# Modeling Spatial and Temporal Variability with the **HATS** Abstract Behavioral Modeling Language*

Dave Clarke[1], Nikolay Diakov[2], Reiner Hähnle[3], Einar Broch Johnsen[4], Ina Schaefer[5], Jan Schäfer[6], Rudolf Schlatte[4], and Peter Y. H. Wong[2]

[1] Katholieke Universiteit Leuven, Belgium
[2] Fredhopper B.V., Amsterdam, The Netherlands
[3] Chalmers University of Technology, Sweden
[4] University of Oslo, Norway
[5] Technische Universität Braunschweig, Germany
[6] Technische Universität Kaiserslautern, Germany

**Abstract.** The *Abstract Behavioral Specification* (ABS) language facilitates to precisely model the behavior of highly configurable, distributed systems. Its basis is Core ABS which is a strongly typed, abstract, object-based, concurrent, fully executable modeling language. Spatial variability of ABS models is represented by feature models, delta modules containing modifications of ABS models, product line configurations linking delta modules with product features and product selections specifying actual product instances. Temporal variability is captured by dynamic delta modules that can be applied to perform runtime updates. The feasibility of ABS is demonstrated by modeling an industrial-scale web merchandising system.

## 1 Introduction

Contemporary software development faces recurring challenges over many different application domains: software systems are concurrent and distributed, they exhibit a large variety of features and deployment scenarios, their requirements change frequently, and new requirements arise unexpectedly. All of these characteristics are increasingly difficult to address on the level of implementation languages, such as C/C++, C#, or Java. Even when it is possible, the result is often a large gap between the implemented system and the requirements documentation resulting in high validation and maintenance costs and impeding traceability.

Further, major IT-trends under way pose new challenges: a prerequisite for cloud computing is the ability to abstract away from physical resource allocation, load distribution, the architecture of the execution platform, etc. This implies the need to specify intended behavior without referring to concrete resources.

* This research is funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu).

Likewise, the emergence of cyber-physical systems and the internet of things emphasize the need for *abstract* behavioral description of highly configurable and diverse systems.

Model-centric approaches to system development are gaining rapidly in popularity in order to provide an abstract representation of system structure and behavior. There is a lot of research involving feature description languages [5], architectural languages for components [17], or UML and state machine-based notations [1, 12, 25]. Development processes, such as software product line engineering [18] distinguish between generic artifact and product-level system development and are specifically designed to use (and reuse) high-level artifacts. Hence, modeling languages capturing system diversity are required to deal with the variability of generic development artifacts. The main limitation of existing modeling approaches is their insularity and the lack of a unified formal semantics. Both is necessary, however, to provide a future generation of development tools that can, for example, generate code from models that is guaranteed to be sound, generate test cases that have a guaranteed coverage, or ensure that the implementation of features obeys their application constraints.

The HATS (Highly Adaptable and Trustworthy Software using Formal Models) project develops an executable modeling language called *Abstract Behavioral Specification* (ABS) language and an accompanying tool framework that promises to overcome the mentioned shortcomings: it facilitates to model precisely the behaviour of highly configurable, distributed systems in an "end-to-end" manner. This means that not only the (concurrent) implementation of features is captured, but also the feature space and the dependencies among them. Furthermore, ABS includes language concepts to represent model evolution, e.g., due to changing requirements. In addition, it is possible to formally specify properties of systems modeled with ABS in form of behavioral contracts. The ABS modeling language aims to fill the gap between structural high-level modeling languages, such as UML, and implementation-close formalisms, including programming languages.

Fig. 1 shows the different language layers constituting the ABS. At its core, ABS is a state-of-the-art, strongly typed, abstract, concurrent, object-based modeling language (Core ABS) that is fully executable. Shared memory is only permitted among closely collaborating synchronous groups of objects. Otherwise, objects communicate asynchronously and use message passing to update the state. This core language is described in Section 2. While ABS is an object-based language and, hence, compatible with the UML world, code reuse by inheritance, which tends to be brittle, is excluded. Instead, system diversity in ABS is captured by delta modeling [19, 20, 22, 21] which represents a set of systems by a designated core system and a set of system deltas specifying modifications to the core system. Delta modeling is an incremental composition technique that is highly compatible with feature-oriented software development [2] and also a good match for agile and evolutionary development approaches [3].

In Section 3, we describe how *spatial variability* is captured using the different language layers of the ABS (cf. Fig. 1). Spatial variability is concerned with the

| Language | Rôle |
|---|---|
| Core ABS | Specifies core behavioural modules (independent of extensions) |
| Micro Textual Variability Language ($\mu$TVL) | Feature models, attributes and constraints on them |
| Delta Modelling Language (DML) | Modifications to core behavioural modules |
| Product Line Configuration Language (CL) | Links features and delta modules, configures deltas with attributes |
| Product Selection Language (PSL) | Feature and attributes selections plus product initialisation block |

Fig. 1: Language definitions in ABS

modeling of anticipated features, as well as known system diversity and deployment scenarios. First, we introduce $\mu$TVL (Micro Textual Variability Language) to represent feature models and feature constraints. Second, we describe DML (Delta Modelling Language) that captures variability of Core ABS models by the concepts of delta modeling. Finally, we present CL (Product Line Configuration Language) for configuring a product line of ABS models and PSL (Product Selection Language) for deriving particular products from an ABS product line.

Even more difficult than spatial variability, but of growing importance is *temporal variability* or evolvability in time. The main difference to spatial variability is that temporal variability is not known in advance and cannot be anticipated. In Section 4, we show how delta modeling can be adapted to deal with temporal evolution of ABS product lines including the possibility to perform runtime system updates.

The measure of success for an ambitious and holistic approach such as it is pursued in HATS is whether one can model not only toy examples, but real industrial scenarios. In Section 5, we present an ABS modeling example that is taken from production code used in the distributed web merchandising platform—Fredhopper Access Server [11]. We conducted this industrial-strength case study along with the development of the ABS language and tools, in order to provide valuable early feedback to guarantee strong results. Section 6 describes the ABS tool set consisting of a parser, type checker, editor, debugger, and code generators. We conclude in Section 7 with an evaluation of what has been achieved with HATS ABS so far.

## 2 Core ABS

*Core ABS* (or simply ABS in the following) is an object-based modeling language. With its object-based model structure, it fits well with modeling languages used in object-oriented analysis and design, such as UML. However, code reuse via class-based inheritance is excluded. Instead, variability and code reuse in ABS models is achieved by specific language constructs as explained in Section 3.

ABS is designed to model distributed systems that communicate asynchronously by exchanging messages. The concurrency model of ABS is similar to that of JCoBox [24], which generalizes the concurrency model of Creol [15, 4] from single concurrent objects to concurrent object groups (COG). COGs can be regarded as object-based runtime components, which have their own heap of objects and solely communicate via asynchronous method calls. The behavior of a COG is represented by cooperative multi-tasking. Cooperative multi-tasking guarantees data-race freedom inside a COG and enables the safe combination of active and reactive behavior.

Beside the object-based part, the core language supports user-defined data types with (non-higher-order) functions and pattern matching. This functional sublanguage of ABS is largely orthogonal to the object-based part and is intended to model data. As such data is immutable, it can safely be exchanged between COGs. Using functional data types to realize most internal data structures of COGs will simplify the specification and verification of COGs.

The ABS language contains non-deterministic constructs; in particular, the outcome of executing concurrency primitives is non-deterministic. While underspecification is used to realize abstraction on data, non-deterministic execution semantics is the prerequisite for abstracting behavior. As ABS is a modeling language, we do not want to make any a priori assumptions about, for example, a concrete scheduling mechanism. Underspecification and non-determinism do not preclude executability: an unknown value is still a value and the outcome of a non-deterministic statement is a set of possible successor states from which one can be picked in simulation and visualization.

In this section, we first describe how to represent data in the ABS. Second, we explain the object-based fragment and the module system of the ABS. Finally, we present the concurrency model that is based on COGs and cooperative multitasking. A complete description of all ABS features is in [10].

## 2.1 Data Types

**Built-In Data Types.** ABS does *not* have primitive types, but a number of built-in data types and operators to work with basic values.

*The Unit Value.* To express that an expression has *no* value, ABS has the data type `Unit`, which has only one, identically named, constructor `Unit`. The `Unit` type is typically used for methods that do not have a return value.

*Logical Values.* ABS supports logical values by the `Bool` data type. It has the two constructors `True` and `False`. The defined operators on `Bool` are *equality* (`==`), *unequality* (`!=`), *negation* (`~`), *logical and* (`&&`), and *logical or* (`||`).

**Example:**

```
~((True && False) || True)
```

*Numbers.* ABS supports unbounded integers by the data type `Int`. Integers are constructed by using *integer literals*, which are positive numbers of an arbitrary length, e.g., `0, 1, 3434, 4711, 42`. ABS supports the standard arithmetic operations on `Int` with the usual precedences, i.e., *negation* (`-`), *addition* (`+`), *subtraction* (`-`), *multiplication* (`*`), *division* (`/`), and *modulo* (`%`).

**Example:**

```
((-5+6)*4)/(2%1)
```

*Character Sequences.* Character sequences are represented by the data type `String`. Strings are constructed by using *string literals*, which are sequences of characters enclosed by double quotes (`"`). There is no special data type for single characters, as a single character can be regarded as a string of length 1. The *concatenation* operator (`+`) can be used to concatenate two strings. The length of a string can be obtained by the `length` function.

**Example:**

```
"Hello" + "World"
```

**Algebraic Data Types.** Immutable values can be defined in ABS by *algebraic data types*. The possible values of data types are defined by a finite set of *data type constructors*. Constructors can have a finite list of parameters, which can refer to built-in types, algebraic data types, or *reference types* (see Section 2.2). The names of data types and constructors start with an upper case letter. The following example defines the data type `Fruit` with three constructors, and the data type `Juice` with the two constructors `Pure` and `Mixed`. With these data types, we can create a cherry-banana juice.

**Example:**

```
data Fruit = Apple | Banana | Cherry;
data Juice = Pure(Fruit) | Mixed(Juice, Juice);
Mixed(Pure(Cherry),Pure(Banana))
```

*Parametric Data Types.* ABS also supports data types with *type parameters* to define generic data types. A typical example is a `List` data type that should be generic in the types of its elements. In ABS, there is the predefined `List` data type, which is defined as follows:

**Example:**

```
data List<T> = Nil | Cons(T, List<T>);
```

*Type Synonyms.* To define shortcuts for types, ABS knows *type synonyms*, which are defined by using the **type** keyword. Semantically, a type synonym is equivalent to its aliased type.

