HATS Abstract Behavioral Specification: The Architectural View[*]

Reiner Hähnle[1], Michiel Helvensteijn[2], Einar Broch Johnsen[3],
Michael Lienhardt[4], Davide Sangiorgi[4], Ina Schaefer[5], and Peter Y. H. Wong[6]

[1] Dept. of Computer Science, TU Darmstadt, haehnle@cs.tu-darmstadt.de
[2] CWI Amsterdam, Michiel.Helvensteijn@cwi.nl
[3] Dept. of Informatics, Univ. of Oslo, einarj@ifi.uio.no
[4] Dept. of Computer Science, Univ. of Bologna,
{Davide.Sangiorgi,lienhard}@cs.unibo.it
[5] Dept. of Computer Science, TU Braunschweig, i.schaefer@tu-braunschweig.de
[6] Fredhopper B.V, Amsterdam, peter.wong@fredhopper.com

Abstract. The Abstract Behavioral Specification (ABS) language is a formal, executable, object-oriented, concurrent modeling language intended for behavioral modeling of complex software systems that exhibit a high degree of variation, such as software product lines. We give an overview of the architectural aspects of ABS: a feature-driven development workflow, a formal notion of deployment components for specifying environmental constraints, and a dynamic component model that is integrated into the language. We employ an industrial case study to demonstrate how the various aspects work together in practice.

1 Introduction

This is the third in a series of reports which together give a comprehensive overview of the possibilities and use cases of the Abstract Behavioral Specification (ABS) language developed within the FP7 EU project HATS (for "Highly Adaptable and Trustworthy Software using Formal Models"). Paper [18] describes the core part of ABS and its formal semantics while the tutorial [7] is about the modeling of variability in ABS using features and deltas. Delta-oriented programming [32] is a feature-oriented code reuse concept that is employed in HATS ABS as an alternative to traditional inheritance-based reuse.

The current paper is focussed on *architectural* aspects of modeling with ABS. A very brief summary of the main ideas of ABS is contained in Sect. 2 below.

In Sect. 3 we give a step-by-step guide on how to create a software product line in ABS from scratch using delta modeling. We make use of *abstract delta modeling* [8], dealing with conflicts explicitly. In Sect. 4 we turn to the question of how the deployment architecture of a system can be represented in a suitably platform-independent manner at the level of abstract models [19]. Sect. 5 reports
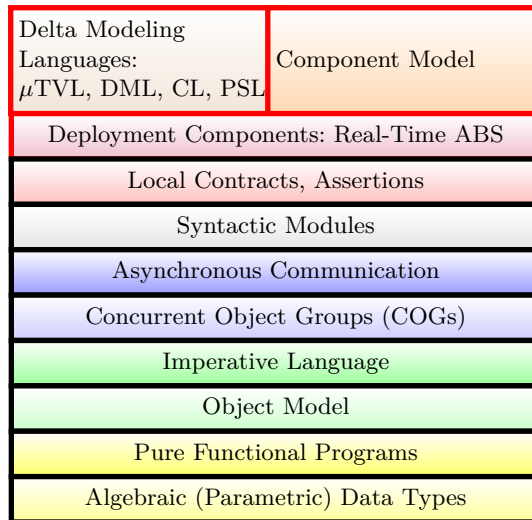
---

on the component model used in HATS which makes it possible to align ABS models with architectural languages and opens the possibility of dynamic reconfiguration [28]. Finally, in Sect. 6 we tell—in the context of an industrial case study [35]—how the concepts discussed in this paper were shaped by application concerns and what the industrial prospects of HATS ABS are. This includes an account on how to unit test ABS models using the ABSUnit framework [14].

2 Abstract Behavioral Specification

The ABS language is designed for formal modeling and specification of concurrent, component-based systems at a level that abstracts away from implementation details, but retains essential behavioral and even deployment aspects. ABS follows a layered approach (see Fig. 1): at the base are functional abstractions around a standard notion of parametric algebraic data types. Next we have a OO-imperative layer similar to (but much simpler than) JAVA. The concurrency model of ABS is two-tiered: at the lower level, so called

Delta Modeling Languages: μTVL, DML, CL, PSL	Component Model
Deployment Components: Real-Time ABS	
Local Contracts, Assertions	
Syntactic Modules	
Asynchronous Communication	
Concurrent Object Groups (COGs)	
Imperative Language	
Object Model	
Pure Functional Programs	
Algebraic (Parametric) Data Types	

Fig. 1. Layered Architecture of ABS

COGs (Concurrent Object Groups) encapsulate synchronous, multi-threaded, shared state computation on a single processor with cooperative scheduling. On top of this is an actor based model with asynchronous calls, message passing, active waiting, and future types. A syntactic module concept and assertions (including pluggable type systems) completes what we call *core ABS*. This language is described in detail in [18].

ABS classes do not support code inheritance and don't define types. Management of code reuse is, instead, realized by *code deltas*, described in [7]. These are named entities that describe the code changes associated to realization of new features. The result is a separation of concerns between architecural/design issues and algorithmic/data type aspects. It helps early prototyping and avoids a disconnect between a system's architecture and its implementation. In Sect. 3 we present the *delta modeling workflow* and demonstrate how it is used to implement software with a high degree of variability such as product lines.

Model-based approaches such as HATS face the challenge that, to be realistic, software models must address deployment issues, such as real-time requirements,

capacity restrictions, latency, etc. *Real-time ABS*, introduced in Sect. 4, uses an additional language layer called *deployment component* to achieve this.

To achieve flexible dynamic behavior, but also to structure and encapsulate the dependencies in a software system, a formal notion of logical *component* is essential. To this end, ABS features a *component model*, which is orthogonal to delta modeling and presented in Sect. 5. Whereas *deployment components* are used to identify the deployment structure of the modeled system in terms of locations, logical *components* are used to identify the logical structure of the system in terms of units of behavior. In particular, a logical component may be distributed over several deployment components and several logical components may be (partly) located in the same deployment component.

We stress that all ABS language constructs have a formal semantics, details of which can be found in the technical deliverables of the HATS project [11–13]. In addition, ABS has been designed with an eye on analysability. A wide variety of tools for simulation, testing, resource estimation, safety analysis, and verification of ABS models are available.

3 The Delta Modeling Workflow

Variability at the level of abstract behavioral specifications (or source code) is represented in the ABS using the concept of delta modeling. Delta modeling was introduced by Schaefer et al. [31, 32] as a novel modeling and programming language approach for software-based product lines. It can be seen as an alternative to feature-oriented programming [3]. Both approaches aim 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 this section, we describe briefly how delta modeling is instantiated in ABS to represent software product lines. A more detailed account is [7]. We also introduce the *delta modeling workflow* for ABS, a step-by-step guide to building a product line, which leverages the flexibility of delta modeling.

