MetaABS and Dynamic Model Updates

Radu Muschevici, José Proença and Dave Clarke
DistriNet & IBBT, Dept. Computer Science
KU Leuven, Belgium

Abstract

Long-lived, dependable systems evolve during their life time: new functionality is added, errors are discovered, new requirements appear, etc. It is often infeasible to stop and redeploy a system in order to upgrade it, due to the associated downtime. A system that can adapt to changes dynamically is more dependable. In this paper we present dynamic ABS: to cope with the need to modify ABS code at runtime, we introduce a new reflective layer that allows introspection and manipulation of running code. This layer is exposed in a language extension called MetaABS. The new dynamic back end and the MetaABS language provide a framework for implementing and analysing evolving software in ABS.

1 Introduction

Meta-programming is generally understood as the ability to observe and modify the structure and behaviour of a program from within a program, either statically or at runtime. A meta-programming interface exposes basic elements of the programming language and the runtime environment to the programmer, enabling their inspection and modification. While it exposes these elements, it also abstracts away from their implementation.

Languages that support meta-programming commonly achieve this by providing *reflection*, that is, the ability of a program to inspect and modify itself at runtime. Thus the meta-program (the program transforming program) and the program that is transformed are the same. Reflection is decomposed into *introspection*, meaning the ability of a program to examine itself, and *intercession*, which enables a program to modify its state and behaviour. In other words, introspection and intercession provide, respectively, read and write access to elements of the language. For example, the

Java Reflection API is a meta-programming interface that provides methods to examine, and, to a very limited extent, modify the runtime properties of objects including their class, interfaces, fields and methods.

Several approaches for meta-programming exist, explained better in the related work section below. This paper describes a meta-programming mechanism for the ABS language [15], introducing MetaABS. The nature of ABS brought several challenges in the development of MetaABS.

- The ABS language was designed as a balanced compromise between a simplistic model that can be easily verified and formally analysed, and a language rich enough to model real world and complex scenarios. Hence MetaABS should also strive for both objectives: have a small core that can be easily analysed and still be practical enough to be used in larger examples, and to have an executable engine.

- There is an explicit concurrency model in ABS, where remote objects communicate only asynchronously, and concurrent local objects have a cooperative scheduling in a single thread. Meta-objects in MetaABS also need to fit this concurrency model.

- Some static meta-programming capabilities already exist in ABS: it has a core language extended with other small languages that describe how to generate a family of software products by applying program transformations, wrapped in delta modules. MetaABS needs to make sure the same static functionality is made available at runtime.

The main contributions of this paper are (1) MetaABS, a meta-programming facility for the ABS language, and (2) a dynamic Java back end that supports it. Further, we illustrate their application in adapting ABS models at runtime. The first application shows how user-defined process scheduling algorithms are defined and dynamically attached to groups of concurrent objects. Secondly, we show how a model is transformed based on the software product line at its core. Executable ABS models can include a collection of related software products. While only one product can run at any time, MetaABS allows us to adapt the current product to a different one at runtime.

The purpose of MetaABS is to provide a unified interface for various runtime models, either developed and under development, for the ABS language. By adding meta-programming capabilities to ABS we allow some model analysis tasks to be encoded directly in ABS and performed at runtime.

1.1 Abstract Behavioural Specification (ABS) language

ABS is a concurrent, multi-paradigm modelling language [15]. Syntax-wise, ABS resembles standard programming languages like Java. Nevertheless ABS is more a modelling than a programming language, because the design of ABS is strongly focused on providing a language that is easy to analyse. High execution performance, for example, is not a design goal of ABS.

ABS supports first-order functional programming with algebraic data types. Functional code is guaranteed to be free of side effects. One consequence of this is that functional code may not use object-oriented features [1, 15]. ABS also supports class-based, object-oriented programming with standard imperative constructs. ABS is especially designed for modelling 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, concurrently running cogs at runtime. Cogs can be regarded as autonomous runtime components that are executed concurrently, share no state and communicate via method calls. Cogs can reference objects of other cogs, however, these *far* references can only be used as targets for asynchronous method calls.

ABS has a nominal type system with interface-based subtyping. ABS does not support class inheritance and overloading, and instead code reuse can be achieved in ABS by using *deltas*. Delta-oriented programming [23] is an approach that aims at developing a set of programs simultaneously from a single code base, following the software product line (SPL) engineering approach [21]. In delta-oriented programming, features defined by a feature model are associated with code modules that describe modifications to a *core* model. These modules are called *delta modules* (or *deltas* for short).

1.2 MetaABS

MetaABS comprises a set of operations (a *meta-object protocol* [19]) that expose internals of ABS models, such as classes, methods, object state, concurrent object groups (cogs), task schedulers and message queues, making it possible to observe and modify a model while it is executed. Some research tracks being currently pursued with respect to the ABS language include the scheduling of tasks inside concurrent object groups; the dynamic reconfiguration of software products; deployment component configuration; and runtime method dispatch.