**Example:**

```
type Catalog = Map<String, Product>;
```

*Functions.* Functions in ABS are used for working with data types. They are always side-effect free. A function is defined by using the **def** keyword. Also, functions can have type parameters to abstract from concrete types. For example, the predefined `head` function is over a parametric data type `A` is declared as follows:

**Example:**

```
def A head<A>(List<A> list) = ...
```

*Pattern Matching.* In order to conveniently work with algebraic data types, ABS supports the pattern matching. A *pattern* can be a *bound variable*, in which case it matches against its value, a *free variable*, in which case the variable matches everything and is bound to the matched value, the placeholder _ (underscore) which matches anything, but does not establish a binding, and a *constructor pattern*, in which case the value must match the corresponding constructor. Pattern matching can be defined using the **case** expression. In the following example, pattern matching is used to get the set of all ingredients of a given juice. The data type `Set`, with its constructors `Insert` and `EmptySet`, as well as the function `union` are predefined in ABS.

**Example:**

```
def Set<Fruit> ingredients(Juice juice) =
  case juice {
    Mixed(j1,j2) => union(ingredients(j1),ingredients(j2));
    Pure(fruit) => Insert(fruit,EmptySet);
  } ;
```

## 2.2 Object-Based Programming

*Classes.* ABS models are structured into *classes*. A class declaration consists of the class name, a list of constructor arguments, a list of interfaces that the class implements, a list of fields (instance variables), an init block, and a list of methods. All of these except the class name are optional.

**Example:**

```
class IPing(Pong pong, Int pingCount) implements Ping {
  Int pingsLeft = pingCount; // A field definition

  // The init block contains non−trivial field initializations.
  // All constructor arguments are fields as well and can be used here.
  {
    ...
  }

  // The special run() method is invoked once upon object creation.
  // It specifies the object's active behavior.
  Unit run() {
    while (pingsLeft > 0) {
       pong ! hi("Hello");
       pingsLeft = pingsLeft - 1;
    }
  }
}
```

*Interfaces.* In ABS, classes are not types. Instead, all object references are typed by *interfaces*. An interface declaration consists of the interface name, a list of interfaces that the interface inherits from, a list of methods that have to be implemented by classes implementing the interface. All elements except the interface name are optional. A class has to implement at least the methods that are listed in its interface(s). These methods listed in the implemented interfaces can be called from the outside. All other methods the class implements are private and can only be invoked on `this`. Interfaces do not contain field declarations. Hence, there is no way of accessing fields of an object different from `this`.

**Example:**

```
interface Empty {
  Unit doNothing();
}

class IEmpty implements Empty {
  Unit doNothing() { skip; }
  Unit thisIsPrivate() { skip; }
}
```

*Statements.* ABS supports standard statements and expressions known from object-oriented languages, such as Java. These include: assignments, conditional statements and loops. The most basic statement is **skip**, which does nothing. While not particularly useful, it has its place in empty method bodies of abstract behavioral models which will be concretized later. Variables in ABS are defined in the usual way by giving a type, name and initial value. Variable names must begin with a lowercase letter, followed by a combination of letters, numbers and the underscore (_) character. Variable assignments consist of a left-hand side naming a variable and a right-hand side which can be any type-correct expression.

**Example:**
```
String x = "Hello";
x = x + " World!";
```

The conditional statement has the same syntax as in Java. The conditional expression must be of type `Bool` (i.e., evaluate to `True` or `False`), the consequent and optional alternate parts of the conditional statement are blocks which can introduce local variables.

**Example:**
```
if (contains(ingredients(juice), Banana)) {
  result = "I love bananas!";
} else {
  skip;
}
```

The while loop is standard as well, consisting of a Boolean expression and a block, to be evaluated until the expression evaluates to `False`.

**Example:**
```
while (True) {
  skip; // This loops forever.
}
```

*Modules.* An *ABS Model* is a set of *modules*, where each module is defined in an ABS file, which typically ends with `.abs`. A file can have multiple module definitions, but a single module must be completely defined in one file.

Modules define named scopes for declarations which can be interfaces, classes, or data types, and provide name spaces and a means for implementation hiding. All declarations defined in a module are by default hidden and cannot be used by other modules. In order to make declarations available to other modules, they have to be explicitly *exported*. In order to use declarations of other modules, they have to be explicitly *imported*. The following example shows how names can be exported and imported. Like Java packages, modules in ABS are *flat*. Even though module names are often made hierarchical by using periods, such a structure has no special meaning in ABS.

**Example:**
```
module Example.PingPong.Ping;
export IPing, Ping;
import Pong from Example.PingPong.Pong;
```

A module can have an optional *main block*, which defines how a system is started. The following module has a main block that creates an instance of `IPing` and calls its `start` method. See the following subsection for the **new cog** syntax.

**Example:**

```
module Example.PingPong;
import * from Example.PingPong.Ping;
import * from Example.PingPong.Pong;

{
  Pong pong = new cog IPong();
  Ping ping = new cog IPing(pong,5);
}
```

## 2.3 Concurrency Model

ABS is especially designed for modeling concurrent and distributed systems. The concurrency model of ABS is based on the concept of *Concurrent Object Groups* (COGs). A typical ABS system consists of multiple COGs at runtime. COGs can be regarded as autonomous, runtime components that are executed concurrently and share no state.

*Concurrent Object Groups.* A new COG is created by using the **new cog** expression. It takes as argument a class name, which is the class of the first object of the new COG. The result is a reference to the first object. The following example creates a new COG with an initial object of class `IPong`. The reference to this new `IPong` object is stored in the `pong` variable. The `IPong` object lives in a different COG than the COG which created it.

**Example:**

```
Pong pong = new cog IPong();
```

*Asynchronous Method Calls.* Objects communicate by exchanging messages via method calls. When using a synchronous method call, the caller must wait for the call to be returned. This leads to a strong temporal coupling between the caller and the callee. In a distributed setting, the caller must additionally also wait until the corresponding network message has been sent to the target node, which leads to problems for systems with high latency. ABS contains linguistic constructs for synchronous method calls. However, due to the above reasons, communication between COGs may solely be via *asynchronous method calls*. The difference to the synchronous case is that an asynchronous call immediately returns to the caller without waiting for the message to be received and handled by the callee. Asynchronous method calls are indicated by an exclamation mark (!) instead of a dot.

**Example:**

```
pong ! hi("Hello Pong");
```

In order to ensure that COGs only communicate via asynchronous methods calls, ABS provides a pluggable type extension to statically distinguish *far references* to objects in a different COG and *near references* in the same COG. The used type annotations are `[Near]`, `[Far]`, and `[Somewhere]`, where `[Somewhere]` means that the reference is either near or far. The above example can be then typed as follows.

**Example:**

```
[Far] Pong pong = new cog IPong();
```

As synchronous method calls are not allowed on far references the following code will lead to a runtime error in ABS. When using the additional type annotations, the type checker will catch that error at compile time already.

**Example:**

```
Pong pong = new cog IPong();
pong.hi("Hello"); // runtime error
```

*Futures.* Communication between objects usually follows a *request-response pattern*. If a request is sent via a synchronous method call, eventually the called object sends the return value of the call as return value to the caller. When using asynchronous method calls, the caller does not wait for the result of the call. Instead, the asynchronous method call returns a *future*. A future is a placeholder for the result of the method call. Initially, a future is *unresolved*. When the called method has terminated, the future will (automatically) be *resolved* with the result value of the call. The caller can, thus, obtain the result value of the method call at a later point by using the future.

A future in ABS is represented by the predefined data type **Fut**`<T>` where the type parameter `T` corresponds to the return type of the called method. The following example assigns the result of the above asynchronous method call to a future `answerFut`, where the method `hi` is assumed to have `String` as return type. To get the value of the future `answer`, the **get**-expression can be used.

**Example:**

```
Fut<String> answerFut = pong ! hi("Hello Pong");
String answer = answerFut.get;
```

The **get**-expression only returns the value of the future, when the future is resolved. If the future is unresolved, the control flow is *blocked*, until the future is resolved. Hence, synchronous communication can be simulated in ABS by performing an asynchronous method call and waiting for the resolved future using the **get**-expression.

*Cooperative Multi-Tasking.* The ABS approach to handle concurrency relies on strict data encapsulation and on cooperative multi-tasking on the level of COGs. Strict data encapsulation is achieved since all object fields are private and can only be accessed via method calls. Hence, the state of an object does not have to be protected against modifications from the outside.

COG-level cooperative multi-tasking means that all *tasks*[7] run within the scope of a COG. A method call creates a task in the scope of the target object. For asynchronous methods calls, the calling task can continue to run while the issued method call is processed and get its result at a later point in time. Race conditions between the tasks of the same COG are prevented by cooperative multi-tasking. For example, in conventional programming languages that are based on preemptively scheduled threads, the following code is prone to subtle errors:

**Example:**

```
Unit addToState(Int item) {
  itemCount = itemCount + 1;
  itemList = Cons(item, itemList);
}

Int removeFromState() {
  Int result = 0;
  if (itemCount > 0) {
    itemCount = itemCount - 1;
    result = head(itemList); itemList = removeHead(itemList);
  } else {
    // handle error
  }
  return item;
}
```

If a thread running `addToState` is interrupted after its first statement, a second thread running `removeFromState` might try to remove an element from an empty `itemList` which is a typical race condition. In languages like Java race conditions can only be prevented by explicitly synchronizing threads using locks or synchronized blocks.

ABS solves this problem by scheduling tasks only at specific *scheduling points* during program execution which are apparent in the source code. Hence, a COG state is implicitly protected, except at certain points that can be syntactically identified and analyzed. The **suspend** statement introduces a scheduling point, allowing the running task to be suspended and another task of the COG to be scheduled.

[7] Tasks correspond to threads, known from languages such as Java, but are scheduled cooperatively instead of preemptively.

**Example:**

```
// This loops forever, blocking the COG
while (True) { skip; }

// This loops forever, in parallel with other tasks
while (True) { suspend; }
```

With the **await** statement, one can create a conditional scheduling point, where the running task is suspended, until the specified condition becomes true.

**Example:**

```
Bool flag = False;

Bool waitUntilTrue () {
  await flag; // we rely on some other task to set the flag for us
  return flag; // will always return True
}
```

The **await** statement is also a way to synchronize with the future of an asynchronous method call without blocking the entire COG.