3.1 Delta Modeling in ABS

Variability of software product lines at the requirements level is predominantly represented in terms of product features, where a feature is a user-visible product characteristic or an increment to functionality. A feature model [22] provides the set of possible product variants by associating them with the set of realized features. Features at this modeling level are merely names.

Delta modeling is a flexible, yet modular approach to implement different product variants by reusable artifacts. In delta modeling, the realization of a software product line is divided into a core module and a set of delta modules. The core module implements the functionality common to all products of the software product line. Delta modules encapsulate modifications to the core product in order to realize other products. The modifications can include additions

and removals of product entities and modifications of entities that are hierarchically composed. A particular product variant can be automatically derived by applying the modifications of a selected subset of the given delta modules to the core product. Which delta modules have to be applied for a specific product variant is based on the selection of desired features for this product variant. In order to automate this selection, each delta is associated with an application condition that is a Boolean constraint over the features in the feature module. If the constraint is satisfied for a specific feature selection, the respective delta has to be applied to generate the associated product variant. To avoid or resolve conflicts when two delta modules modify the same product entity in an incompatible manner, delta module application can be partially ordered. This also ensures that for any feature selection a uniquely defined product is generated.

The ABS incarnation of delta modeling is based on four languages (μTVL, DML, CL, PSL) which are defined on top of core ABS (see Fig. 1). The feature description language μTVL is used to describe the variability of a product line in terms of features and their attributes. At this level of abstraction, a feature is a name representing user-visible system functionality. Attributes represent microvariability within features. μTVL is a textual description language for feature models and intended as a basis for other, e.g., graphical modeling formalisms.

The delta modeling language DML is used to specify delta modules containing modifications of a core ABS model. A delta module in ABS can add and remove classes and add as well as remove interfaces that are implemented by a class. Additionally, a delta module can modify the internal structure of classes by adding and removing fields and methods. Methods can also be modified by overriding the method body or by wrapping the previous implementation of the method using the `original()` call. Deltas can also be parameterized by specific values. Parameters are instantiated with concrete values during product generation, e.g., with the value that is set for a feature attribute.

The configuration language CL links μTVL feature models with the DML delta modules that implement the corresponding behavioral modifications. A CL specification provides application conditions for delta modules. They determine to which feature configurations modules are applied to in a `when` clause and provide an order to resolve conflicts between deltas modifying the same model entities The application ordering is given in `after` clauses attached to deltas stating that the delta must be applied after the given deltas if these are also applied during product generation.

The product selection language PSL is used to define the actual product of an ABS product line. A PSL script corresponds to a particular product variant and consists of two parts, namely, a specification of the features and their attributes selected for a product and an initialization block, which is often merely a call to an appropriate *main* method, even though it may contain configuration code. To generate the product specified by the PSL script, all deltas with a valid application condition for the given feature selection are applied to the core ABS model in some order compliant with the order specified in the CL script. Finally, the initialization block is added to the core program.

3.2 Delta Modeling Workflow in ABS

Abstract Delta Modeling [8,9] lends itself particularly well to a systematic development workflow for software product lines. One such workflow, dubbed *delta modeling workflow* [16,17], was adapted to ABS. In the following we describe the workflow specifically with ABS in mind, which has not been done before. It is also used in the case study in Sect. 6.

The workflow gives step-by-step instructions for development of a software product line from scratch, directing developers to satisfy local constraints (more formally given in [16]) to guarantee desirable properties for the whole product line. These properties are described in Sect. 3.3.
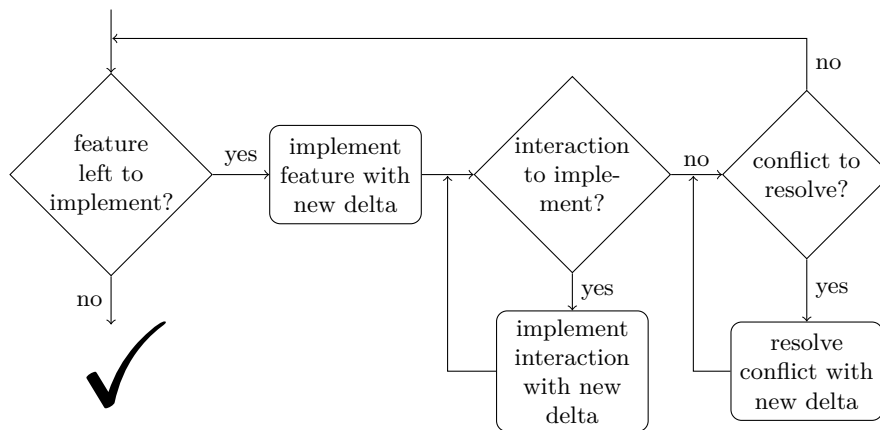

Fig. 2. Overview of the development workflow

A product line should be developed based on a product line specification, consisting of a feature model (which should include at least a subfeature hierarchy), and either formal or informal descriptions of each feature. When we start following the workflow, we assume that such a specification exists.

Briefly, features are implemented as a linear extension of the subfeature hierarchy. Base features are implemented first, subfeatures later, with one delta for each feature. Then, for every set of implemented features that should interact, but do not, we implement that interaction with a delta. Next, for every two deltas whose implementations are in conflict, a conflict resolving delta is written to resolve that issue. Fig. 2 shows a flow-chart that summarizes this process.

It is often suitable to put code common to all products into the core product. In the case of ABS, this means at least the following:

```
class Main { Unit run() {} }
{ new Main(); }
```
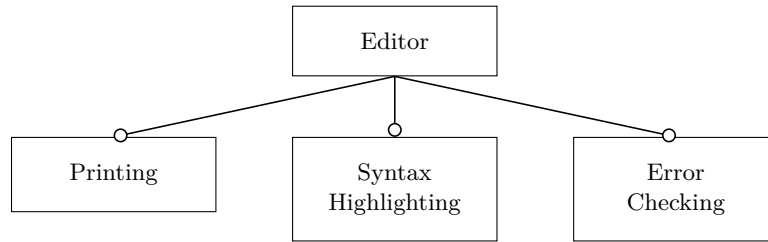
Fig. 3. Feature model of the Editor product line

We start with a `Main` class with an empty `run` method. The second line creates a new `Main` instance, implicitly calling the `run` method, which will later be modified by deltas. It is possible to put mandatory features into the core product, but it is recommended that all features are implemented by deltas, as this makes the product line more robust to evolution [33], and promotes separation of concerns.