We designed MetaABS based on requirements provided by several runtime analysis use cases within the HATS European project,[1] such as the two

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

applications mentioned in the next subsection. For example, the scheduling of tasks within a cog for real-time systems has been investigated by Bjørk et al. [3], but the authors could execute only within the abstract ABS interpreter implemented in the Maude [7] rewrite engine. We implemented application-level scheduling at the level of cogs; schedulers are configured using MetaABS. Now, such models are executable on the standard Java VM.

Other uses for MetaABS include the implementation of the ABS component model developed by Lienhardt et al. [20] and resource analysis using deployment components by Johnsen et al. [17]. In the future we also plan to explore user-configurable method dispatch in ABS. An advantage of using MetaABS for these tasks, beyond having a "standard" interface for accessing model internals, is that it does not require extending the ABS language, by means of annotations, or otherwise.

1.3 Applications

Meta-programming opens the possibility of modifying program aspects that were initially intended to be static, such as the execution semantics of the language or the code itself. This paper explores two example of such applications, which are presented respectively in Sections 4 and 5. They investigate how to dynamically change the order of evaluation of parallel tasks in a single cog, and how to reconfigure a model's structure and behaviour by applying delta modules at runtime. Both these problems arise naturally in ABS due to (1) the existence of non-determinism for selecting tasks within a cog and to (2) the presence of a core mechanism that allows the production of a family of software products based on code transformations.

We now focus on the former application to motivate MetaABS. Using user-defined schedulers of concurrent objects has already been study in the context of ABS for real-time systems [3]. The authors include special annotations to ABS code defining functions that select the task that should be executed next, and manage aspects such as duration and deadlines of tasks.

```
// A scheduler which switches strategy based on the length of the queue
def Process lengthSensitive(Int limit, List<Process> l) =
  if (length(l) < limit) then shortestProc(l) else earliestProc(l);

[Scheduler: lengthSensitive(limit,queue)]
class ServerImp (Int limit) implements Server { ... }
```

The example above defines a function `lenghSensitive` that acts as a scheduler for the cogs of instances of `ServerImp`. Annotations are written within square brackets, and the keyword **queue** acts as an argument that will bind to the

queue of tasks during execution. Our meta-programming approach avoids the usage of special annotations, using reflection instead.

```
class LenghSensitive(Int limit) implements Scheduler {
  Process schedule(List<Process> l) { ... }
}
Cog g = ...
g.setScheduler(new LengthSensitive(limit));
```

Two main advantages emerge from this approach. On one hand the scheduling mechanism can be included in a more general framework for meta-programming problems. On the other hand MetaABS provides a Java back end that compiles into executable Java programs, contrary to the previous approach that has only a hand-crafted simulator implemented in the Maude rewriting system [7]. Bringing real-time issues the Java platform allows us to use JVM's time measuring capabilities instead of the more artificial notion of time in Maude, which is implemented by assigning a cost to every expression.

1.4 Organisation of the paper

We provide an overview of related work in Section 2. Section 3 details the MetaABS API. Sections 4 and 5 present two applications of MetaABS for run-time model analysis: user-configurable process schedulers and the dynamic reconfiguration of software products. In order to allow the modification of model elements, the support of the ABS back end is required. Therefore, in addition to adding introspection capabilities to the standard ABS Java back end, we design a so-called *dynamic Java back end*, which readily enables the modification of a model's structural and behavioural elements. The ABS dynamic Java back end is presented in Section 6. Section 7 concludes this paper.

2 Related Work

Meta-programming consists of writing programs that can write or transform other programs (or themselves), and has appeared in a multitude of flavours. Different classifications for such programs exist, including: generative vs. intensional (creating or analysing programs), compile-time vs. run-time (when the programs are written or transformed), heterogeneous vs. homogeneous (what programs are transformed: others or themselves), and lexical vs. syntactical (over strings or over abstract syntax trees). MetaABS can be categorised as an intensional, run-time, homogeneous, and syntactical approach, and is based on reflection – possibly the most common mechanism for this

category, supported by mainstream languages such as Java, C#, and Python. In this section we present a brief overview over some meta-programming approaches and describe existing research tracks in the context of ABS language that deal with some forms of meta-programming.

Observe that the delta mechanism of ABS provides per se meta-level functionality. Similarly to aspect oriented programming [18], a collection of delta modules transforms a core program by applying a sequence of transformations. A higher-level heterogeneous meta-programming approach has been explored for ABS (cf. Chapter 4 of Deliverable 1.3 [11]), using the meta-programming language is Rascal[2] to generate and transform full ABS code, including delta modules and descriptions of the product lines. The homogeneous approach by MetaABS allows manipulating ABS programs at run-time, opening new possibilities for evolvability.