**Example:**

```
  Fut<String> answerFut = ping ! hi("Hello Ping");
  skip; // do some processing ...
  await answerFut?;
  String answer = answerFut.get; // guaranteed not to block
```

A method for reasoning about absence of race conditions in ABS is to inspect each **suspend** and **await** statement, and check if the task at this point leaves the COG in an orderly state (i.e., establishes the COG invariant). At all other points, the COG is implicitly protected against concurrent modifications.

## 3 Spatial Variability Modeling

Spatial variability captures different variants of a software products coexisting at the same point in time. This variability can often be phrased in terms of the *features* offered by the software. Finer-grained configuration parameters are represented using *attributes* of features. Software product line engineering [18] aims at developing similar product variants by reuse. The ABS incarnation of this approach is realised by four languages ($\mu$TVL, DML, CL, PSL) on top of core ABS. These languages express spatial variability at the level of product features and as changes to the behavior of a core product, along with providing the configuration of the product line artifacts, and the ultimate selection of a product via the specification of the relevant features and their attributes.

The feature description language $\mu$TVL 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 just a name. Attributes represent micro-variability within features. DML is used to specify delta modules which, when selected, are used to modify a core ABS model. Delta modules implement spatial variability at the level of core ABS. Although DML delta modules are used to implement features, they are written independently of any feature. They are, in fact, reusable for different ABS product lines as they may be written independently of a specific application context. CL specifications link $\mu$TVL feature models with the DML delta modules that implement the corresponding behavioral modifications. A CL specification provides *application conditions* for each delta module, which are constraints over features and their attributes governing when the delta module is applicable. A CL specification also specifies constraints on the ordering of applicable delta modules, in order to avoid potential ambiguity in different delta application orders. A PSL script consists of two parts, namely, a specification of the features and their attributes selected for a product and an initialization code block, which typically is just a call to an appropriate *main* method, though it may contain additional configuration. The feature selection part of a PSL specification is checked against a $\mu$TVL feature model, and is also used to determine which delta modules to apply, namely, those whose application condition is true given the feature selection. To generate the product specified by the PSL script, all deltas with valid application condition are applied to the core ABS model in some order compliant with the order specified in the CL script, and then the initialisation block is added to the core program.

The following application, the core of a multilingual "Hello World" program, will be used to illustrate these languages and their interaction. Here is the core of the `MultiLingualHelloWorld` product line in core ABS:

**Example:**

```
interface Greeting {
  String say_hello();
}
class Greeter implements Greeting {
  String say_hello() {
    return "Hello world";
  }
}
class Application {
  Unit main() {
    Greeting bob;
    bob = new Greeter();
    String s = "";
    s = bob.say_hello();
  }
}
```

Interface `Greeting`, class `Greeter` and class `Application` form the core modules of the ABS implementation.

### 3.1 Feature Modeling

$\mu$TVL is a feature modelling language, pronounced either *micro textual variability language* or simply *mu tee vee ell*. It is an extended subset of TVL [5, 6], which was developed to serve as a reference language for specifying feature models. $\mu$TVL is textual, as opposed to diagrammatic, and aims to be scalable, concise, modular, and comprehensive. A feature model is represented textually as a forest of nested features, each with a collection of boolean or integer attributes. Additional cross-tree dependencies can be expressed in the feature model. $\mu$TVL allows a feature model with multiple roots (hence, multiple trees) to express orthogonal variability [18], which is useful for representing application or platform models in an orthogonal fashion.

The grammar of $\mu$TVL is given in Fig. 2. FID is the set of valid feature names, and AID of valid attribute names. Attributes and values in $\mu$TVL range either over integers or over booleans. Extensions to include other data types is unproblematic, as long as any relevant constraints can be encoded by integer constraints.

```
           Model ::= (root FeatureDecl)* FeatureExtension*

     FeatureDecl ::= FID [{ [Group] AttributeDecl* Constraint* }]
FeatureExtension ::= extension FID { AttributeDecl* Constraint*}

           Group ::= group Cardinality { [opt] FeatureDecl, ([opt] FeatureDecl)* }
     Cardinality ::= allof | oneof | [n1 .. *] | [n1 .. n2]
   AttributeDecl ::= Int AID ; | Int AID in [ Limit .. Limit ] ; | Bool AID ;
           Limit ::= n | *

      Constraint ::= Expr ; | ifin: Expr ; | ifout: Expr ;
                     | require: FID ; | exclude: FID ;
            Expr ::= true | false | n | FID | AID | FID.AID
                     | UnOp Expr | Expr BinOp Expr | ( Expr )
            UnOp ::= ! | -
           BinOp ::= || | && | -> | <-> | == | != | > | < | >= | <= | + | - | * | / | %
```

Fig. 2: Syntax of $\mu$TVL ($n$ ranges over integers)

The *Model* clause specifies a number of "orthogonal" root feature models and some extensions. A root feature model, *FeatureDecl*, contains the name of a feature (FID), followed by a specification of any sub-features, the feature's attributes and any relevant constraints. Extensions, *FeatureExtension*, specify additional attributes and constraints, typically cross-tree dependencies. The *Group* clause specifies the sub-features of a feature. This consists of a specification of the cardinality of the group, plus a number of possibly optional sub-features. The *Cardinality* clause describes the number of elements of a group that may appear in a result. Keyword `allof` means that all elements of the group must appear. Keyword `oneof` means that one element must appear. Range descriptions `[`$n_1$ `..` $*$`]` and `[`$n_1$ `..` $n_2$`]` specify the range of values on the number of elements of the group. These can be bounded below and above or unbounded above ($*$). Zero or one instances of each feature can be present in the ultimate model—this means that cardinality specifies not that features can be multiply instantiated, rather it specifies the number of selections that can be made for a choice: by analogy, $\{Apple, Banana\}$ is a valid choice of 2 elements from the set $\{Apple, Orange, Banana\}$, whereas $\{Orange, Orange\}$ is not.

The *AttributeDecl* clause specifies both integer (bounded or unbounded) and boolean attributes of features. The *Limit* clause is used to specify the bounds, where $n$ is some integer and $*$ indicates that an attribute is unbounded below and/or above.

The *Constraint* clause specifies constraints on the presence of features and on attribute values. An `ifin` constraint is only applicable if the current feature is selected. Similarly, an `ifout` constraint is only applicable if the current feature is not selected. An `include` clause specifies that the current feature requires some other feature, whereas `exclude` expresses the mutual incompatibility between the current feature and some other feature. The *Expr* clause ultimately expresses a boolean constraint over the presence of features and attribute values. Features are referred to by identity (FID). Attributes are referred to either using an unqualified name (AID), for in scope attributes, or using a qualified name (FID.AID) for attributes of other features. Unary, *UnOp*, and binary operators, *BinOP*, are standard.

*Example 1.* The following is a feature model for the `MultiLingualHelloWorld` product line, which describes software that outputs *"hello world"* in multiple languages some number of times.

```
root MultiLingualHelloWorld {
  group allof {
    Language {
      group oneof { English, Dutch, French, German }
    },
    opt Repeat {
      Int times in [0..1000];
      times > 0;
    }
  }
}

extension English {
  ifin: Repeat ->
        (Repeat.times >= 2 && Repeat.times <= 5);
}
```

This feature model introduces two core features, `Language` and `Repeat`. The `Language` feature corresponds to one of four possible features: `English`, `Dutch`, `French`, or `German`. The `Repeat` feature has no sub-features, and it has an attribute `times` with range from 0 to 1000, with an added condition that it must be strictly greater than 0. The extension for the `English` feature states that when the `English` and `Repeat` features are both present, the attribute `times` must be between 2 and 5, inclusive.

This feature model can be depicted as a feature diagram using standard notations [7], as shown in Fig. 3.

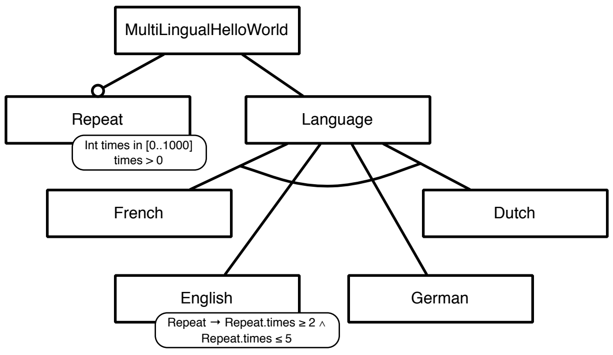


Fig. 3: Feature Diagram for the *MultiLingualHelloWorld* example

## 3.2 Delta Modeling

Variability at the level of abstract behavioral specifications (or source code) is achieved using delta modeling. The concept of delta modeling was introduced by Schaefer et al. [19, 20, 22, 21] as a novel modeling and programming language approach for software-based product lines, and can be seen as an direct alternative to feature-oriented programming [2]. 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 delta-oriented programming [20], *application conditions*, conditions over the set of features and their attributes, are associated with modules of program modifications (add, remove, or otherwise modify code), called *delta modules*. The implementation of a software product line in delta-oriented programming is divided into a *core module* and a set of delta modules. The core module consists of the classes that implement a complete product of the corresponding product line. 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. For representing spatial variability in the ABS language, we adapt these ideas to ABS models.

$$\begin{array}{rcl}
\mathit{DeltaDecl} & ::= & \texttt{delta}\ \mathit{TypeId}\ [\mathit{DeltaParamDecls}] \\
 & & \qquad\qquad \{\mathit{ClassOrInterfaceModifier}^{*}\ \} \\
\mathit{ClassOrInterfaceModifier} & ::= & \texttt{adds}\ \mathit{ClassDecl} \\
 & | & \texttt{modifies class}\ \mathit{TypeId}\ \mathit{ImplementsModifier}^{*} \\
 & & \qquad\qquad \{\ \mathit{Modifier}^{*}\ \} \\
 & | & \texttt{removes class}\ \mathit{TypeId}\ \texttt{;} \\
 & | & \texttt{adds}\ \mathit{InterfaceDecl} \\
 & | & \texttt{modifies interface}\ \mathit{TypeId}\ \mathit{ImplementsModifier}^{*} \\
 & & \qquad\qquad \{\ \mathit{Modifier}^{*}\ \} \\
 & | & \texttt{removes interface}\ \mathit{TypeId}\ \texttt{;} \\
\mathit{ImplementsModifier} & ::= & \texttt{adds}\ \mathit{TypeId} \\
 & | & \texttt{removes}\ \mathit{TypeId} \\
\mathit{Modifier} & ::= & \texttt{adds}\ \mathit{FieldDecl} \\
 & | & \texttt{removes}\ \mathit{FieldDecl} \\
 & | & \texttt{adds}\ \mathit{MethDecl} \\
 & | & \texttt{modifies}\ \mathit{MethDecl} \\
 & | & \texttt{removes}\ \mathit{MethSig} \\
\mathit{DeltaParamDecls} & ::= & \texttt{(}\mathit{DeltaParamDecl}\ (\texttt{,}\ \mathit{DeltaParamDecl})^{*}\ \texttt{)} \\
\mathit{DeltaParamDecl} & ::= & \mathit{Identifier}\ \mathit{HasCondition}^{*} \\
 & | & \mathit{Type}\ \mathit{Identifier} \\
\mathit{HasCondition} & ::= & \texttt{hasField}\ \mathit{FieldDecl} \\
 & | & \texttt{hasMethod}\ \mathit{MethSig} \\
 & | & \texttt{hasInterface}\ \mathit{TypeId}
\end{array}$$

Fig. 4: Syntax of Delta Modules

Fig. 4 specifies the syntax of delta modules over core ABS models. The grammar uses nonterminals from the core ABS language, indicated in purple (gray). Their names should be sufficiently suggestive.

