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 increasingly emphasizes variability by developing families of products for a range of application contexts or user requirements. 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. ABS also supports deployment models, with a separation of concerns between execution cost and server capacity. This allows the model-based assessment of deployment choices on a product's quality of service. This paper combines deployment models with the variability concepts of ABS, to model deployment choices as features when designing a family of products.

1 Introduction

Variability is prevalent in modern software in order to satisfy a range of application contexts or user requirements [32]. A software product line (SPL) realizes this variability through a family of product variants (e.g., [27]). A specific product is obtained by selecting features from a feature model [34]; these models typically focus on the functionality and software quality attributes of different features and products. To express variability in system *design*, features typically take the form of architectural models, behavioral models, and test suites [33]. Architectural variability focuses on the presence of component variants, and can be described using, e.g., the Variability Modeling Language [25], UML stereotypes [13], or (hierarchical) component models such as Koala [35]. In Delta modeling [9, 28, 29], a set of deltas specifies modifications to a core product. Δ-MontiArch applies delta modeling to architectural description [14]; a delta can add or remove components, ports, and connections between components.

Whereas architectural models describe 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*. To describe deployment architectures we use a separation of concerns between the *application model*, which requires resources, and the *deployment scenario*, which reflects the computing environment and provides heterogeneous amounts of resources.

* Partly funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu).

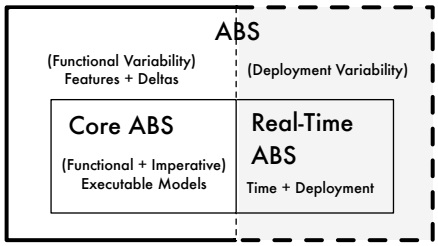

Fig. 1. ABS language extension.

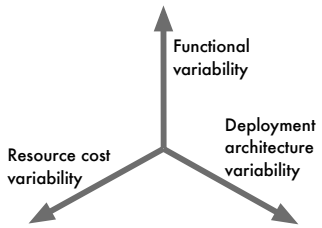

Fig. 2. The SPL variability space with deployment variability.

This paper integrates deployment variability in SPL models such that different targeted deployment architectures may be taken into account early in the design of the SPL. We aim at a reasonable orthogonality between functional and deployment variability in the SPL model. The starting point for this work is the **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 be easy to use for software developers; it is *executable* to support 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 [8, 10], and (2) ABS supports the modeling of deployment decisions based on the modeling concept of *deployment components* [21] in Real-Time ABS [6]. Real-Time ABS leverages resources and their dynamic management to the abstraction level of software models. Fig. 1 shows how functional variability modeling in ABS and time and deployment models in Real-Time ABS both extend Core ABS. Although these extensions of ABS coexist, they 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 Fig. 1.

Our approach to deployment variability for SPL models makes a separation of concerns between cost and capacity which introduces two new variation points in the variability space of ABS feature models (depicted in Fig. 2):

- **Resource cost variability**: These features determine the costs associated with the SPL's logical artifacts; and
- **Deployment architecture variability**: These features determine how the logical artifacts are deployed on locations with different capacities.

The main contributions of the paper are:

- an integration of delta models with deployment components in ABS;
- this integration allows orthogonality between functional and deployment variability, such that features expressing resource cost and deployment architectures are kept in different trees in the ABS feature models;
- 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 SPLs.

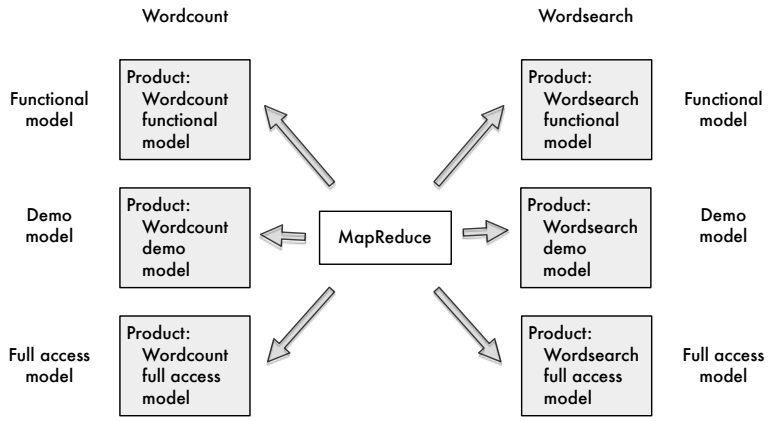

Fig. 3. A family of products sharing an underlying MapReduce structure.