Also, we begin with a minimal ABS product line configuration: the list of features and the list of desired products, which can be empty.

```
productline name {
    features d_1, d_2, ..., d_n; }
```

In the following workflow description, we will use a subset of the Editor product line example, a complete version of which may be found in [9]. It describes a set of code editor widgets, as may be found in integrated development environments. The feature model of the Editor product line is shown in Fig. 3. Now we walk through the Delta Modeling Workflow depicted in Fig. 2.

Feature left to implement? In this stage of the workflow, we choose the next feature to implement. Essentially we walk through the subfeature hierarchy of the feature model in a topological order, i.e., base features first, subfeatures later. If all features have been implemented, we are finished.

For the example, we would have to start with the `Editor` feature. Any of the three features on the second level may be chosen next.

Implement Feature with New Delta Having chosen a feature f, we now write a "feature delta" d_f to implement it:

```
delta d_f { ... }
```

The delta may add, remove or modify any classes and methods necessary to realize the functionality of f, while preserving the functionality of all superfeatures. The developer only has to consider the feature-local code: the core product and the deltas implementing superfeatures of f. We now show the four feature deltas of the Editor product line (we leave out some details for the sake of brevity):

```
delta D_Editor {
  adds class Model { ... }
  adds class Font {
    Unit setColor(Color c) { ... }
    Color getColor() { ... }
    Unit setUnderlined(bool u) { ... }
    bool getUnderlined() { ... }
  }
  adds class Editor { Model model;
    { model = new Model(); ... }
    Model getChar(int c) {
      return model.getChar(c);
    }
    Font getFont(int c) {
      return new Font();
    } }
  modifies class Main {
    modifies Unit run() {
      new Editor();
    } } }
```

```
delta D_SyntaxHighlighting {
  adds class Highlighter(Model m) {
    Model model;
    { model = m; }
    Color getColor(int c) { ... }
  }
  modifies class Editor {
    modifies getFont(int c) {
      Font f = D_Editor.original(c);
      Highlighter h =
          new Highlighter(getModel());
      f.setColor(h.getColor(c));
      return f;
    }
  }
}
```

```
delta D_Printing {
  modifies class Editor {
    adds Unit print() {
      // print the plain text
    } } }
```

```
delta D_ErrorChecking {
  modifies class Editor {
    modifies Font getFont(int c) {
      Font f = D_Editor.original(c);
      f.setUnderlined(
          getModel().isError(c));
      return f;
    } }
}
```

Finally, we add the following line to the ABS product line configuration:

```
delta d_f when f after d_s;
```

where d_s is the delta implementing the superfeature of f. If f has no superfeature, the after clause may be omitted. Our example requires the following product line configuration:

```
productline Editor {
  features Editor, Printing, SyntaxHighlighting, ErrorChecking;
  delta D_Editor when Editor;
  delta D_Printing when Printing after D_Editor;
  delta D_SyntaxHighlighting when SyntaxHighlighting after D_Editor;
  delta D_ErrorChecking when ErrorChecking after D_Editor; }
```

Interaction to Implement At the feature modeling and specification level, two features f and g may be independently realizable, but require extra functionality when both are selected. This behavior is not implemented by the feature deltas, so a new delta needs to be created. In our example, this is the case for the

features Printing and Syntax Highlighting. When printing, we would like the syntax highlighting colors to be used.

Implement Interaction The new delta $d_{f,g}$ must implement the required interaction without breaking the features f and g or their superfeatures. It may change anything introduced by feature deltas d_f and d_g. When overwriting methods, it may also access the original methods using the syntax d_f.`original()` and d_g.`original()`. In our example:

```
delta D_Printing_SyntaxHighlighting {
  modifies class Editor {
    modifies Unit print() {
      // print as before, but use colors of D_SyntaxHighlighting.font(c)
    } } }
```

Then we add the following to the ABS product line specification:

```
delta d_{f,g} when f && g after d_f, d_g;
```

In our example:

```
delta D_Printing_SyntaxHighlighting when Printing && SyntaxHighlighting
    after D_Printing, D_SyntaxHighlighting;
```

This may be generalized to interaction between more than two features.

Conflict to Resolve? By adding delta d_f, deltas $d_{f,g}$ (for different g) and conflict resolving deltas introduced earlier in the current iteration, we may have introduced an implementation conflict: two deltas d_1, d_2 that are independent, but modify the same method in a different way. In our example, this is the case for `D_SyntaxHighlighting` and `D_ErrorChecking`, as they both modify the `font` method in a different way, and are not ordered in the product line configuration. For each such conflict, we write a delta to resolve it.

Resolve Conflict The resolving delta $d_{1,2}$ must overwrite the methods causing the conflict, while not breaking the features implemented by d_1, d_2, or their superfeatures. Typically, $d_{1,2}$ invokes d_1.`original()` and d_2.`original()` to combine the functionality of the conflicting deltas. In our example:

```
delta D_SyntaxHighlighting_ErrorChecking {
  modifies class Editor {
    modifies Font getFont(int c) {
      Font result = D_Editor.original(c);
      result.setColor(D_SyntaxHighlighting.original(c).getColor());
      result.setUnderlined(D_ErrorChecking.original(c).getUnderlined());
      return result;
    } } }
```

We add the following to the ABS product line specification:

```
delta d_{1,2} when (λ(d_1)) && (λ(d_2)) after d_1, d_2;
```

where $\lambda(d)$ is the when clause of delta d. In our example:

```
delta D_SyntaxHighlighting_ErrorChecking when (SyntaxHighlighting) &&
    (ErrorChecking) after D_SyntaxHighlighting, D_ErrorChecking;
```

3.3 Discussion

The workflow has some useful properties, which we briefly explain. Any feature, as well as any conflict resolution and feature interaction, can be developed independently of others that are conceptually unrelated to it. For example, all feature deltas in our example could be developed at the same time and in isolation. As could the interaction implementing delta and the conflict resolving delta.

Then there are various properties that are guaranteed in the product lines resulting from this workflow. There is a minimum of code duplication. Every delta implements some specific functionality and every product that needs that functionality employs the same delta to use it. And when two features are conceptually unrelated, two unordered deltas are developed for them. This means that two unrelated features cannot unintentionally and silently overwrite each others' code. In other words, there is no *overspecification*.

Furthermore, product lines will be globally unambiguous, meaning that for every valid feature configuration, there is a uniquely generated product. Lastly, if local constraints are met, it is guaranteed that at the end of the workflow, all necessary features and feature combinations have in fact been implemented (this is because deltas in ABS satisfy the *non-interference property* [16]). The product line implementation is *complete*.

4 Deployment Modeling

The functional correctness of a product largely depends on its high-level behavioral specification, independent of the platform on which the resulting code will be deployed. However, different deployment platforms may be envisaged for different products in a software product line, and the choice of deployment platform for a specific product may hugely influence its quality of service. For example, limitations in the processing capacity of the CPU of a cell phone may restrict the features that can be selected, and the capacity of a server may influence the response time for a service for peaks in the client traffic. In this section, we give an overview of how deployment concerns are captured in ABS models.

