyType key, List<OutValueType> value) {
    ...
  }
}
```

Figure 8: Modifying the computation performed by MapReduce

```
delta DFixedCost (Int cost);
uses MapReduce;
modifies class Worker {
  modifies Unit onMapEmit(OutKeyType key, OutValueType value) {
    [Cost: cost] skip;
  }
  modifies Unit onReduceEmit(OutValueType value) {
    [Cost: cost] skip;
  }
}
```

Figure 9: Modifying the cost model of the base product

```
delta DBoundedDeployment (Int capacity, Int maxWorkers);
uses MapReduce;
modifies class MapReduce {
  modifies Worker getWorker() {
    if (emptySet(workers) && nWorkers < maxWorkers) {
      DeploymentComponent dc = new DeploymentComponent("worker " + toString(nWorkers + 1),
          CPU(capacity));
      [DC: dc] Worker w = new cog Worker(this);
      workers = insertElement(workers, w);
      nWorkers = nWorkers + 1;
    }
    await ~(emptySet(workers));
    Worker w = take(workers);
    workers = remove(workers, w);
    return w;
  }
}
```

Figure 10: Modifying the deployment model of the MapReduce example

deployment architecture and resource cost, the *load* on the individual deployment components, defined as the actual incurred cost per accounting period, can be recorded and visualized.

We illustrate how deployment variability for products can be validated using the simulation tool of ABS, by comparing the performance of two different deployments of the Wordcount product, varying the number of available machines between 5 (the “Demo” version) and 20 (the “Full” version), but keeping the cost model, input data and computation model constant. The graphs in Figure 13 shows the total load of all machines over simulated time for the two products. The figure shows two typical instances of a typical MapReduce workload; first, the map processes execute until they are finished, then the reduce processes execute. The start of the reduce phase can be observed in the graph of Figure 13 as the second spike in processing activity. It can be seen that the demo version takes over twice as much simulated time to complete its execution, while the full version completes its execution earlier by incurring a load that is higher than for the demo version (while still decreasing as the map processes terminate).

Similar qualitative investigations can be performed regarding the influence of varying cost

```
productline MapReduceSPL;

features
  Wordcount, Wordsearch;                                          // Functional features
  NoCost, FixedCost, WorstCaseCost, MeasuredCost,                  // Resource cost features
  NoDeploymentScenario, UnlimitedMachines, LimitedMachines,        // Deployment architecture features

delta DOccurrences when Wordsearch;
delta DFixedCost(Cost.cost) when Cost;
delta DUnboundedDeployment(UnlimitedMachines.capacity) when UnlimitedMachines;
delta DBoundedDeployment(LimitedMachines.capacity, LimitedMachines.machinelimit)
      when LimitedMachines;
...
```

Figure 11: Product line configuration for the MapReduce example in ABS.

```
product WordcountModel (Wordcount, NoCost, NoDeploymentScenario);
product WordcountFull (Wordcount, Cost{cost=10}, UnlimitedMachines{capacity=20});
product WordcountDemo (Wordcount, Cost{cost=10}, LimitedMachines{capacity=20, machinelimit=2});
product WordsearchModel (Wordsearch, NoCost, NoDeploymentScenario);
product WordsearchFull (Wordsearch, Cost{cost=10}, UnlimitedMachines{capacity=20});
product WordsearchDemo (Wordsearch, Cost{cost=10}, LimitedMachines{capacity=20, machinelimit=2});
...
```

Figure 12: Specific Products for the MapReduce example in ABS.

models (e.g., worst-case vs. average cost) and more involved deployment strategies.

8 Related Work

The concurrency model of ABS is akin to concurrent objects and Actor-based computation, in which software units with encapsulated processors communicate asynchronously [4, 15, 19, 39]. Their inherent compositionality allows concurrent objects to be naturally distributed on different locations, because only an object's local state is needed to execute its methods. In previous work on deployment architecture [3, 20–22], the authors have introduced *deployment components* as a modeling concept for deployment architectures, which captures restricted resources shared between a group of concurrent objects, and shown how components with parametric resources may be used to capture a model's behavior for different assumptions about the available resources. The formal details of this approach are given in [21]; two larger case studies on virtualized systems deployed on the cloud are presented in [10, 23]. Whereas the focus of this paper has been on processing resources, deployment components with restricted memory has been addressed in [3] and more abstract approaches to user-defined resource management and user-defined cost annotations are discussed in [21, 22] respectively. Our approach to deployment modeling would be a natural fit for resource-sensitive deployment in other Actor-based approaches.