Paper overview. Sect. 2 motivates our work by an example of deployment variability. Sect. 3 presents modeling in the abstract behavioral specification language ABS and Sect. 4 delta modeling and its realization in ABS. Sect. 5 combines delta-oriented variability with deployment modeling, and discusses how to extend a feature model with deployment variability. Sect. 6 revisits the example, Sect. 7 discusses related work, and Sect. 8 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 parallelizable jobs on distinct subsets of 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 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 SPLs.

Our example uses MapReduce to model an SPL with a range of services which inspect a set of documents. Individual products may implement, e.g., Wordcount, which counts the occurrences of words in the given documents, and Wordsearch, which searches for documents in which a given word occurs. For simplicity, we assume that a service either provides the Wordcount or the Wordsearch feature. The services are implemented on a cluster of computers, using MapReduce.

To attract clients to the word count and word search services, freely available demo versions offer the same functionality as the full versions, albeit with a lower quality of service. When the services are deployed, the demo versions will run on a few machines, whereas the full versions have access to the full power of the cluster. Our model has 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. This SPL (see Fig. 3) is a running example in the paper.

3 Behavioral and Deployment Modeling in ABS

The abstract behavioral specification language ABS targets the executable design of distributed object-oriented systems. It has a formally defined kernel called Core ABS [18]. ABS is based on concurrent object groups (COGs), akin to concurrent objects [7,19], Actors [1], and Erlang processes [5]. COGs support interleaved concurrency based on guarded commands. ABS has a functional and an imperative layer, combined with a Java-like syntax. Real-Time ABS [6] extends Core ABS models with (dense) time; in this paper we do not specify execution time directly but rather *observe* time by measurements of the executing model.

ABS has a *functional layer* with 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. The modeler can define additional types to succinctly express data structures of the problem domain.

The *imperative layer* of ABS describes side-effectful computation, concurrency, communication and synchronization. ABS objects are *active* in the sense that their run method, if defined, gets called upon creation. Communication and synchronization are decoupled: Communication is based on asynchronous method calls. After executing f=o!m(e), which assigns the call to a *future variable* f, the caller proceeds execution *without blocking* while m(e) executes in the context of o. 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. Scheduling is cooperative via **await** g statements, which suspend the current process until g (a condition over object or future variable state) becomes true. The remaining statements of ABS (assignment, object creation, conditionals and loops) are designed to be familiar to a Java programmer.

Deployment Modeling. One purpose of describing deployment in a modeling language is to differentiate execution time based on *where* the execution takes place, i.e., the model should express how the execution time varies with the available *capacity* of the chosen deployment architecture. For this purpose, Real-Time ABS extends Core ABS with primitives to describe *deployment architectures* which express how distributed systems are mapped on physical and/or virtual media with many locations. Real-Time ABS lifts deployment architectures to the abstraction level of the modeling language, where the physical or virtual media are represented by *deployment components* [20].

A *deployment component* is part of the model's deployment architecture, on which a number of COGs are deployed. Deployment components are first-class citizens and they support a number of methods for load monitoring and load balancing purposes (cf. [20]). Each deployment component has an *execution ca-*

pacity, which is the amount of resources available per accounting period. By default, all objects execute in a default (root) environment with unrestricted capacity. Other deployment components with restricted capacities may be created to capture different deployment architectures. COGs are created on the same deployment component as their creator by default; a different deployment component may be selected by an optional *deployment annotation* [DC: dc] to object creation, for a deployment component dc.

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.

The resource consumption of executing statements in the Real-Time ABS model is expressed 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 may be gradually transformed to provide more realistic resource-sensitive behavior by inserting more fine-grained cost annotations. The automated static analysis tool COSTABS [2] can compute a worst-case approximation of resource consumption, 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 in the model.

4 Delta-Oriented Variability in ABS

This section describes how 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 [8, 10]. Fig. 4 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 can also express other