Modeling Timed Behavior in ABS *Real-Time ABS* [4] is an extension of ABS in which the timed behavior of ABS models may be captured. An untimed

ABS model is a model in Real-Time ABS in which execution takes zero time; thus, standard statements in ABS are assumed to execute in zero time. Timing aspects may be added incrementally to an untimed behavioral model. Our approach extends ABS with a combination of *explicit* and *implicit* time models. In the explicit approach, the modeler specifies the passage of time in terms of duration statements with best and worst-case time. These statements are inserted into the model, and capture the duration of computations which do not depend on the deployment architecture. This is the standard approach to modeling timed behavior, known from, e.g., timed automata in UPPAAL [23]. In the implicit approach, the actual passage of time is measured during execution and may depend on the capacity and load of the server where a computation occurs.

Real-Time ABS introduces two new data types into the functional sublanguage of ABS: Time, which has the constructor Time(r), and Duration which has the constructors InfDuration and Duration(r), where r is a value of the type Rat of rational numbers. Let f be a function defined in the functional sublanguage of ABS, which recurses through some data structure x of type T, and let size be a measure of the size of this data structure. Consider a method m which takes as input such a value x and returns the result of applying f to x. Let us assume that the time needed for this computation depends on the size of x; e.g., the time needed for the computation will be between a duration 0.5*size(x) and a duration 4*size(x). We can specify an interface I which provides the method m and a class C which implements this method, including the duration of its computation using the *explicit* time model, as follows:

```
interface I { Int m(T x) }
class C implements I {
  Int m (T x){ duration(0.5*size(x), 4*size(x)); return f(x);
  } }
```

The object-oriented perspective on timed behavior is captured in terms of *deadlines* to method calls. Every method activation in Real-Time ABS has an associated deadline, which decrements with the passage of time. This deadline can be accessed inside the method body using the expression **deadline**(). Deadlines are *soft*; i.e., **deadline**() may become negative but this does not in itself stop the execution of the method. By default the deadline associated with a method activation is infinite, so in an untimed model deadlines will never be missed. Other deadlines may be introduced by means of call-site *annotations*.

Let o be an object of a class implementing method m. We consider a method n which invokes m on o, and let scale(d,r) be a scaling function which multiplies a duration d by a rational number r. The method n specifies a deadline for this call, given as an annotation and expressed in terms of its own deadline (if its own deadline is InfDuration, then the deadline to m will also be unlimited). Method n may be defined as follows:

```
Int n (T x){ [Deadline: scale(deadline(),0.9)] return o.m(x); }
```

In the *implicit* approach to modeling time in ABS, time is not specified directly in terms of durations, but rather *observed* on the model. This is done by comparing clock values from a global clock, which can be read by an expression **now**() of type Time. We specify an interface J with a method timer(x) which, given a value of type T, returns a value of type Duration, and implement timer(x) in a class D such that it measures the time it takes to call the method m above:

```
interface J { Duration timer (T x) }
class D implements J (I o) {
  Duration timer (T x){ Time start; Int y;
    start = now(); y=o.m(x); return timeDifference(now(),start);
  } }
```

With the implicit time model, no assumptions about specific durations are involved. The duration depends on how quickly the method call is effectuated by the object o. The duration is observed by comparing the time before and after making the call. As a consequence, the duration needed to execute a statement with the implicit time model depends on the *capacity* of the specific deployment model and on *synchronization* with (slower) objects.

Deployment Components A *deployment component* in Real-Time ABS captures the execution capacity associated with a number of COGs. Deployment components are first-class citizens in Real-Time ABS, and specify an amount of resources shared by their allocated objects. Deployment components may be dynamically created depending on the control flow of the ABS model or statically created in the main block of the model. We assume a deployment component environment with unlimited resources, to which the root object of a model is allocated. 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, a model without explicit deployment components runs in environment, which places no restrictions on the execution capacity of the model. A model may be extended with other deployment components with different processing capacities.

Given interfaces I, J and classes C, D as defined above, we can specify a deployment architecture, where two C objects are put on different deployment components Server1 and Server2, and interact with the D objects on the same deployment component Client. Deployment components have the type DC and are instances of the class DeploymentComponent, taking as parameters a name, given as a string, and a set of restrictions on resources. Here we focus on processing capacity, which is specified by the constructor CPUCapacity(r), where r represents the amount of available abstract processing resources between observable points in time. Below, we create three deployment components Server1, Server2, and Server3, with processing capacities 50, 100, and unlimited (i.e., Server3 has no resource restrictions). The local variables mainserver, backupserver, and clientserver refer to these deployment components. Objects are explicitly allocated to servers via annotations. The keyword **cog** indicates the creation of a new COG.

```
{ // The main block initializes a static deployment architecture:
  DC mainserver = new DeploymentComponent("Server1", set[CPUCapacity(50)]);
  DC backupserver = new DeploymentComponent("Server2", set[CPUCapacity(100)]);
  DC clientserver = new DeploymentComponent("Server3", EmptySet);
  [DC: mainserver] I object1 = new cog C;
  [DC: backupserver] I object2 = new cog C;
  [DC: clientserver] J client1 = new cog D(object1);
  [DC: clientserver] J client2 = new cog D(object2);
}
```

Resource Costs The resource capacity of a deployment component determines how much computation may occur in the objects allocated to that deployment component. Objects allocated to the deployment component compete for the shared resources 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 between observable (discrete) points in time, after which the resources are renewed.

The cost of executing a statement in the ABS model is determined by a default value which can be set as a compiler option (e.g., defaultcost=10). However, the default cost does not discriminate between the statements. If e is a complex expression, then the statement x=e should have a significantly higher cost than skip. For this reason, more fine-grained costs can be inserted into the model via annotations. For example, the exact cost of computing function f defined on p. 10 may be given as a function g depending on the size of the input x. Consequently, in the context of deployment components, we can specify a resource-sensitive re-implementation of interface I without predefined duration in class C2 as follows:

```
class C2 implements I {
  Int m (T x){ [Cost: g(size(x))] return f(x);
  } }
```

It is the responsibility of the modeler to specify the execution costs in the model. A behavioral model with default costs may be gradually refined to provide more realistic resource-sensitive behavior. For the computation of the cost function g in our example above, the modeler may be assisted by the COSTABS tool [1], which computes a worst-case approximation of the cost function for f in terms of the input value x based on static analysis techniques, given the ABS definition of the expression f. However, the modeler may also want to capture resource consumption at a more abstract level during the early stages of the system design, for example to make resource limitations explicit before a further refinement of a model. Therefore, cost annotations may be used by the modeler to abstractly represent some computation which remains to be fully specified. For example, the class C3 below may represent a draft version of our method m which specifies the worst-case cost of the computation even before the function f has been defined:

```
class C3 implements I {
  Int m (T x){ [Cost: size(x)*size(x)] return 0;
  } }
```

Costs need not depend merely on data values, but may also reflect overhead in general, as captured by expressions in ABS; e.g., a cost expression can be a constant value or depend on the current load of the deployment component on which the computation occurs.