Software diversity has recently been surveyed by Schaefer et al. [35], but deployment variability is not considered. However, deployment variability has recently been studied in the context of feature models. For example, Garcia et al. [12] describe a feature model that captures the architectural and technological variability of multilayer applications and present a model driven development process which uses this feature model at its core. In contrast our paper considers a much simpler feature model, but it is integrated in a full SPL framework and explicitly linked to executable models which can be compared by tool-based analysis. Without considering variability, Wagelaar et al. [38] introduce a platform ontology and modeling framework based on description logic, which can be used to automatically configure various reusable concrete platforms that can be later be integrated with a platform-independent model using the Model Driven

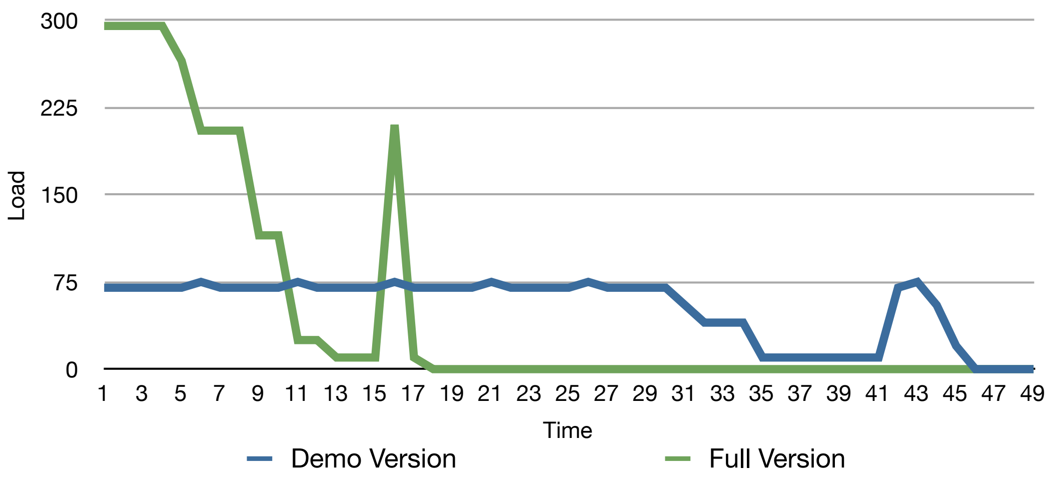

Figure 13: Simulation results for varying deployment models with constant cost + functional model

Architecture approach. We follow a similar approach based on the extending a purely functional model with deployment features, but our framework is based on simpler concepts which does not introduce the overhead of description logic. In the context of QoS variability, Kattepur et al. [24] study a modeling and analysis framework for testing the QoS of an orchestration before deployment to determine realistic Service Level Agreement contracts; their analysis uses probabilistic model of QoS. Our work similarly allows the model-based comparison of QoS variability, but focuses on deployment architecture and processing capacity rather than orchestration.

The MapReduce programming pattern which is the basis for the example of this paper, has been formalized and studied from different perspectives. Yang et al. [40] develop a CSP model of MapReduce, with a focus on the correctness of the communication between the processes. Lämmel [25] develops a rigorous description of MapReduce using Haskell, resulting in an executable specification of MapReduce. Ono et al. [28] formalize an abstract model of MapReduce using the proof assistant Coq, and use this formalization to verify JML annotations of MapReduce applications. However, none of these works focus on deployment strategies or relate MapReduce to deployment variability in SPLs

9 Conclusion

Software today is increasingly often developed as a range of products for embedded devices with restricted resource capacity or for virtualized utility computing. For a software product line (SPL) targeting such platforms, the deployment of different products in the range should also be considered as a variation point in the SPL.