The *DeltaDecl* clause specifies the syntax of delta modules, which consists of an unique identifier, a list of parameters, and a body containing a sequence of class and interface modifiers. The *ClassOrInterfaceModifier* clause describes the syntax of modifications at the level of classes and interfaces. Such a modification can add a class or interface declaration, modify an existing class or interface, or remove a class or interface. The *ImplementsModifier* clause describes how to modify the interfaces a class implements or extends, either by adding new or removing existing interfaces.

The *Modifier* clause specifies the modifications that can occur within a class or interface body. These include (where relevant) adding and removing fields and method signatures (from interfaces), and modifying methods, which amounts to replacing a method with a new implementation, but enabling the original method to be called using the **original**`()` keyword. The semantics of calling **original**`()` is essentially the same as `Super()` from feature-oriented programming [2], and `proceed` from context-oriented programming [13], and similar to ordinary `super` calls in standard object-oriented languages, as well as the `around` advice (without quantification) from aspect-oriented programming [16].

Delta modules in the ABS language can be parameterised both by attribute values and by class names, to enable the application of a single delta in more than one context. This is in contrast to delta modules presented in the literature [20, 22, 21], which are unparameterised. The *HasCondition* describes constraints on class arguments to which a delta may be applied. These constraints state, for instance, the methods and fields that a class or interface is expected to have.

Below are some delta modules for the `MultiLingualHelloWorld` product line defined above which represent the `Repeat`, `German` and `Dutch` features. A delta module for the `French` feature can be implemented in a similar fashion.

**Example:**

```
delta Rpt (Int times) {
  modifies class Greeter {
    modifies String say_hello() {
      String result = "";
        Int i = 0;
        while (i < times) {
          result = result + original();
          i = i + 1;
        }
        return result;
     }
  }
}
delta De {
  modifies class Greeter {
    modifies String say_hello() {
      return "Hallo Welt";
    }
  }
}
delta Nl {
  modifies class Greeter {
    modifies String say_hello() {
      return "Hallo wereld";
    }
  }
}
```

The delta module `De`, for example, implements a single class modifier for `Greeter`, which in turn has a single method modifier. This method modifier replaces the method `say_hello()` of class `Greeter` to return the German text "Hallo Welt". The delta module `Rpt` has a single parameter for the number of times that the hello string should be repeated. This delta module replaces the method `say_hello()` in class `Greeter` with new ABS code. However, in this case, the code being replaced is also included via the special method call **original**`()`.

## 3.3 Product Line Configuration

The product line configuration language (CL) links $\mu$TVL feature models with DML delta modules to provide a complete specification of the spatial variability in an ABS product line [22, 21]. A product line configuration script consists of the set of features assumed to exist 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 specified by after clauses constrains the order in which delta modules can be applied to the core module. The syntax of the product line configuration language is given in Fig. 5.

```
       Configuration ::= productline Name { Features ; DeltaClauses }

            Features ::= features FID ( , FID )*

        DeltaClauses ::= DeltaClause ( , DeltaClause)*
         DeltaClause ::= delta DeltaSpec [AfterCondition] [ApplicationCondition] ;

           DeltaSpec ::= Name [( DeltaArguments )]
      DeltaArguments ::= DeltaArgument ( , DeltaArgument)*
       DeltaArgument ::= FID | FID.AID | PureExpr

      AfterCondition ::= after Name ( , Name )*
ApplicationCondition ::= when PureExpr
```

Fig. 5: CL grammar

The *Configuration* clause specifies the name of the product line, the set of features it provides using a *Features* clause, and the set of delta modules used to implement those features by *DeltaClause* clauses. In a *DeltaClause*, the *AfterCondition* clause specifies the delta modules that the current delta module must be applied after. The *DeltaSpec* clause names a specific delta module and specifies any parameters that need to be passed. These parameters are either attributes from the feature model or constant values specified in core ABS as a *PureExpr*, that is, an expression without side effects. The *ApplicationCondition* clause specifies a predicate describing under which feature configurations the given delta module is applied. This condition is phrased in terms of the presence and absence of features and feature combinations, as well as using attributes of features and integer and boolean constants.

The `MultiLingualHelloWorld` product line is configured as follows:

**Example:**

```
productline MultiLingualHelloWorld {
  features English, German, French, Dutch, Repeat;

  delta Rpt(Repeat.times) after De, Fr, Nl when Repeat;
  delta De when German;
  delta Fr when French;
  delta Nl when Dutch;
}
```

The example CL configuration script specifies, for example, that the `De` delta module is applicable when the `German` feature is selected. Note that there is no delta module corresponding to the feature `English`, as the core provides support for the English feature as a default. In addition, `Rpt` is configured such that it has to be applied **after** all the language-specific delta modules. The delta module `Rpt`'s argument is filled with the `times` attribute of feature `Repeat`, which ultimately will be specified in a PSL script (Section 3.4).

## 3.4 Product Selection and Generation

To generate a product from an ABS product line, a product selection is specified using the *product selection language* (PSL). A product selection states which features are to be included in the product and specifies concrete values for their attributes. In addition, some core ABS code is provided to initialise the selected product. A product selection is checked against a $\mu$TVL feature model for validity, possibly after adding any implied features. An implied feature is a parent feature in the $\mu$TVL feature model. Then, the product selection is used by the configuration file to guide the application of the delta modules during the generation of the final software product. Fig. 6 specifies the grammar of PSL.

```
            Selection ::= product Name ( FeatureSpecs ) { InitBlock }

         FeatureSpecs ::= FeatureSpec (, FeatureSpec)*
          FeatureSpec ::= FID [AttributeAssignments]

 AttributeAssignments ::= { AttributeAssignment (, AttributeAssignment)* }
  AttributeAssignment ::= AID = Value

            InitBlock ::= { Core ABS code }
```

Fig. 6: PSL grammar

The *Selection* clause specifies a product by giving it a name, by stating the features (*FeatureSpec*) to be included in the product and the concrete values for their attributes (*AttributeAssignment*) and by specifying an initialisation block (*InitBlock*). This can be any core ABS code, but typically, it will be a simple call to some already present *main* method. Initialisation blocks are specified in the product selection language to enable product lines with multiple entry points to the code base.

Here are some candidate product selections for the `MultiLingualHelloWorld` product line:

**Example:**

```
// basic product with no deltas
product P1 (English) {
  Application.main();
}

// apply delta Fr
product P2 (French) {
  Application.main();
}

// apply deltas De and Repeat
product P3 (German, Repeat{times=10}) {
  Application.main();
}

// apply delta Repeat to core with feature English, but the application should
// be refused because "times > 5"
product P4 (English, Repeat{times=6}) {
  Application.main();
}
```

The example specifies four products: `P1`, `P2`, `P3`, and `P4`. In the case of the product `P1`, the parameter `English` means the product consists of this feature and of the features implied by the feature model. In this case, the implied features are `Language` and the root `MultiLingualHelloWorld`, according to the feature model in Example 1. In `P3` and `P4`, the parameters also include attribute values, in these cases assigning a value to the attribute `times` of the `Repeat` feature. The block of ABS code associated to each product provides its initialisation code. Every product in our example executes the main method of the class `Application`, which is included in the core module.

## 3.5 Product Generation

Given a core ABS program $P$, a set of delta modules $\Delta$, a product line configuration $C$, a feature model $FM$, and a product selection $p$, the final software product, which will be a core ABS program, is derived as follows:

- first, check that the product selection $p$ satisfies the constraints imposed by the feature model $FM$;
- second, select the delta modules from $\Delta$ with a valid application condition with respect to $p$;
- third, apply the delta modules to the core program $P$ in some order respecting the partial order described in $C$; and
- finally, add the initialisation block to the resulting ABS code.

The selection of product `P3` in our running example results in the following Core ABS model. The parameter `times` from feature `Rpt` has been replaced by the value 10, as specified in the product selection. Class `Application` and interface `Greeting` are as above in the core module. The **original**`()` call to the previous version of the `say_hello` method is replaced with a call to a renamed version of the unchanged method.

**Example:**

```
class Greeter implements Greeting {
  String say_hello_original() {
    return "Hallo Welt";
  }
  String say_hello() {
    String result = "";
    Int i = 0;
    while (i < 10) {
      result = result + say_hello_original();
      i = i + 1;
    }
    return result;
  }
}
{ // initialisation block
  Application.main();
}
```

## 4 Temporal Variability Modeling

The basic idea of temporal variability modeling in ABS is to capture how products in an ABS product line safely evolve over time. Evolution takes place during system execution in order to accomodate necessary changes after the deployment of the products; e.g., bug fixes, feature extensions or modifications, or changes in user requirements. The main design target of ABS models are concurrent and distributed systems in which objects communicate asynchronously. Consequently, it is not straightforward to halt a deployed product during execution in order to let it evolve. Rather, temporal evolution must happen asynchronously at runtime. In ABS, we follow an approach developed for class-based inheritance for distributed concurrent objects [14], but we adapt it to delta modeling.