Dynamic Deployment Architectures The example presented in this section concentrates on giving simple intuitions for the modeling of deployment architectures in ABS in terms of a static deployment scenario. A full presentation of this work, including the syntax and formal semantics of such deployment architectures, is given in [13,20]. Obviously, the approach may be extended to support the modeling of load-balancing strategies. We have considered two such extensions, based on adding an expression load(n) which returns the average load of the current deployment component over the last n time intervals. First, by including resources as first-class citizens of ABS and allowing (virtual) resources to be reallocated between deployment components [19]. Second, by allowing objects to be marshaled and reallocated between deployment components [21]. Furthermore, we have studied the application of the deployment component framework to memory resources and its integration with COSTABS in [2].

5   The ABS Component Model

Components are an intuitive tool to achieve unplanned dynamic reconfiguration. In a component system, an application is structured into several distinct pieces called *component*. Each of these components has dependencies towards functionalities located in other components; such dependencies are collected into the *output ports*. The component itself, however, offers functionalities to the other components, and these are collected into the *input ports*. Communication from an output port to an input port is possible when a *binding* between the two ports exists. Dynamic reconfiguration in such a system is then achieved by adding and removing components and by re-binding. Hence, updates and modifications acting on applications are possible without stopping them.

5.1   Related Work

While the idea of component is simple, bringing it into a concrete programming language is not easy. The informal description of components talks about the structure of a system, and how this structure can change at runtime, but does not mention program execution. As a matter of fact, many implementations of components do *not* merge into one coherent model (i) the execution of the program, generally implemented using a classic object-oriented language like JAVA

or C++, and (ii) the component structure, generally described in an annex Architecture Description Language (ADL). This approach makes it simple to add components to an existing language, however, unplanned dynamic reconfiguration becomes hard, as it is difficult to express modifications of the component structure using objects (as these are just supposed to describe the execution of the programs). For instance, models like Click [29] do not allow runtime modifications while OSGi [30] only allows the addition of new classes and objects: component deletion or binding modification are not supported. In this respect, a more flexible model is Fractal [5], which reifies components and ports into objects. Using an API, in Fractal it is possible to modify bindings at runtime and to add new components; still, it is difficult for the programmer to ensure that reconfiguration will not cause state inconsistencies.

Formal approaches to component models have been studied e.g., [6,25–28,34]. These models have the advantage of having a precise semantics, which clearly defines what is a component, a port and a binding (when such a construct is included). This helps to understand how dynamic reconfiguration can be implemented and how it interacts with the normal execution of a program. In particular, Oz/K [27] and COMP [26] propose a way to integrate in a unified model both components and objects. Oz/K, however, has a complex communication pattern and deals with adaptation via the use of *passivation*, which is a tricky operator [24] and in the current state of the art breaks most techniques for behavioral analysis. In contrast, COMP offers support for dynamic reconfiguration, but its integration into the semantics of ABS appears complex.

5.2 Our Approach

Most component models have a notion of component that is distinct from the objects used to represent the data and the execution of programs. Such languages are structured in two layers, one using objects for the main execution of the program, one using components for dynamic reconfiguration. Even though such a separation seems natural, it makes the integration of requests for reconfiguration into the program's workflow difficult. In contrast, in our approach we went for a uniform description of objects and components; i.e., we enhance objects and COGs—the core ingredients of ABS—with the core elements of components (ports, bindings, consistency, and hierarchy) to enable dynamic reconfiguration.

We achieved this by exploiting the similarities between objects (and object groups) and components. Most importantly, the methods of an object closely resemble the input ports of a component. In contrast, objects do not have explicit output ports, but the dependencies of an object can be stored in internal fields. Thus, rebinding an output port corresponds to the assignment of a new value to the field. Standard objects, however, cannot ensure consistency of the rebinding. Indeed, suppose we wished to treat certain object fields as output ports: we could add methods to the object for their rebinding; but it would be difficult, in presence of concurrency, to ensure that a call to one of these methods does not harm ongoing computations. For instance, if we need to update a field (like the driver of a printer), then we would first want to wait for the termination

of all current executions referring to that field (e.g., printing jobs). COGs (object groups) in ABS offer a mechanism to avoid race conditions at the level of methods, by ensuring that there is at most one task running in a COG. But this mechanism is not sufficient to deal with rebinding where we may need to wait for *several* methods to finish before performing it. Another difference between object and component models is that the latter talk about *locations*. Locations structure a system, possibly hierarchically, and can be used to express dynamic addition or removal of code, as well as distributed computation.

To ensure the consistent modifications of bindings and the possibility to ship new pieces of code at runtime, we add four elements to the ABS core language:

1. A notion of output port distinct from an object's fields. The former (identified with the keyword **port**) represent the object's dependencies and can be modified only when the object is in a *safe* state; the latter constitute the inner state of an object and can be modified with ordinary assignments.
2. The possibility to annotate methods with the keyword **critical**: this specifies that the object, while this method is executing, is not in a safe state.
3. A new primitive to wait for an object to be in a safe state. Thus, it becomes possible to wait for all executions using a given port to finish, before rebinding the port to a new object.
4. We add locations. Our semantics structures an ABS model into a tree of locations that can contain object groups, and that can move within the hierarchy. Using locations, it is possible to model the addition of new pieces of code to a program at runtime. Moreover, it is also possible to model distribution (each top-level location being a different computer) and code mobility (by moving a sub-location from a computer to another one).

The resulting component language remains close to the underlying ABS language and, in fact, is a conservative extension of ABS (i.e., a core ABS model is valid in our language and its semantics is unchanged). As shown in the following example, introducing the new primitives into a given ABS model is simple. In contrast to previous component models, our language does not strongly separate objects and components. Three major features of the informal notion of component—ports, consistency, and location—are represented in the language as follows: (i) output ports are taken care of at the level of our enhanced objects; (ii) consistency is taken care of at the level of COGs; (iii) information about locations is added explicitly.