In this paper, we have integrated an approach to describing explicit deployment scenarios by resource restricted deployment components into a formal modeling language for SPL engineering. This integration is based on delta models to systematize the derivation of product variants, and demonstrated in the ABS modeling language. The integration proposed in this paper emphasizes orthogonality between functional features, resource cost features, and deployment architecture features, to facilitate finding the best match between functional features and a target deployment architecture for a specific product. The analysis supported by ABS allows the validation of deployment decisions for specific products in the SPL, which may entail a refinement of the feature model. Resource cost variability can be exploited in this context to compare product performance under different cost models such as fixed cost, measured simulation cost, and worst-case cost.

The approach has been demonstrated on an example of a SPL using the highly parallelizable MapReduce programming pattern as its common base product, and used to compare the performance of full versions to restricted demo versions of products. A restriction of the current work is the concrete semantics used for simulation, which necessitates a per-product trial and failure approach to validation. An interesting extension of the work presented in this paper is to use a symbolic semantics and apply symbolic execution techniques to analyze the deployment sensitive SPL models. This could allow the analysis to be lifted from concrete deployment scenarios for specific products to a more generalized analysis.

References

[1] G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems.* The MIT Press, Cambridge, Mass., 1986.

[2] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. COSTABS: a cost and termination analyzer for ABS. In O. Kiselyov and S. Thompson, editors, *Proc. Workshop on Partial Evaluation and Program Manipulation (PEPM'12)*, pages 151–154. ACM, 2012.

[3] E. Albert, S. Genaim, M. Gómez-Zamalloa, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Simulating concurrent behaviors with worst-case cost bounds. In M. Butler and W. Schulte, editors, *FM 2011*, volume 6664 of *Lecture Notes in Computer Science*, pages 353–368. Springer, June 2011.

[4] J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.

[5] J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 2012. Available online: `http://dx.doi.org/10.1007/s11334-012-0184-5`. To appear.

[6] D. Caromel and L. Henrio. *A Theory of Distributed Object.* Springer, 2005.

[7] D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Y. H. Wong. Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In M. Bernardo and V. Issarny, editors, *Proc. 11th Intl. School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM 2011)*, volume 6659 of *Lecture Notes in Computer Science*, pages 417–457. Springer, 2011.

[8] D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. In E. Visser and J. Järvi, editors, *Proc. Ninth International Conference on Generative Programming and Component Engineering, (GPCE'10)*, pages 13–22. ACM, 2010.

[9] D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte. Variability modelling in the abs language. In B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th Intl. Symposium on Formal Methods for Components and Objects (FMCO'10)*, volume 6957 of *Lecture Notes in Computer Science*, pages 204–224. Springer, 2012.

[10] F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, and P. Y. H. Wong. Formal modeling of resource management for cloud architectures: An industrial case study. In F. D. Paoli, E. Pimentel, and G. Zavattaro, editors, *Proc. European Conference on Service-Oriented and Cloud Computing (ESOCC 2012)*, volume 7592 of *Lecture Notes in Computer Science*, pages 91–106. Springer, Sept. 2012.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI'04)*, pages 137–150. USENIX Association, 2004.

[12] J. Garcia-Alonso, J. B. Olmeda, and J. M. Murillo. Architectural variability management in multi-layer web applications through feature models. In *Proc. 4th Intl. Workshop on Feature-Oriented Software Development (FOSD'12)*, pages 29–36. ACM, 2012.

[13] H. Gomaa. *Designing Software Product Lines with UML.* Professional. Addison-Wesley, 2005.

[14] A. Haber, T. Kutz, H. Rendel, B. Rumpe, and I. Schaefer. Delta-oriented architectural variability using monticore. In *5th European Conference on Software Architecture (ECSA'11), Companion Volume*, page 6. ACM, 2011. Workshop on Software Architecture Variability (SAVA).

[15] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.

[16] S. A. Hendrickson and A. van der Hoek. Modeling product line architectures through change sets and relationships. In *Proceedings of the 29th international conference on Software Engineering (ICSE'07)*, pages 189–198. IEEE Computer Society, 2007.

[17] R. Jain. *The Art of Computer Systems Performance Analysis.* Wiley, 1991.

[18] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.