### 4.1 Dynamic Delta Modules

In order to facilitate the modeling of temporal variability for high-level ABS models, it is crucial that evolution is expressed at the abstraction level of the modeling language. Therefore, temporal variability is captured by a *series of asynchronous changes* to the executing product, where each change addresses one of the structuring concepts of the modeling language and where the series of changes together bring about the desired overall modification of product behavior. In ABS, the main structuring concepts for system variability are a designated

```
             DynDeltaDecl ::= dyndelta TypeId [DeltaParamDecls]
                                  { (DeltaModifier | ClassOrInterfaceModifier)* }

            DeltaModifier ::= adds delta DeltaDecl
                            | modifies delta DeltaDecl

                DeltaDecl ::= TypeId [DeltaParamDecls]
                                  {ClassOrInterfaceModifier* }

ClassOrInterfaceModifier ::= adds ClassDecl
                            | adds InterfaceDecl
                            | modifies class TypeId ImplementsModifier*
                                  { Modifier* }
                            | simplifies class TypeId
                                  { Simplifier* }

      ImplementsModifier ::= adds TypeId

                 Modifier ::= adds FieldDecl
                            | adds MethDecl
                            | modifies MethDecl

               Simplifier ::= removes FieldDecl
                            | removes MethDecl
```

Fig. 7: Syntax for dynamic delta declarations in ABS (the clause for *DeltaParamDecls* is defined in Fig. 4)

core module and delta modules, which contain modifier operations for classes, interfaces, field and method declarations (see Fig. 4).

Temporal variability in ABS is expressed in terms of *dynamic delta modules*. Dynamic delta modules can add classes or interfaces to the core module or modify classes or interfaces that are already contained in the core module. Additionally, dynamic delta modules can change delta modules, meaning they can add classes or interfaces which are in turn added by the delta modules or alter the contained modification operations. The grammar for dynamic delta modules is given in Fig. 7. Dynamic delta modules allow us to add new class and interface declarations to the core module or delta modules, and to change existing class declarations in the following ways:

– add new fields and method definitions
– modify existing method definitions
– simplify the class by removing redundant field and method declarations

Thus, the dynamic delta modules are slightly more restrictive than standard delta modules used for spatial variability. In particular, classes and interfaces cannot be removed, and the modifiers are distinguished from the simplifiers (in Fig. 4, the simplifiers are in the same syntactic category as the modifiers).

In order to illustrate the usage of dynamic delta modules, the `MultiLingual HelloWorld` product line is modified to make a more personal greeting. The dynamic delta module `ExtendedGreeting` changes the `Greeter` class that is defined in the core module by adding a method `i_am_bob()` which returns a personalized message and modifies the `say_hello()` method to output this message. The delta modules `De` and `Nl` are modified to include a translated version of the `i_am_bob()` method:

**Example:**

```
dyndelta ExtendedGreeting {
  modifies class Greeter { // Adds a new method to the class Greeter
    adds String i_am_bob () { return ", I am Bob!"; }
    modifies String say_hello() {
         return = original() + this.i_am_bob();
     }
   }

modifies delta De {
 modifies class Greeter {
  modifies String i_am_bob() {return ", ich bin Bob!";}
  }
 }

modifies delta Nl {
 modifies class Greeter {
  modifies String i_am_bob() {return ", ick ben Bob!";}
  }
 }
}
```

## 4.2 Restrictions on Dynamic Delta Modules

The rationale for the syntactic restrictions introduced in the dynamic delta modules is to guarantee that the temporal evolution of the product's code does not give rise to runtime errors. This guarantee is provided by a static analysis which identifies runtime applicability conditions for the different elements of the dynamic delta module. In order to apply a change to a class, other changes may be required to have taken place already. In the asynchronous setting of ABS, this cannot be statically controlled. For example, if new code in a change to a class $D$ invokes a method on an instance of another class $C$, that method must be available. If the method is introduced as a change to $C$, the static analysis of the change to $D$ will generate an applicability condition to require that this change to $C$ has already occurred at runtime. The distinction between modifiers and simplifiers corresponds to the distinction between runtime applicability conditions at the level of classes and at the level of instances of those classes [14]. The removal of classes, as well as the removal and modification of interfaces, are disallowed because they would require a inspectionvof all references of the involved interfaces (including references in messages), which is a very heavy operation in the asynchronous distributed setting targeted by ABS.

## 5 Case Study

In this section, we present a case study of spatial and temporal variability modeling with the ABS based on the Fredhopper Access Server (FAS) [11]. In particular, we focus on its Replication System component. First, we describe how we model the core components of the existing Java implementation of the FAS replication system using Core ABS. Second, we capture some of the system's spatial variabilities using the languages presented in Section 3. Third, we illustrate how to represent temporal variability, including changing of available features. Finally, we describe how the HATS tools suite assists our modeling activities.

### 5.1 Fredhopper Access Server

The Fredhopper Access Server (FAS) is a service-oriented, server-based software system, which provides search and merchandising IT services to e-Commerce companies, such as large catalogue trading, travel booking, or classified advertising, etc. Each FAS installation is deployed to a customer according to the FAS deployment architecture. Fig. 8 shows an example setup.

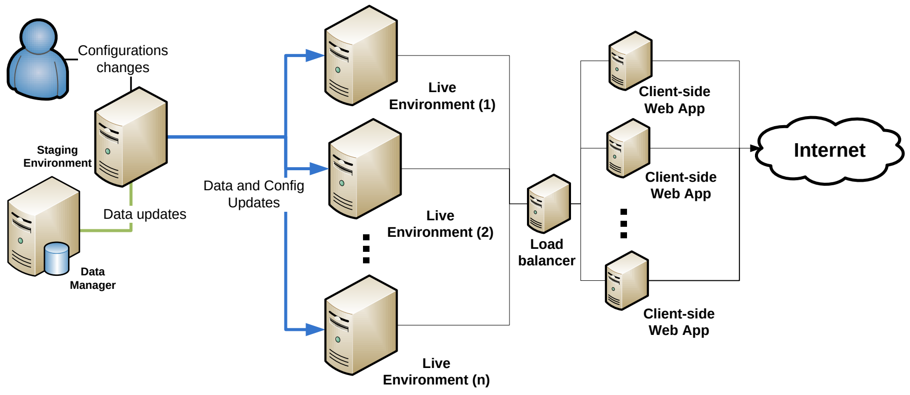


Fig. 8: Example of FAS deployment

A FAS deployment consists of a set of live environments and a staging environment. A live environment processes queries from client web applications via web services. The 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. A more detailed description of the replication system can be found in [8].

For the purpose of this case study, we focus on the synchronisation client component in the live environment and the synchronisation server component in the staging environment. The synchronisation server and a set of synchronisation clients together constitute the replication system. The synchronization server distributes configuration and data updates to the synchronization clients running on the live environments. It is responsible for determining the schedule of replication as well as the content of each replication item for its staging environment. A replication item represents a single unit of replicable data. It is either a file directory, a set of files whose name matches a regular expression or a database journal. A replication snapshot is a set of replication items. The synchronisation client connects to the synchronisation server component in the staging environment and responds to incoming updates resulting from changes to data and configuration.

The synchronisation server and clients do not communicate directly. The synchronisation server creates connection threads that serve as the interface to the server-side of the replication protocol. In the existing implementation, connection threads are Java thread objects. The synchronisation client, on the other hand, schedules client jobs to handle communications to the client-side of the replication protocol. In the existing implementation, a client job is a Java object that is scheduled using a third party scheduling library. The synchronisation server and clients communicate via connection threads and client jobs asynchronously via sockets.

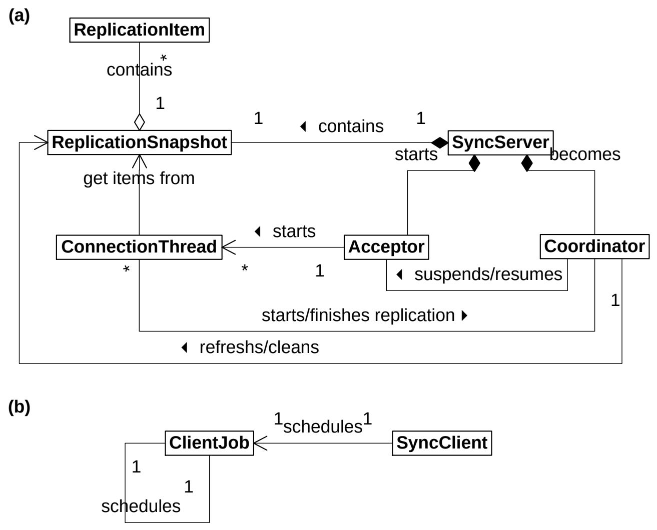


Fig. 9: Class diagram of (a) synchronisation server and (b) synchronisation client

Fig. 9(a) and (b) show the UML class diagram of the synchronisation server and client respectively. The synchronisation server consists of the following components: an acceptor, one or more connection threads, a coordinator, a SyncServer and a replication snapshot. The synchronisation client, on the other hand consists of a SyncClient and one or more client jobs.

## 5.2 Modeling the Replication System with Core ABS

We now provide a description of the individual components of the replication system in our ABS model. Listing 1.1 shows the ABS interfaces for the components of the synchronisation server. For brevity, we have omitted ABS class definitions. More details on the ABS model described in this section can be found on the HATS project website http://www.hats-project-eu.

```
interface Command { Unit command(Command command); }
interface ConnectionThread extends Command { }
interface Node { DataBase getDataBase(); }
interface ServerNode extends Node { Set<Schedule> getSchedule(); }

interface Acceptor {
  [Far] ConnectionThread getConnection(ClientJob job);
  Bool isAcceptingConnection();
  Unit suspendConnection();
  Unit resumingConnection(); }

interface Coordinator {
  Unit process();
  Unit startUpdate(ConnectionThread worker);
  Unit finishUpdate(ConnectionThread worker); }

interface SyncServer extends ServerNode {
  Acceptor getAcceptor();
  [Far] Coordinator getCoordinator();
  [Near] ReplicationSnapshot getReplicationSnapshot(); }

interface ReplicationSnapshot {
  Unit refreshSnapshot(Bool refreshSnapshot);
  Unit clearSnapshot();
  Int getIndexingId();
  Set<ReplicationItem> getItems();
  Bool hasUpdated(); }

interface ReplicationItem {
  FileEntry getContents();
  ReplicationItemType getType();
  FileId getAbsoluteDir();
  Unit refresh();
  Unit cleanup(); }
```

Listing 1.1: ABS interfaces of the synchronisation server components

The *Acceptor* component is responsible for accepting connections from the synchronisation clients and is specified by interface `Acceptor` shown in Listing 1.1. The interface provides a method for a client job to obtain a reference to a connection thread, as well as methods to enable and disable the synchronisation server to accept a new client job connection. The connection thread and the client job are specified by interfaces `ConnectionThread` and `ClientJob` in Listing 1.1 and Listing 1.2, respectively.

```
interface SyncClient extends Node {
  [Far] Acceptor getAcceptor();
  ClientDataBase getClientDataBase();
  Unit becomesState(State state);
  Unit setAcceptor(Acceptor acceptor); }

interface ClientJob extends Command {
  Bool registerReplicationItems(CheckPoint checkpoint);
  Maybe<FileSize> processFile(FileId id);
  Unit processContent(File file);
  Unit receiveSchedule(); }
```

Listing 1.2: ABS interfaces of the synchronisation client components

Each *connection thread* is instantiated by the `Acceptor`. After the `Acceptor` accepts a connection from a client job, it instantiates a `ConnectionThread` to carry out the replication protocol. The connection thread is specified by interface `ConnectionThread` shown in Listing 1.1. `ConnectionThread` exposes a single method `command()`, which is asynchronously invoked by `ClientJob` objects to determine the current state of a replication.