5.3 Example

We illustrate our approach with an example inspired from the Virtual Office case study of the HATS project [10]. This case study supposes an open environment with resources like computers, projectors or printers that are used to build different workflows. Assume we want to define a workflow that takes a document (a resource modeled with the class Document), modifies it using another resource (modeled with the class Operator) and then sends it to a printer

(modeled with the class Printer). We suppose that the protocol used by Operators is complicated, so we isolate it into a dedicated class. Finally, we want to be able to change protocol at runtime, without disrupting the execution of previous instances of the workflow. Such a workflow can be defined as follows:

```
class OperatorFrontEnd(Operator op) {
 port Operator _op;

 critical Document modify(Document doc) { ... }

 { rebind _op = op; }
}

class WFController(Document doc, Operator op, Printer p) {
 port Document _doc;
 port Printer _p;
 OperatorFrontEnd _opfe;

 critical void newInstanceWF() { ... }

 void changeOperator(Operator op) {
  await(‖this,_opfe‖);
  rebind _opfe._op = op;
 }

 {
  rebind _doc = doc;
  rebind _p = p;
  _opfe = new OperatorFrontEnd(op);
 } }
```

We have two classes: the class OperatorFrontEnd implements the protocol in the method modify(doc); the class WFController encodes the workflow. The elements _op, _doc and _p are *ports* (annotated as **port**) and represent dependencies to external resources. It is only possible to modify their value using the construct **rebind**, which checks whether the object is in a safe state (no critical method in execution) before modifying the port. Moreover, methods modify(doc) and newInstanceWF() make use of these ports in their code, and are thus annotated as **critical** as it would be dangerous to rebind ports during their execution.

The key operations of our component model are shown in the two lines of code in the body of the method changeOperator(op). First is the **await** statement, which waits for objects **this** and _opfe to be in a safe state. By construction, these objects are in a safe state when there are no running instances of the workflow: it is then safe to modify the ports. Second is the **rebind** statement: it will succeed, because the concurrency model of COGs ensures that no workflow instance can be spawned between the end of the **await** and the end of the method. Moreover, the second line shows that it is possible to rebind a port of *another* object, provided that this object is in the same COG as the one doing the rebinding.
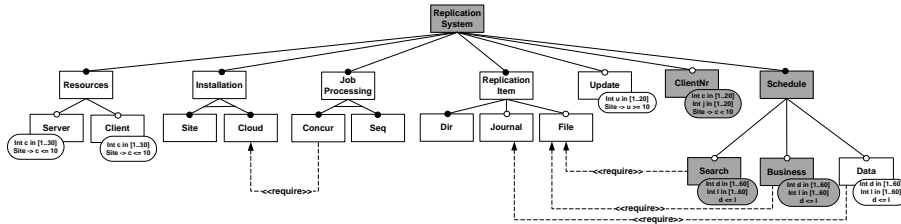
Fig. 4. Feature diagram of the replication system

6 An Industrial Case Study

The Fredhopper Access Server (FAS) is a distributed, concurrent OO system that provides search and merchandising services to e-Commerce companies. Briefly, FAS provides to its clients structured search capabilities within the client's data. FAS consists of a set of live and staging environments. A live environment processes queries from client web applications via web services. FAS aims at providing a constant query capacity to client-side web applications. A staging environment is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments according to the replication protocol. The replication protocol is implemented by the *Replication System*. The replication system consists of a SyncServer at the staging environment and one SyncClient for each live environment. The SyncServer determines the schedule of replication, as well as their contents, while SyncClient receives data and configuration updates.

Modeling Variability There are several variants of the Replication System and we express them as features. Fig. 4 shows the feature diagram of the replication system. For brevity, we consider only features *ReplicationSystem*, *ClientNr*, *Schedule*, *Search*, and *Business*. These are shaded in the feature diagram; full treatment of the complete feature diagram can be found in the HATS project report [15]. We list the μTVL model that describes these features.

```
root ReplicationSystem { group allof {
  opt ClientNr { Int c in [1 .. 20]; Int j in [1 .. 20]; },
  Schedule { group allof {
    opt Search { Int d in [1 .. 60]; Int l in [1 .. 60]; d <= l; },
    opt Business { Int d in [1 .. 60]; Int l in [1 .. 60]; d <= l; }
} } } }
```

The replication system has the optional feature ClientNr for specifying the number of SyncClients participating in the replication protocol. It has the mandatory feature Schedule for specifying replication schedules. Replication schedules dictate when and where the replication system should monitor for changes in the

staging environment to be replicated to the live environments. A replication system may offer one or both of Search and Business features. The feature Search specifies the interval in which the replication system replicates the changes from the search index. The search index is the underlying data structure for providing search capability on customers' product items. The feature Business specifies the intervals for replicating business configuration. The business configuration defines the presentation of search results, such as sorting and promotions.

We employ the delta modeling workflow (Sect. 3) to construct an ABS model of the replication system. We start with an empty product line and define the core product as **class** Main {} { **new** Main(); }. Following the delta modeling workflow, we begin with the base feature ReplicationSystem. We model this feature by the delta SystemDelta:

```
delta SystemDelta {
  modifies class Main {
    adds Unit run() {
      List<Schedule> ss = this.getSchedules();
      Set<ClientId> cs = this.getCids();
      Int maxJobs = this.getMaxJobs();
      Int updates = this.getUpdateInterval();
      new ReplicationSystem(updates, ss, maxJobs, cs);
    } } }
```

The run() method creates a ReplicationSystem according to default setup. In addition, SystemDelta adds the necessary type definitions, such as data types, type synonyms, and core ABS classes and interfaces that model the underlying file system, the SyncClient and the SyncServer. Next to consider is the optional feature ClientNr, implemented by ClientNrDelta:

```
delta ClientNrDelta(Int c, Int j) {
  modifies class Main {
    modifies Set<ClientId> getCids() {
      Int s = c; Set<Int> cs = EmptySet;
      while (s > 0) { cs = Insert(s,cs); s = s-1; } return cs; }
    modifies Int getMaxJobs() { return j; }
  } }
```

The delta modifies getCids() and getMaxJobs() of class Main such that the replication system has synchronisation clients and a maximum number of replication jobs per client. Now we consider the mandatory feature Schedule, implemented by ScheduleDelta:

```
delta ScheduleDelta {
  modifies class Main {
    adds List<Pair<String,List<Item>>> searchItems = ... ;
    adds List<Pair<String,List<Item>>> businessItems = ... ;
    adds List<Schedule> getSchedules() {
      Map<String,Pair<Int,Deadline>> m = this.getScheduleMap();
```

```
      return itemMapToSchedule(Nil, m, concatenates(list[searchItems])); }
  } }
```

This delta adds methods and fields to `Main` to model various schedule informa-
tion such as the types of schedules and their possible file locations from which
changes are replicated. The next feature we consider is the optional feature
Search, implemented by `SearchDelta`:

```
delta SearchDelta(Int d, Int l) {
  modifies class Main {
    modifies Map<String,Pair<Int,Deadline>> getScheduleMap() {
      return put(ScheduleDelta.original(), "Search", Pair(d, Duration(l))); }
  } }
```