[19] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[20] E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Dynamic resource reallocation between deployment components. In J. S. Dong and H. Zhu, editors, *Proc. International Conference on Formal Engineering Methods (ICFEM'10)*, volume 6447 of *Lecture Notes in Computer Science*, pages 646–661. Springer, Nov. 2010.

[21] E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In B. Beckert and C. Marché, editors, *Proc. International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'10)*, volume 6528 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2011.

[22] E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. A formal model of user-defined resources in resource-restricted deployment scenarios. In B. Beckert, F. Damiani, and D. Gurov,

editors, *FoVeOOS'11)*, volume 7421 of *Lecture Notes in Computer Science*, pages 196–213. Springer, 2012.

[23] E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Modeling resource-aware virtualized applications for the cloud in Real-Time ABS. In T. Aoki and K. Tagushi, editors, *Proc. 14th International Conference on Formal Engineering Methods (ICFEM'12)*, volume 7635 of *Lecture Notes in Computer Science*, pages 71–86. Springer, Nov. 2012.

[24] A. Kattepur, S. Sen, B. Baudry, A. Benveniste, and C. Jard. Variability modeling and qos analysis of web services orchestrations. *2012 IEEE 19th International Conference on Web Services*, pages 99–106, 2010.

[25] R. Lämmel. Google's MapReduce programming model - revisited. *Sci. Comput. Program.*, 70(1):1–30, 2008.

[26] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.

[27] N. Loughran, P. Sánchez, A. Garcia, and L. Fuentes. Language support for managing variability in architectural models. In C. Pautasso and É. Tanter, editors, *7th International Symposium on Software Composition (SC'08)*, volume 4954 of *Lecture Notes in Computer Science*, pages 36–51. Springer, 2008.

[28] K. Ono, Y. Hirai, Y. Tanabe, N. Noda, and M. Hagiya. Using Coq in specification and program extraction of Hadoop MapReduce applications. In G. Barthe, A. Pardo, and G. Schneider, editors, *9th International Conference on Software Engineering and Formal Methods (SEFM'11)*, volume 7041 of *Lecture Notes in Computer Science*, pages 350–365. Springer, 2011.

[29] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[30] I. Schaefer. Variability modelling for model-driven development of software product lines. In D. Benavides, D. S. Batory, and P. Grünbacher, editors, *Proc. Fourth International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'10)*, volume 37 of *ICB-Research Report*, pages 85–92. Universität Duisburg-Essen, 2010.

[31] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented Programming of Software Product Lines. In J. Bosch and J. Lee, editors, *Proc. 14th International Conference on Software Product Lines (SPLC'10)*, volume 6287 of *Lecture Notes in Computer Science*, pages 77–91, Jeju, South Korea, 2010. Springer.

[32] I. Schaefer, L. Bettini, and F. Damiani. Compositional type-checking for delta-oriented programming. In P. Borba and S. Chiba, editors, *Proc. 10th Intl. Conf. on Aspect-Oriented Software Development (AOSD'11)*, pages 43–56. ACM, 2011.

[33] I. Schaefer and F. Damiani. Pure delta-oriented programming. In S. Apel, D. S. Batory, K. Czarnecki, F. Heidenreich, C. Kästner, and O. Nierstrasz, editors, *Proc. 2nd Intl. Workshop on Feature-Oriented Software Development (FOSD'10)*, pages 49–56. ACM, 2010.

[34] I. Schaefer and R. Hähnle. Formal methods in software product line engineering. *IEEE Computer*, 44(2):82–85, 2011.

[35] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer (STTT)*, 14(5):477–495, 2012.

[36] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *International Conference on Requirements Engineering (RE'06)*, pages 136–145. IEEE Computer Society, 2006.

[37] R. C. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, 2000.

[38] D. Wagelaar and V. Jonckers. Explicit platform models for MDA. In *MoDELS'05*, pages 367–381, 2005.

[39] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. Object oriented programming, systems, languages, and applications (OOPSLA'05)*, pages 439–453, New York, NY, USA, 2005. ACM Press.

[40] F. Yang, W. Su, H. Zhu, and Q. Li. Formalizing mapreduce with csp. *Engineering of Computer-Based Systems, IEEE International Conference*, pages 358–367, 2010.