The *Coordinator* is responsible for coordinating when the `Acceptor` may accept connections from synchronisation clients. This component also provides methods for preparing replication items before a replication session and clearing them afterwards. The coordinator is specified by interface `Coordinator` shown in Listing 1.1.

The *SyncServer* starts the `Acceptor` and the `Coordinator`. It also keeps a reference to the relevant replication snapshot. The synchronization server is specified by the interface `SyncServer` shown in Listing 1.1.

Listing 1.2 shows the ABS interfaces of the components that are part of the synchronization client. The *SyncClient* communicates with the `SyncServer` via job scheduling. At initialisation time, the `SyncClient` schedules a client job to acquire a replication schedule from the server. Using this schedule, this client job creates a new client job for performing the actual replication. Each client job thereafter is responsible to request replication schedules and set up the subsequent jobs for further replication.

Each *client job* receives replication items from a connection thread and updates the synchronisation client's files (configuration and data). The client job is specified by interface `ClientJob` shown on Listing 1.2. Client jobs may be scheduled either sequentially or concurrently.

Listing 1.3 shows an example main block of the replication system. In this main block, first an exemplary set of changes to the data in the synchronisation server is defined. In our ABS model, the file system is structured as a tree, where non-leaf nodes are directories. The changes to a file directory are specified by the map `items`, whose keys are `CheckPoint` values identifying particular sets of changes. Each key points to a set of `File` values representing updates to those files. A file is identified by its fully qualified name, specifying the path through the directory tree to the file content. As simplification, the file content is denoted by an integer value representing its size. For example, in the listing the key `1` points to updates on files located at `dir1/file` and `dir2/file2`.

```
{
  Map<CheckPoint,Map<FileId,FileContent>> items =
   map[Pair(1,map[file("dir1/file1",1),file("dir2/file2",2)]),..];

  Set<Schedule> schedules =
   set[FileItem("dir2","dir2/dir21"),..];

  Set<ClientId> cids = set[0,..];
  Set<[Far] SyncClient> syncclients = EmptySet;
  Set<ClientId> iterator = cids;
  while (hasNext(iterator)) {
    Pair<Set<ClientId>,ClientId> nt = next(iterator);
    SyncClient syncclient = new cog SyncClientImpl(snd(nt));
    syncclients = insertElement(syncclients,syncclient);
    iterator = fst(nt); }

  SyncServer syncserver =
    new cog SyncServerImpl(items,schedules);

  Fut<Acceptor> acc = syncserver!getAcceptor(); await acc?;
  [Far] Acceptor acceptor = acc.get;

  Set<SyncClient> clientIterator = syncclients;
  while (hasNext(clientIterator)) {
    Pair<Set<SyncClient>,SyncClient> nt = next(clientIterator);
    SyncClient syncclient = snd(nt);
    syncclient!setAcceptor(acceptor);
    clientIterator = fst(nt); }
}
```

Listing 1.3: An example main block

The main block defines a set of schedules for replication and instantiates a corresponding set of synchronisation clients, each as a separate COG. Additionally, a synchronisation server is instantiated in another COG, and a reference to its acceptor class is obtained. Afterwards, the main block passes this reference to the clients, which triggers the replication protocol. By instantiating all clients and the server as separate COGs, all connection threads and client jobs belong to separate COGs and, thus, can only communicate via asynchronous method calls.

## 5.3 Spatial Variability of the Replication System

The replication system can exist in several variants. Listing 1.4 shows the corresponding feature model. The replication system has a feature `JobProcessing`, which requires an alternative choice between the two features `Seq` and `Concur`, capturing the choice between sequential and concurrent client job processing, respectively. Additionally, the replication system has a feature *ReplicationItem* which allows choosing between three replication item types represented by the features `Dir`, `File` and `Journal`. The `Dir` feature is mandatory, that is, all versions of the replication system support replicating complete file directories. Moreover, the `Journal` feature requires the feature `Seq` which means that variants of the replication system that support database journal replication may only schedule client jobs sequentially.

```
root ReplicationSystem {
  group allof {
    JobProcessing {
      group oneof { opt Seq, opt Concur }
    },
    ReplicationItem {
      group [1..*] {
        Dir, opt File, opt Journal { require: Seq; }
      }
    }
  }
}
```

Listing 1.4: Feature model of the replication system in $\mu$TVL

The core model of the replication system supports sequential client job processing. This functionality is implemented by the active class `ClientJobImpl`. A partial ABS class definition of `ClientJobImpl` is shown in Listing 1.5. Each instance of `ClientJob` initialises the Boolean field `newJob` to `False` and invokes its `run` method. This method in turn invokes `scheduleNewJob()` asynchronously. The method `scheduleNewJob()` waits for field `newJob` to become `True` before creating a new instance of `ClientJob`. Setting `newJob` to `True` at the end of the `run` method ensures that each client job is scheduled sequentially. The method `becomeState()` is invoked synchronously at specific points inside the `run` method to ensure that, while scheduling client jobs sequentially, the synchronisation client follows a predefined *client state machine* [8].

The lower half of Listing 1.5 defines the delta module `Concurrent` which specifies a single class modifier for class `ClientJobImpl` that has two method modifiers. The first modifier removes the await statement from `scheduleNewJob()`

in such a way that a new instance of `ClientJob` is created as soon as the current `ClientJob` instance releases the lock of this object group. This potentially allows scheduling client jobs concurrently. The second modifier updates method `becomeState` so that the synchronisation client is not required to follow the client state machine which only applies to sequential scheduling.

```
class ClientJobImpl([Far] InternalClient client, JobType job)
implements ClientJob {
  Bool newJob = False;
  Unit scheduleNewJob() {
    await newJob;
    new ClientJobImpl(this.client, Replication);
  }
  Unit run() { .. this!scheduleNewJob(); .. newJob = True; .. }
  Unit becomeState(State state) { .. }
  ..
}

delta Concurrent {
  modifies class ClientJobImpl {
    modifies Unit scheduleNewJob() {
      new ClientJobImpl(this.client, Replication);
    }
    modifies Unit becomeState(State state) { .. }
  }
}
```

Listing 1.5: Core module and delta module for job processing

Listing 1.6 shows a partial definition of the classes `DirectoryItem` and `ReplicationSnapshotImpl`. The class `DirectoryItem` defines a replication item for a complete file directory and the class `ReplicationSnapshotImpl` implements `ReplicationSnapshot`. The method `replicationItem` defined in the class `ReplicationSnapshotImpl` takes a replication schedule, creates a corresponding `ReplicationItem` object and adds it to the set of replication items. By default, this method only handles replication schedules for complete file directories.

```
class DirectoryItem(FileId qualified, ServerDataBase db)
implements ReplicationItem { .. }

class ReplicationSnapshotImpl(
  ServerDataBase db, Set<Schedule> schedules)
  implements ReplicationSnapshot {

  Set<ReplicationItem> items = EmptySet;

  Unit replicationItem(Schedule schedule) {
    if (isSearchItem(schedule)) {
      FileId qualified = left(item(schedule));
      ReplicationItem item = new DirectoryItem(qualified,this.db);
      this.items = Insert(item,this.items);
    }
  }
  ..
}
```

Listing 1.6: Partial core implementation of replication item

In Listing 1.7, two delta modules are depicted that implement the necessary functionality for other types of replication items. The delta module `File` is applied for handling file set replication and has two class modifiers. The first modifier adds class `FilePattern`, an implementation of interface `ReplicationItem` handling replicating file sets that matches a regular expression. The second modifier updates the method `replicationItem` to handle replication schedules with file sets. The delta module `Journal` contains the necessary modifications for handling database journal replication. It has two class modifiers to add a new implementation of interface `ReplicationItem` and to update the method `replicationItem` to handle replication schedules with data base journals.

```
delta File {
  adds class FilePattern(FileId qualified, String pattern,
                         ServerDataBase db)
    implements ReplicationItem { .. }

  modifies class ReplicationSnapshotImpl {
    modifies Unit replicationItem(Schedule schedule) {
      original();
      if (isFileItem(schedule)) {
        Pair<FileId,String> it = right(item(schedule));
        ReplicationItem item =
          new FilePattern(fst(it),snd(it),this.db);
        items = Insert(item,items);
      }
    }
  }
}

delta Journal {
  adds class JournalItem(FileId qualified, ServerDataBase db)
    implements ReplicationItem { .. }

  modifies class ReplicationSnapshotImpl {
    modifies Unit replicationItem(Schedule schedule) {
      original();
      if (isJournalItem(schedule)) {
        FileId qualified = left(item(schedule));
        ReplicationItem item = new JournalItem(qualified,this.db);
        this.items = Insert(item,this.items);
      }
    }
  }
}
```

Listing 1.7: Delta modules for replication items

Listing 1.8 shows the product line configuration of the replication system in CL, where the features `Dir` and `Seq` are the features provided by the core module. The application condition for delta `Concurrent` states that this delta is applied if and only if feature `Concur` is selected and feature `Journal` is not selected. This application condition respects the constraint specified in the feature model shown in Listing 1.4.

Listing 1.8 also shows some example product selections for the replication system product line specified in PSL. For brevity, details of the main blocks have been omitted. For example, product `DS` defines a variant of the replication system that supports the core set of features. Product `DFJS` is a variant that supports all types of replication items, and product `DFC` supports both directory and file set replication, as well as concurrent client job scheduling.

```
productline ReplicationSystem {
  features Dir, File, Journal, Seq, Concur;
  delta File when File;
  delta Journal when Journal;
  delta Concurrent when Concur && (~ Journal);
}

product DS (Dir, Seq) { .. } // default product (core)
product DFC (Dir, File, Concur) { .. } // file pattern, concurrent
product DC (Dir, Concur) { .. } // directory, concurrent
product DFJS (Dir, File, Journal, Seq) { .. } // directory, concurrent
```

Listing 1.8: Product line configuration and product selections for the replication system

### 5.4 Temporal Variability of the Replication System

Existing products have to evolve to meet the market demand for new features. Thus, a product line may have to be changed simultaneously in several dimensions, which makes the management of the evolution a difficult task. A complete re-modeling of an evolved product line has a high cost. Therefore, it is beneficial to re-use model artifacts from previous versions of the product line in a compositional and incremental manner.