This modifies method `getScheduleMap()` to set the interval between replicating
the search index directory and the deadline for each such replication job as
specified by feature Search. Since replicating the search index directory is the
default schedule as defined by feature Schedule, this delta only modifies the
specification of the schedule. Finally, we consider feature Business, implemented
by `BusinessDelta`:

```
delta BusinessDelta(Int d, Int f) {
 modifies class Main {
  modifies Map<String,Pair<Int,Deadline>> getScheduleMap() {
   return put(ScheduleDelta.original(), "Business rules", Pair(d, Duration(l)));
  }
  modifies List<Schedule> getSchedules() { ... }
  } }
```

Similar to `SearchDelta`, this delta modifies method `getScheduleMap()` to set the
interval between replicating a set of file locations and the deadline for each such
replication job. In addition, it modifies `getSchedules()` to add schedules for
business configuration to the replication system, the details of which we omit.
We notice that `BusinessDelta` causes a conflict with `SearchDelta`. We resolve
this conflict by providing the resolving delta `SBDelta`:

```
delta SBDelta {
  modifies class Main {
    modifies List<Schedule> getSchedules() {
      return appendRight(SearchDelta.original(), BusinessDelta.original()); }
  } }
```

`SBDelta` resolves the conflict between `BusinessDelta` and `SearchDelta` by insisting
that the returned list of schedules contains the list of search index directory
replication schedules *followed by* the list of business configuration replication
schedules. Note that while both deltas modify `getScheduleMap()`, the order in
which the modifications are applied needs not be specified, therefore, `SBDelta`
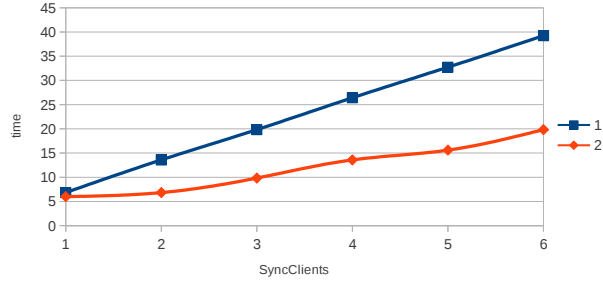
Fig. 5. Average execution time of client jobs

does not provide a conflict resolver for those modifications. With no further feature interaction or conflict resolution to implement in this iteration, and no further features to implement, we obtain the complete product line.

Resource Simulation Using Real-Time ABS and deployment components (Sect. 4), we augment the replication system model with resource information such as processing power. We simulate the effects of processing power during execution of the replication system with the Maude backend of the ABS compiler. The following shows partial definitions of classes `ConnectionThread`, `ClientJob`.

```
class ConnectionThread {
 Unit run() {
  [Cost:size(sch)] this.start(sch); ... [Cost:length(fs)] this.register(fs); ...
  [Cost:length(fs)] this.transfer(fs); ... [Cost:size(sch)] this.finish(sch); }}
class ClientJob(Client client, ...) {
 Int total = 0;
 Unit run() { Time bt = now(); ... total = timeDifference(bt,now()); }}
```

Each `ConnectionThread` object, created by SyncServer, provides a `run()` method for interacting with a `ClientJob` object to fulfill the staging environment side of the replication protocol, while each `ClientJob` object, created by `SyncClient`, provides a `run()` method to fulfill the live environment side of the replication protocol. We provide cost annotations for specific method invocations of the `run()` method of `ConnectionThread` to describe the amount of CPU resources.[7] We inject time stamps at the beginning and the end of the `run()` method of `ClientJob` to calculate the execution time of a client job. Fig. 5 shows a graph of the average execution time (in simulated time units) of client jobs depending on the number of SyncClients and compares one vs. two CPUs. The graph shows that with a single CPU, the client job execution time increases linearly with the number of SyncClients, while with two CPUs this is no longer the case.

---

[7] Cost expressions are abstractions from concrete values obtained using the combination of real-time simulation and static cost analysis [1].

Unit Testing During development of the replication system unit tests were written to validate the class methods and to detect regressions. We created the ABSUnit testing framework, based on the xUnit architecture, for writing unit tests for ABS [14]. We illustrate ABSUnit with method `processFile(id)` of `ClientJob`. This method checks whether a file named `id` exists in the underlying database and returns its size. We define an interface `ClientJobTest` as the type of the test suite. It defines a test method `test()` and data points `getData()`:

```
type Data = Map<Fn,Maybe<Size>>;
[Suite] interface ClientJobTest {
  [Test] Unit test(Data ds);
  [DataPoint] Set<Data> getData(); }
```

The return value of data points serves as input to the test method. The following listing shows a part of the class `TestImpl` that implements the methods `test()` and `getData()` from the interface `ClientJobTest`.

```
interface Job { Maybe<Size> processFile(Fn id); Unit setDB(DataBase db); }
[SuiteImpl] class TestImpl implements ClientJobTest {
  Set<Data> testData = ...; ABSAssert aut = ...
  Set<Data> getData() { return testData }
  Job getCJ(DataBase db) { return null; }
  Unit test(Data ds) {
    DataBase db = new cog TestDataBase(ds); Job job = this.getCJ(db);
    Set<Fn> ids = keys(ds);
    while (hasNext(ids)) {
      Pair<Set<Fn>,Fn> nt = next(ids); Fn i = snd(nt); ids = fst(nt);
      Maybe<Size> s = job.processFile(i);
      Comparator cmp = new MComp(lookup(ds,i),s);
      aut.assertEquals(cmp); }}}
```

Method `test()` defines a test case on `processFile(id)`. Class `MComp` provides a comparator between two `Maybe<Size>` values. To ensure the client job object under test is prepared for unit testing, we define a delta to remove the `run()` method, add a setter, add a mock implementation of the database for testing, and assign type `Job` to `ClientJob`, so we can add a mock database to the object under test. This is a *major advantage of delta modeling*: code needed only for testing is encapsulated in test deltas and does not clutter up productive code.

```
delta JobTestDelta {
  modifies class ClientJob implements Job {
    removes Unit run(); adds Unit setDB(DataBase db) { this.db = db; }}
  modifies class TestImpl {
    modifies Job getCJ(DataBase db) {
      Job cj = new ClientJob(null); cj.setDB(db); return cj; }}}
```