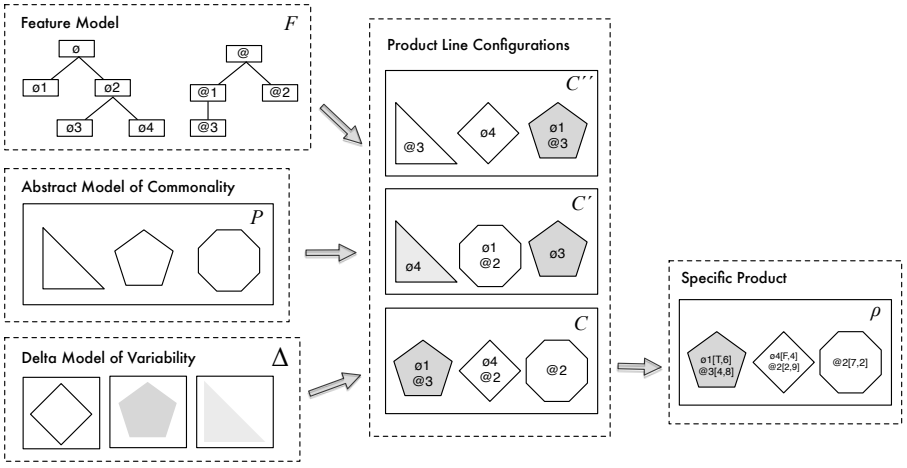

Fig. 4. A graphical representation of a Delta-Oriented variability model.

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 [8, 10].

Example 1. 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. [29–31] as a modeling and programming language approach for SPLs. 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 Fig. 4), 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 Variability in ABS

Feature models usually describe functional variability in a software product line. This section discusses lifting deployment variability to ABS feature models and its interaction with functional variability. Our approach aims to establish orthogonality between the functional and deployment aspects in the SPL model in order to maintain multiple axes of variability (see Fig. 2). The further separation of concerns between cost and capacity in the deployment models of ABS is reflected in the feature models as well.

Thus, variability in a deployment-aware SPL comprises these variation points in the feature models:

Functional variability: These features determine the functional behavior of a product and are used as in standard SPL engineering.

Resource cost variability: These features describe the choice of how the incurred resource cost is estimated during execution of the model. The basic feature is the *no cost* feature, typically selected for functional analysis of the SPL model. Other cost models are fixed-cost for selected jobs (similar to costs in a basic queuing network or simulation model; see, e.g., [17]), and data-sensitive costs. These can be either measured, real cost for selected jobs or worst-case approximations (which may depend on data flow as well as control flow). All of these can be expressed via cost annotations.

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 basic feature is the *undeployed* feature which does not impose any capacity restrictions on the execution. This feature is typically selected together with *no cost* during functional analysis and testing. When analyzing non-functional properties, features describe how selected parts of the logical architecture are deployed on deployment components with restricted capacity, either statically or (for virtualized deployment) dynamically.

Example 2. We extend the feature model of Example 1 with a Resources tree for resource costs, and a Deployments tree for deployment architecture. The Resources root has the basic feature NoCost, the feature FixedCost for a basic data-independent cost model specified in the attribute cost, the feature WorstcaseCost for a worst-case cost model in terms of the size of the input files, and MeasuredCost for using the actual incurred cost measured during execution of the model. The Deployments root has three alternative features related to the number of available machines in the physical deployment architecture; the capacity of each machine is 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 {
  List<Pair<OutKeyType, List<OutValueType>>>
      mapReduce(List<Pair<InKeyType, InValueType>> docs); // invoked by client
  Unit finished(Worker w); // invoked by workers when finished with 1 task
}

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

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

interface Worker extends IMap, IReduce { }
```

Fig. 5. Interfaces of the base model of the MapReduce example in ABS.

6 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.

6.1 Commonalities in the ABS Base Product

Fig. 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, call invokeReduce on the workers and collate and return the final result.

The base product in our example implements a word count function (computing word occurrences over a list of files), without a resource or deployment model. Worker objects are reused from a pool, but there is no bound on the number of workers created. Workers add themselves back to the pool by calling finished.

Figure 6 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 Fig. 6 is equivalent to the

```
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);
    List<OutValueType> result = reduceResults;
    reduceResults = Nil;
    master!finished(this);
    return result;
  }

  Unit emitReduceResult(OutValueType value) {
    this.onReduceEmit(value); // variation point for cost model
    reduceResults = Cons(value, reduceResults);
  }

  // variation point for functional model
  Unit reduce(OutKeyType key, List<OutValueType> value) {
    OutValueType result = 0;
    ... // sum up value list into result variable ...
    this.emitReduceResult(result);
  }

  // variation point for cost model
  Unit onReduceStart(OutValueType value) { skip; }
  Unit onReduceEmit(OutValueType value) { skip; }
}
```

Fig. 6. The reduce part of the Wordcount example in the Worker class.

one shown in the original MapReduce paper [11]. The mapping functions of the worker objects are implemented in the same way.

6.2 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 varying the deployment and cost model, as explained in Section 5. 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 Fig. 6 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 Fig. 6), 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. The following delta shows a delta fragment that modifies the functionality of the base model:

```
delta DOccurrences;
uses MapReduce;
modifies type OutValueType = String; // Change the method signatures
modifies class Worker {
  modifies Unit map(InKeyType key, InValueType value) {
    ... // change non−public map method to compute occurrences
  }
  modifies Unit reduce(OutKeyType key, List<OutValueType> value) {
    ... // change non−public reduce method to compute occurrences
  }
}
```

By modifying the type synonyms InKeyType, InValueType, OutKeyType and OutValueType from the base model, we can change the data types and method signatures of the model without having to change any code in the MapReduce class. Modifying the methods map and reduce of the Worker class changes the computation performed by the product. The new map and reduce methods use emitMapResult and emitReduceResult as in the base model; hence they do not need to care about invocation or return value handling protocols.

Resource cost variability. Costs are incurred during (and because of) computational activity. This means that cost model and functional model are related. However, the two aspects can be decoupled by associating costs with events, for example in MapReduce *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:

```
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;
  }
}
```

This FixedCost delta assigns a cost (given as a delta attribute) to each computation of an intermediate result; the feature attribute is passed in as a delta parameter. Starting a mapping step carries no cost, hence the onMapStart method is not modified. In general, costs are introduced into MapReduce by modifying the methods onMapStart and onReduceStart for assigning costs to starting a computation step, and by modifying onMapEmit and onReduceEmit for assigning costs to the production of a result. Figure 6 shows where these methods are invoked.

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 MapReduce class. As mentioned, this class manages a pool of Worker instances which is by default of unbounded size. To change this behavior, the modeler implements a delta that overrides a method getWorker

```
productline MapReduceSPL;

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

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

Fig. 7. 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});
```

Fig. 8. Specifying Products for the MapReduce example in ABS.

(and also the method finished of the MapReduce implementation in case the new getWorker method does not use the resource pool of the base model). The capacity and number of deployment components can be adjusted via delta parameters:

```
delta DBoundedDeployment (Int capacity, Int maxWorkers);
uses MapReduce;
modifies class MapReduce {
  ... // adjust behavior of resource pool and capacities of created deployment components
}
```

The product line configuration. The feature model presented in Section 5 extends the SPL of Section 2 with resource cost variability, resulting in an SPL with 14 different products. Fig. 7 shows part of the product line configuration for the SPL and Fig. 8 shows the specification of some of the derivable products.

6.3 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 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

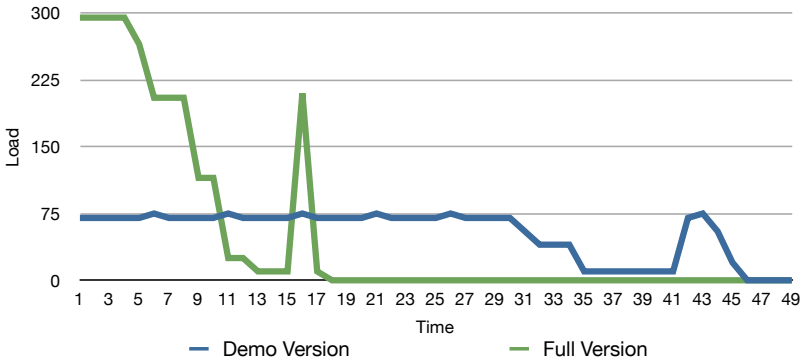

Fig. 9. Varying deployment model, constant cost and functional model

between 5 (the "Demo" version) and 20 (the "Full" version), but keeping the cost model, input data and computation model constant. The graphs in Fig. 9 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 Fig. 9 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 models (e.g., worst-case vs. average cost) and more involved deployment strategies.

7 Related Work

The inherent compositionality of the concurrency model considered in this paper allows objects to be naturally distributed on different locations, because only an object's local state is needed to execute its methods. In previous work [4,20,21], the authors have introduced *deployment components* as a modeling concept to 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 [3,22]. Our approach to deployment modeling would be a natural fit for resource-sensitive deployment in other Actor-based approaches, e.g., [5,15].

Deployment variability is not considered in the recent software diversity survey [33], but it has been studied in the context of feature models. For example, a feature model that captures the architectural and technological variability of

multilayer applications is described in [12] together with an associated model-driven development process. 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, a platform ontology and modeling framework based on description logic is proposed by [36], 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 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, [23] 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. [37] develop a CSP model of MapReduce, with a focus on the correctness of the communication between the processes. [24] develops a rigorous description of MapReduce using Haskell, resulting in an executable specification of MapReduce. [26] formalizes 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.

8 Conclusion

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

This paper integrates explicit resource restricted deployment scenarios 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 proposed integration 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 supported analysis 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 to compare product performance under different cost models such as fixed cost, measured simulation cost, and worst-case cost.

The approach is demonstrated on an SPL using the 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 presented work is the concrete semantics, which necessitates a per-product trial and failure approach to validation. An interesting extension of our work 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, 1986.
2. E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. COSTABS: a cost and termination analyzer for ABS. In *PEPM'12*, pages 151–154. ACM, 2012.
3. E. Albert, F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. L. T. Tarifa, and P. Y. H. Wong. Formal modeling and analysis of resource management for cloud architectures. an industrial case study using Real-Time ABS. *J. of Service-Oriented Computing and Applications*, 2014. To appear.
4. 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 *FM 2011*, *LNCS* 6664, pages 353–368. Springer, June 2011.
5. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
6. 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*, 9(1):29–43, 2013.
7. D. Caromel and L. Henrio. *A Theory of Distributed Object*. Springer, 2005.
8. 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 *SFM'11*, *LNCS* 6659, pages 417–457. Springer, 2011.
9. D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. In *GPCE'10*, pages 13–22. ACM, 2010.
10. D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte. Variability modelling in the ABS language. In *FMCO'10*, *LNCS* 6957, pages 204–224. Springer, 2012.
11. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04*, pages 137–150. USENIX, 2004.
12. J. Garcia-Alonso, J. B. Olmeda, and J. M. Murillo. Architectural variability management in multi-layer web applications through feature models. In *FOSD'12*, pages 29–36. ACM, 2012.
13. H. Gomaa. *Designing Software Product Lines with UML*. Addison-Wesley, 2005.
14. A. Haber, T. Kutz, H. Rendel, B. Rumpe, and I. Schaefer. Delta-oriented architectural variability using Monticore. In *Software Architecture (ECSA'11), Companion Vol.*, 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 *ICSE'07*, pages 189–198. IEEE, 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 *FMCO'10*, *LNCS* 6957, 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 *ICFEM'10*, *LNCS* 6447, 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 *FoVeOOS'10*, *LNCS* 6528, pages 46–60. Springer, 2011.
22. E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Modeling resource-aware virtualized applications for the cloud in Real-Time ABS. In *ICFEM'12*, *LNCS* 7635, pages 71–86. Springer, Nov. 2012.
23. A. Kattepur, S. Sen, B. Baudry, A. Benveniste, and C. Jard. Variability modeling and QoS analysis of web services orchestrations. *ICWS*, pages 99–106. IEEE 2010.
24. R. Lämmel. Google's MapReduce programming model - revisited. *Sci. Comput. Program.*, 70(1):1–30, 2008.
25. N. Loughran, P. Sánchez, A. Garcia, and L. Fuentes. Language support for managing variability in architectural models. In *Software Composition (SC'08)*, *LNCS* 4954, pages 36–51. Springer, 2008.
26. K. Ono, Y. Hirai, Y. Tanabe, N. Noda, and M. Hagiya. Using Coq in specification and program extraction of Hadoop MapReduce applications. In *SEFM'11*, *LNCS* 7041, pages 350–365. Springer, 2011.
27. K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, 2005.
28. I. Schaefer. Variability modelling for model-driven development of software product lines. In *VaMoS'10*, *ICB-Res. Rep.* 37, pages 85–92. Univ. Duisburg-Essen, 2010.
29. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented Programming of Software Product Lines. In *SPLC'10*, *LNCS* 6287, pages 77–91. Springer, 2010.
30. I. Schaefer, L. Bettini, and F. Damiani. Compositional type-checking for delta-oriented programming. In *AOSD'11*, pages 43–56. ACM, 2011.
31. I. Schaefer and F. Damiani. Pure delta-oriented programming. In *FOSD'10*, pages 49–56. ACM, 2010.
32. I. Schaefer and R. Hähnle. Formal methods in software product line engineering. *IEEE Computer*, 44(2):82–85, 2011.
33. 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. *Software Tools for Technology Transfer (STTT)*, 14(5):477–495, 2012.
34. P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *Intl. Conf. on Requirements Engineering (RE'06)*, pages 136–145. IEEE, 2006.
35. 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.
36. D. Wagelaar and V. Jonckers. Explicit platform models for MDA. In *MoDELS'05*, pages 367–381, 2005.
37. F. Yang, W. Su, H. Zhu, and Q. Li. Formalizing MapReduce with CSP. *Intl. Conf. on Engineering of Computer-Based Systems*, pages 358–367. IEEE 2010.