As an evolution from the current versions, Fredhopper aims to develop a loosely coupled, pluggable architecture for FAS. This architecture will allow components to be added and removed from a FAS deployment dynamically at runtime. One component that will benefit from this pluggable architecture is the Search Engine Optimizer (SEO) component. Search engine optimization improves the visibility of a client's website in search engines via search results. The SEO component includes an indexing facility for these search results which must also be replicated from the staging environment across all live environments.

The replication system currently supports replicating complete directories, file sets, and database journals. In order to add support for replication of SEO search result indices, we use dynamic delta modules to change the ABS model of the replication system product line. The dynamic delta module `IndexItemDelta` modifies the replication system such that an SEO component can be added to a FAS deployment at runtime. Listing 1.9 shows a partial implementation of this dynamic delta module. Read from top to bottom, the dynamic delta module `IndexItemDelta` in Listing 1.9 provides the following modifications:

1. Addition of an interface `Indexer` to model a generic indexer.
2. Addition of a class `SEOIndexer` implementing the `Indexer` interface for the SEO indexing facility. For brevity, implementation details are omitted.
3. Addition of the interface `IndexItem` that extends the interface `ReplicationItem` to associate an `Indexer` to a replication item.
4. Modification of the class `ReplicationSnapshotImpl` so that method `refreshSnapshot()` creates a new instance of the class `Indexer`, before refreshing individual items.
5. Modification of the delta module `File` in the following ways:
   (a) Modification of the class `ReplicationSnapshotImpl` (which is already modified by the delta module `File`) in such a way that the method `createReplicationItems()` associates the indexer with file set replication items.
   (b) Modification of the class `FilePattern` (that is added by the delta module `File`) such that it implements the interface`IndexItem` and such that the method `refresh()`, which refreshes the files contained in the replication item, acquires a set of indices (`Set<Index>`) stored at the file locations pointed to by the replication item. In the ABS model, we represent indices by the algebraic data type `Index`.

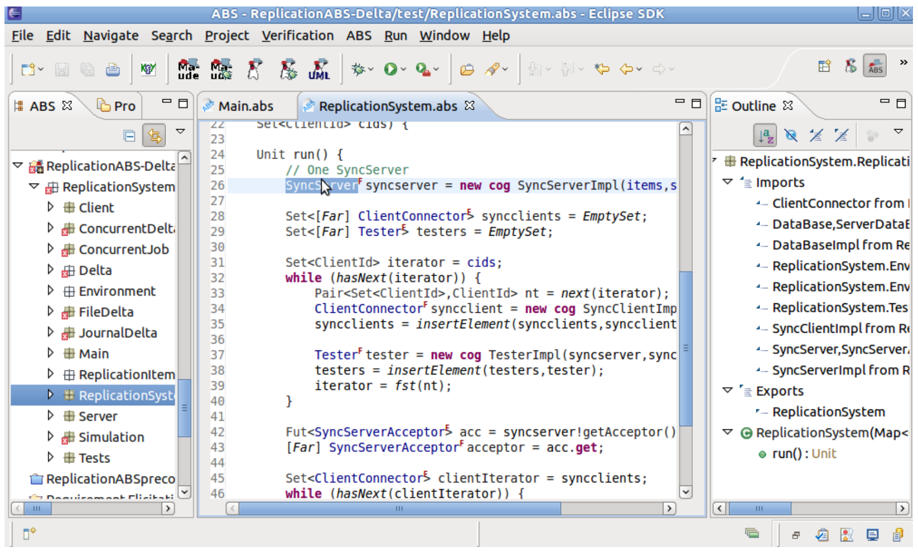
```
dyndelta IndexItemDelta {

  adds interface Indexer { // adding new indexer interface
    Set<Index> getIndex(FileId id, String pattern);
  }

  // a default implementation of the indexer
  adds class SEOIndexer(ServerDataBase db)
  implements Indexer { .. }

  // extends replication item to handle indices
  adds interface IndexItem extends ReplicationItem {
     Unit setIndexer(Indexer indexer);
  }

  modifies class ReplicationSnapshotImpl { // adds indexing support
    adds Indexer indexer = null;
    modifies Unit refreshSnapshot(Bool refreshSnapshot) {
      this.indexer = new SEOIndexer(this.db);
      original();
    }
  }

  modifies delta File { // adds indexing support
    modifies class ReplicationSnapshotImpl {
      modifies Unit createReplicationItems() {
        original();
        if (isFileItem(schedule)) {
          Pair<FileId,String> it = right(item(schedule));
          IndexItem item =
            new FilePattern(fst(it),snd(it),this.db);
          item.setIndexer(indexer);
          items = Insert(item,items);
        }
      }
    }

    modifies class FilePattern adds IndexItem {
      adds Set<Index> indices = EmptySet;
      adds Indexer indexer = null;
      adds Unit setIndexer(Indexer i) { this.indexer = i; }
      modifies Unit refresh() {
        original();
        this.indices = indexer.getIndex(qualified,pattern);
      }
    }
  }
}
```

Listing 1.9: Dynamic delta `IndexItemDelta` for supporting the SEO component

The dynamic delta module `IndexItemDelta` only provides support for search result indices used by SEO components. However, it can be composed with another delta module that provides a different implementation of the interface `Indexer`, thereby reusing dynamic delta module `IndexItemDelta` to support the indexing facility of other pluggable components.

## 6 ABS Tool Suite

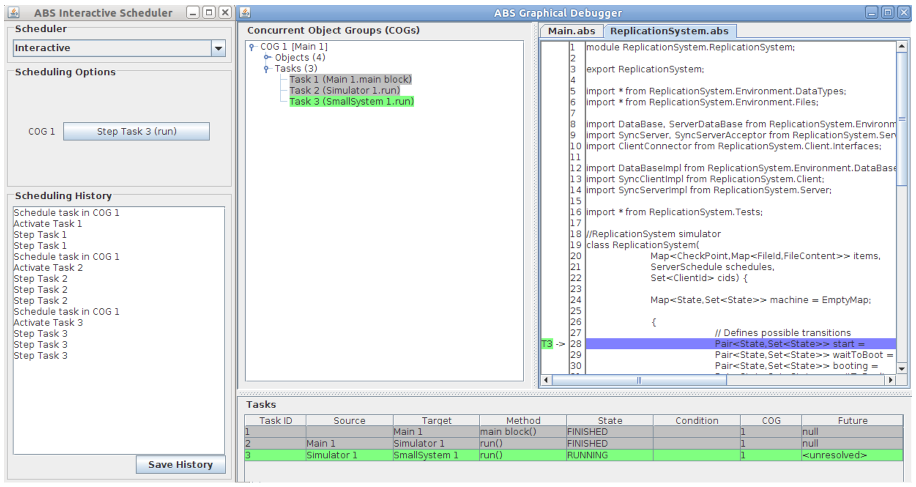
The ABS tool suite supports the modeling of highly configurable, distributed and concurrent systems in ABS. Specifically, the ABS compiler front-end takes a full ABS model as input, which includes the core ABS model in the Core ABS language, a feature model descriptions in $\mu$TVL, delta modules in DML, a product line configuration in CL, product selections in PSL. It checks the model for syntax and semantic errors and translates it into an internal representation. Several different back-ends can then be used to translate the internal representation into different languages, like Maude and Java, which allows ABS models to be executed and analyzed on these platforms.



```
    Set<ClientId> cids) {

    Unit run() {
        // One SyncServer
        SyncServer syncserver = new cog SyncServerImpl(items,s

        Set<[Far] ClientConnector> syncclients = EmptySet;
        Set<[Far] Tester> testers = EmptySet;

        Set<ClientId> iterator = cids;
        while (hasNext(iterator)) {
            Pair<Set<ClientId>,ClientId> nt = next(iterator);
            ClientConnector syncclient = new cog SyncClientImp
            syncclients = insertElement(syncclients,syncclient

            Tester tester = new cog TesterImpl(syncserver,sync
            testers = insertElement(testers,tester);
            iterator = fst(nt);
        }

        Fut<SyncServerAcceptor> acc = syncserver!getAcceptor()
        [Far] SyncServerAcceptor acceptor = acc.get;

        Set<ClientConnector> clientIterator = syncclients;
        while (hasNext(clientIterator)) {
```

Fig. 10: ABS Eclipse Perspective

Moreover, the HATS framework comes with a plug-in for Eclipse, a graphical debugger and a sequence diagram visualizer. The plug-in provides an Eclipse perspective for navigating, editing, parsing and type checking ABS models. It also provides integrations with different back-ends, allowing the generation of both Java and Maude code from ABS models and the execution and debugging of the generated code directly in the Eclipse perspective. Fig. 10 shows a screen shot of the Eclipse perspective, taken when conducting the replication system case study. The left hand pane in the figure shows the ABS module view. In this view, ABS modules, imports/exports, classes, interfaces, functions and data types of an open ABS Eclipse project are shown, independent of the actual ABS (.abs) files where they are defined in. The middle pane depicts the ABS editor view for editing ABS (.abs) files. The editor provides syntax highlighting, jump-to-declaration, content assistance, and code completion. The right hand pane is the ABS outline view, which shows the structure of the active ABS file. It is updated as the user edits the file.



```
Schedule task in COG 1
Activate Task 1
Step Task 1
Step Task 1
Schedule task in COG 1
Activate Task 2
Step Task 2
Step Task 2
Step Task 2
Schedule task in COG 1
Activate Task 3
Step Task 3
Step Task 3
Step Task 3
```

```
COG 1 [Main 1]
  Objects (4)
  Tasks (3)
    Task 1 (Main 1.main block)
    Task 2 (Simulator 1.run)
    Task 3 (SmallSystem 1.run)
```