The ABSUnit framework comes with a test runner generator that is built into to the ABS frontend. The test runner generator takes `.abs` files of the system

under test and returns an `.abs` file defining a main block that executes the test cases concurrently. Here is the test runner for test interface `ClientJobTest`:

```
{ Set<Fut<Unit>> fs = EmptySet; Fut<Unit> f;
  ClientJobTest gd = new TestImpl(); Set<Data> ds = gd.getData();
  while (hasNext(ds)) {
   Pair<Set<Data>,Data> nt = next(ds); Data d = snd(nt); ds = fst(nt);
   ClientJobTest gd = new cog TestImpl(); f = gd!test(d); fs = Insert(f,fs); }
  Pair<Set<Fut<Unit>>,Fut<Unit>> n = Pair(EmptySet,f);
  while (hasNext(fs)) { n = next(fs); f = snd(n); fs = fst(n); f.get; }}
```

7 Conclusion

We gave an overview over the solutions to architectural issues provided by the ABS language developed in the EU FP7 project HATS. In contrast to many other behavioral modeling formalisms, ABS provides first-class support for feature models *and* connects them to implementations by a variant of feature-oriented programming called *delta modeling*. This allows to formally define a systematic *delta modeling workflow* for a feature-driven modeling process, which integrates very well with standard quality assurance techniques such as unit testing where it achieves a separation of concerns. As all software is deployed inside a wider system architecture, it is crucial to model and analyze constraints coming from deployment, which in ABS is done by *deployment components*. To structure and dynamically reconfigure a system one needs a suitable notion of components. ABS components are a conservative extension of ABS with a formal semantics. A case study, which is briefly reported in Sect. 6, demonstrates that the ABS approach scales to industrial applications. In a future paper we will concentrate on the *tool chain* shipped with the ABS environment, which contains a wide range of analysis and code generation tools.

References

1. 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.
2. E. Albert, S. Genaim, M. Gómez-Zamalloa, E. B. Johnsen, R. Schlatte, and S. L. T. Tarifa. Simulating Concurrent Behaviors with Worst-Case Cost Bounds. In *17th Interational Symposium on Formal Methods (FM 2011)*, volume 6664 of *LNCS*, pages 353–368. Springer, 2011.
3. D. S. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.*, 30(6), 2004.
4. 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.

5. E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. The Fractal Component Model and its Support in Java. *Software - Practice and Experience*, 36(11-12), 2006.
6. G. Castagna, J. Vitek, and F. Z. Nardelli. The Seal calculus. *Inf. Comput.*, 201(1), 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, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 417–457. Springer-Verlag, 2011.
8. D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract Delta Modeling. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 13–22, New York, NY, USA, Oct. 2010. ACM.
9. D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. Accepted to Special Issue of MSCS, to appear.
10. Evaluation of Core Framework, Aug. 2010. Deliverable 5.2 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.
11. Report on the Core ABS Language and Methodology: Part A, Mar. 2010. Part of Deliverable 1.1 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.
12. Full ABS Modeling Framework, Mar. 2011. Deliverable 1.2 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.
13. A configurable deployment architecture, Feb. 2012. Deliverable 2.1 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.
14. Debugging, visualization, and test generation, Mar. 2012. Deliverable 2.3 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.
15. Evaluation of Modeling, Mar. 2012. Deliverable 5.3 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.
16. M. Helvensteijn. Delta Modeling Workflow. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-intensive Systems, Leipzig, Germany, January 25-27 2012*, ACM International Conference Proceedings Series. ACM, 2012.
17. M. Helvensteijn, R. Muschevici, and P. Wong. Delta Modeling in Practice, a Fredhopper Case Study. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-intensive Systems, Leipzig, Germany, January 25-27 2012*, ACM International Conference Proceedings Series. ACM, 2012.
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 *LNCS*, pages 142–164. Springer-Verlag, 2011.
19. 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-Verlag, Nov. 2010.
20. 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-Verlag, 2011.

21. E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. A formal model of object mobility in resource-restricted deployment scenarios. In F. Arbab and P. Ölveczky, editors, *Proc. 8th International Symposium on Formal Aspects of Component Software (FACS 2011)*, volume 7253 of *Lecture Notes in Computer Science*, pages 187–. Springer-Verlag, 2012.

22. K. Kang, J. Lee, and P. Donohoe. Feature-Oriented Project Line Engineering. *IEEE Software*, 19(4), 2002.

23. 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.

24. S. Lenglet, A. Schmitt, and J.-B. Stefani. Howe's Method for Calculi with Passivation. In M. Bravetti and G. Zavattaro, editors, *CONCUR 2009 - Concurrency Theory*, volume 5710 of *LNCS*, pages 448–462. Springer-Verlag, 2009.

25. F. Levi and D. Sangiorgi. Mobile safe ambients. *ACM. Trans. Prog. Languages and Systems, vol. 25, no 1*, 2003.

26. M. Lienhardt, I. Lanese, M. Bravetti, D. Sangiorgi, G. Zavattaro, Y. Welsch, J. Schäfer, and A. Poetzsch-Heffter. A component model for the ABS language. In *Formal Methods for Components and Objects (FMCO) 2010*, volume 6957 of *LNCS*, pages 165–185. Springer, 2010.

27. M. Lienhardt, A. Schmitt, and J.-B. Stefani. Oz/k: A kernel language for component-based open programming. In *GPCE'07: Proceedings of the 6th international conference on Generative Programming and Component Engineering*, pages 43–52, New York, NY, USA, 2007. ACM.

28. F. Montesi and D. Sangiorgi. A model of evolvable components. In M. Wirsing, M. Hofmann, and A. Rauschmayer, editors, *Trustworthly Global Computing*, volume 6084 of *Lecture Notes in Computer Science*, pages 153–171. Springer-Verlag, 2010.

29. R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. In *ACM Symposium on Operating Systems Principles*, 1999.

30. OSGi Alliance. *Osgi Service Platform, Release 3*, 2003.

31. I. Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Proc. of 4th Intl. Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, 2010.

32. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proc. of 14th Software Product Line Conference (SPLC 2010)*, Sept. 2010.

33. I. Schaefer and F. Damiani. Pure Delta-oriented Programming. In S. Apel, D. Batory, K. Czarnecki, F. Heidenreich, C. Kästner, and O. Nierstrasz, editors, *Proc. 2nd International Workshop on Feature-Oriented Software Development (FOSD'10) Eindhoven, The Netherlands*, pages 49–56. ACM Press, 2010.

34. A. Schmitt and J.-B. Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Global Computing*, volume 3267 of *LNCS*. Springer, 2005.

35. P. Y. H. Wong, N. Diakov, and I. Schaefer. Modelling Distributed Adaptable Object Oriented Systems using HATS Approach: A Fredhopper Case Study. In B. Beckert, F. Damiani, and D. Gurov, editors, *2nd International Conference on Formal Verification of Object-Oriented Software*, volume 7421 of *LNCS*. Springer-Verlag, 2012.