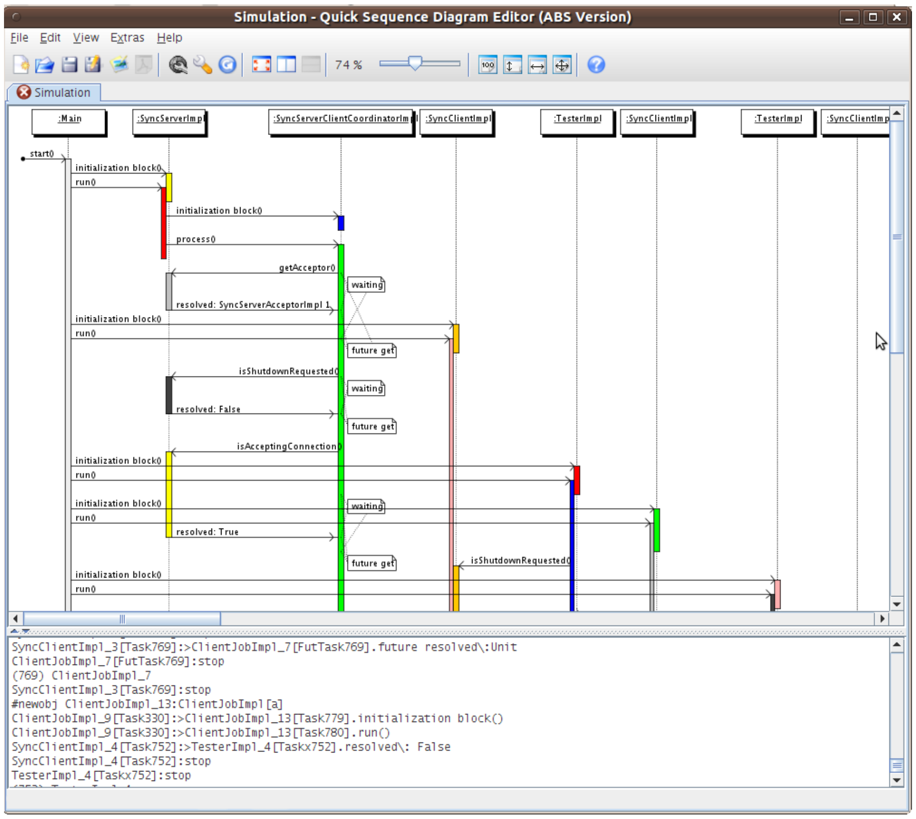
```
module ReplicationSystem.ReplicationSystem;

export ReplicationSystem;

import * from ReplicationSystem.Environment.DataTypes;
import * from ReplicationSystem.Environment.Files;

import DataBase, ServerDataBase from ReplicationSystem.Environm
import SyncServer, SyncServerAcceptor from ReplicationSystem.Ser
import ClientConnector from ReplicationSystem.Client.Interfaces;

import DataBaseImpl from ReplicationSystem.Environment.DataBase
import SyncClientImpl from ReplicationSystem.Client;
import SyncServerImpl from ReplicationSystem.Server;

import * from ReplicationSystem.Tests;

//ReplicationSystem simulator
class ReplicationSystem(
        Map<CheckPoint,Map<FileId,FileContent>> items,
        ServerSchedule schedules,
        Set<ClientId> cids) {

        Map<State,Set<State>> machine = EmptyMap;

        {
                // Defines possible transitions
                Pair<State,Set<State>> start =
                Pair<State,Set<State>> waitToBoot =
                Pair<State,Set<State>> booting =
```

| Task ID | Source | Target | Method | State | Condition | COG | Future |
|---|---|---|---|---|---|---|---|
| 1 | | Main 1 | main block() | FINISHED | | 1 | null |
| 2 | Main 1 | Simulator 1 | run() | FINISHED | | 1 | null |
| 3 | Simulator 1 | SmallSystem 1 | run() | RUNNING | | 1 | <unresolved> |

Fig. 11: ABS Graphical Debugger

Fig. 11 shows a screen shot of the ABS graphical debugger, taken when conducting the replication system case study. The graphical debugger allows stepping through ABS models, either interactively or via a seeded random sequence of steps. The left hand pane in the figure shows the current scheduler and the sequence of steps that have already been executed. The right pane shows the current state of the execution. The window includes the state of individual concurrent object groups, a list of concurrent tasks, and an ABS model view, relating active tasks to the currently executing statement in the ABS model.

Fig. 12 shows a screen shot of the ABS sequence diagram visualizer, taken when conducting the replication system case study. The visualizer is used in conjunction with the graphical debugger. When stepping through ABS models via the debugger, the visualizer shows the asynchronous messages sent between objects in various concurrent object groups.



```
SyncClientImpl_3[Task769]:>ClientJobImpl_7[FutTask769].future resolved\:Unit
ClientJobImpl_7[FutTask769]:stop
(769) ClientJobImpl_7
SyncClientImpl_3[Task769]:stop
#newobj ClientJobImpl_13:ClientJobImpl[a]
ClientJobImpl_9[Task330]:>ClientJobImpl_13[Task779].initialization block()
ClientJobImpl_9[Task330]:>ClientJobImpl_13[Task780].run()
SyncClientImpl_4[Task752]:>TesterImpl_4[Taskx752].resolved\: False
SyncClientImpl_4[Task752]:stop
TesterImpl_4[Taskx752]:stop
```

Fig. 12: ABS Sequence Diagram Visualizer

In the course of the HATS project, the development of the tool suite and the case studies go hand-in-hand. On the one hand, the tool suite encourages the practical application of the ABS. This is evidently shown by modeling a large piece of production code, such as the replication system, with the ABS. On the other hand, the continuous application of the ABS to real-life case studies steers the development of the ABS tool suite ever closer to an industrial-strength framework.

## 7 Conclusion

The ABS modeling language of the HATS project, together with the HATS ABS tool suite constitutes a new model-based approach to the development of concurrent systems that exhibit a high degree of variability.

The Core ABS language (Section 2) is a state-of-the-art, strongly typed, abstract, concurrent, object-based modeling language that is fully executable.

Tightly coupled groups of objects with shared memory are encapsulated into *concurrent object groups* (COGs) that realize data-race free computation via cooperative multi-tasking. Objects from different COGs may only use asynchronous communication and message passing. Future types make it possible to continue computation, while waiting for the result of an asynchronous method call. Functional expressions over parametric algebraic data types are used to model data in an implementation-independent way.

A novelty of ABS is the possibility to define software product lines with the help of feature models and delta modeling. Delta modeling is a flexible code reuse technique that has been shown to be particularly suitable for the description of product lines (see [9, 20] for a thorough discussion). In contrast to architectural languages, ABS provides a formal link between features and their implementation. A dynamic version of delta modules allows describing the evolution of ABS models at runtime (Section 4).

As illustrated in Section 6, the ABS language is integrated into a tool suite that includes editing, parsing, type checking, compilation (into Java and Maude), debugging, and visualization. The ABS language has a formal, mathematical semantics which is the basis of advanced tools, partially under development, that address test case generation, model mining, functional verification, various static analyses, resource analysis, etc.[8] The aim of HATS is to position ABS and its tool suite as a *single source* technology for architectural modeling, functional modeling, verification, and implementation of highly configurable, concurrent software systems [23].

The case study in Section 5 demonstrates that this is not just a vision, but a realistic goal. We illustrated how to model a core component of a complex, industrial distributed system using the ABS language. Specifically, we have shown:

– How to use the core ABS language to model the concurrent aspects of the replication system.
– How to refine the core model using the ABS language to express important variabilities of the replication system to capture all possible component variants.
– How to capture evolution (variabilities over time) with the ABS language in order to reduce modeling cost.

The HATS tool suite, documentation, as well as several case studies, including the one discussed here, are available from http://www.hats-project.eu.

## References

1. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, J.: Component-based product line engineering with UML. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)

[8] A description of these is beyond the scope of this tutorial.

2. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Software Eng. 30(6) (2004)
3. Beck, K.: Extreme Programming. Addison-Wesley (1999)
4. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: European Symposium on Programming (ESOP'07). Lecture Notes in Computer Science, vol. 4421, pp. 316–330. Springer-Verlag (2007)
5. Boucher, Q., Classen, A., Faber, P., Heymans, P.: Introducing TVL, a text-based feature modelling language. In: Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January 27-29. pp. 159–162. University of Duisburg-Essen (January 2010), http://www.vamos-workshop.net/2010
6. Classen, A., Boucher, Q., Heymans, P.: A text-based approach to feature modelling: Syntax and semantics of TVL. Science of Computer Programming (Nov 2010), http://linkinghub.elsevier.com/retrieve/pii/S0167642310001899
7. Czarnecki, K., Eisenecker, U.: Generative programming. Addison-Wesley (2000)
8. Evaluation of Core Framework (Aug 2010), deliverable 5.2 of project FP7-231620 (HATS), available at http://www.hats-project.eu
9. Final Report on Feature Selection and Integration (Mar 2011), deliverable 2.2b of project FP7-231620 (HATS), available at http://www.hats-project.eu
10. Full ABS Modeling Framework (Mar 2011), deliverable 1.2 of project FP7-231620 (HATS), available at http://www.hats-project.eu
11. Fredhopper Access Server, http://www.fredhopper.com
12. Gomaa, H.: Designing Software Product Lines with UML. Addison Wesley (2004)
13. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented Programming. Journal of Object Technology (March/April 2008)
14. Johnsen, E.B., Kyas, M., Yu, I.C.: Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In: Cavalcanti, A., Dams, D. (eds.) Proc. 16th International Symposium on Formal Methods (FM'09). Lecture Notes in Computer Science, vol. 5850, pp. 596–611. Springer-Verlag (Nov 2009)
15. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and System Modeling 6(1), 35–58 (Mar 2007)
16. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: Proceedings of the European Conference on Object-Oriented Programming. Lecture Notes in Computer Science, vol. 1241 (1997)
17. Medvidovic, N., Taylor, R.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering (2000)
18. Pohl, K., Böckle, G., Van Der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
19. Schaefer, I.: 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)
20. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Proc. of 15th Software Product Line Conference (SPLC 2010) (Sep 2010)
21. Schaefer, I., Bettini, L., Damiani, F.: Compositional Type-Checking for Delta-oriented Programming. In: Intl. Conference on Aspect-oriented Software Development (AOSD'11) (2011), (to appear)
22. Schaefer, I., Damiani, F.: Pure Delta-oriented Programming. In: FOSD 2010 (2010)

23. Schaefer, I., Hähnle, R.: Formal methods in software product line engineering. IEEE Computer 44(2), 82–85 (Feb 2011)
24. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: European Conference on Object-Oriented Programming (ECOOP'10). Lecture Notes in Computer Science, vol. 6183, pp. 275–299. Springer-Verlag (Jun 2010)
25. Ziadi, T., Hélouët, L., Jézéquel, J.M.: Towards a UML Profile for Software Product Lines. In: Workshop on Product Familiy Engineering (PFE). pp. 129–139 (2003)