Lisp was one of the first languages to support homogeneous meta-programming, using expressions based on macros. Macros provide a generative approach for meta-programming, allowing macros to be expanded at compile-time. Macros in Scheme benefit also from being fully hygienic, that is, they avoid name clashes during macro expansion. Later other generative homogeneous approaches improved the macro expansion mechanism of Lisp using multi-stage programming, starting with the introduction of MetaML [27], reaching more modern and rich languages. Multi-stage programming provides a finer control of the temporal aspects, allowing multiple levels of programs generated inside programs to co-exist, wrapped in so-called quasi-quotations. Other powerful and (type) safe languages borrowed these ideas, such as MetaKlaim [12], MetaOCaml [26], Template Haskell [25] (which fully supports code compilation at runtime), and also in Scala macros are being experimented.[3]

In the context of object-oriented programming, a common way to write meta-programs is based on *reflection*. For example, the Java reflection API[4] allows the reification of program elements such as classes into special objects that can support reflective operations. Bracha and Ungar [5] expose some of the limitations of Java's core reflection mechanism and propose the use of a new level of indirection which they call *mirrors*. The authors claim that reflective APIs built around mirrors adhere to the design principles of encapsulation, stratification, and ontological correspondence. This means the behaviour of mirrors is encapsulated, the meta-level facilities are separated

[2] http://www.rascal-mpl.org

[3] http://docs.scala-lang.org/overviews/macros/overview.html

[4] http://docs.oracle.com/javase/tutorial/reflect

from the base-level functionality, and the ontologies of the meta-level and the language that is manipulated have a correspondence. The MetaABS follows this approach and uses mirrors to encapsulate the meta-level transformations. Furthermore, we developed a new Java back end that supports these mirrors, that is, we compile ABS programs with the MetaABS layer into executable Java code with meta-level functionality that goes beyond Java-reflection. Currently we do not know of any realisation of mirrors in Java. Delta modules in MetaABS are seen as a sequence of meta-level operations, and thus can be applied dynamically.

A challenge addressed by MetaABS that is not present in most object-oriented languages is *concurrency*. Objects in ABS live in cogs, and communication between objects from different cogs must be asynchronous. This principal is extended to the mirror objects that must also belong to a given cog, and modification of remote objects must be done asynchronously. Meta-programming of concurrent languages has been less study than for sequential languages. For example, Schneider and Lumpe propose a meta-level approach for concurrent object-oriented programs based on the Form-calculus, a variant of the π-calculus introduced by the authors [24], and Neuendorffer explored in his PhD dissertation meta-programming in actor-oriented languages [2], focusing on the reconfiguration of actors at run-time.

With respect to the application of MetaABS, this paper uses two examples to illustrate different aspects of this meta-programming language. The first application was initially presented Bjork et al., where they introduce real-time with user defined schedulers as a means to control the scheduling policy at the level of active objects; the user can define scheduling policies as functions that operate on a `queue` data type representing the available processes, and return a single process from that queue. Objects can be annotated with a dedicated scheduling function, which can also access the state of the object in question. Their work relies on *annotations*. We use reflective operations provided by MetaABS to configure real-time parameters and schedulers.

MetaABS is also designed to enable models of dynamic software product lines in ABS. Based on the delta mechanism to build a family of software products, Damiani and Schaefer [8] described a model to dynamically modify the existing software product to another with different features. Later Helvensteijn also explored the modelling of delta models that evolve dynamically [14]. In our second example of an application of MetaABS we further explore dynamic evolution of software product lines, where the feature model changes and code is removed or added to a model at runtime. The usage of MetaABS allows the treatment of such problems under a single meta-level framework, without requiring the development of specialised tools to man-

age the redeployment of software products whenever it needs to be modified.

Finally, safe dynamic software updates remain a very interesting application of MetaABS that still needs to be investigated carefully for the context of ABS. Several studies have been made in this direction, such as the work on incremental dynamic updates by Wernli et al. [28], where concurrent threads can share memory, and the work on dynamic classes by Johnsen et al. [16], where communication between objects is exclusively asynchronously.

3 MetaABS Interface

MetaABS is a largely object-oriented reflective interface to the ABS language. It provides an abstraction of the underlying ABS runtime. MetaABS is implemented as a library alongside the ABS standard library. It is easily extensible should new requirements arise. Extending MetaABS does not require changing the ABS language itself. This section lists the main types that MetaABS introduces and shows the provided operations in Figure 1.

3.1 MetaABS Types

Object Mirrors An `ObjectMirror` reflects on an existing ABS object. One obtains an object mirror by invoking the built-in function `reflect(object)` on any given ABS object. The object mirror provides a set of reflective operations such as for getting or setting the object's class and its concurrent group affiliation (cf. Figure 1). We opted for a mirror based design [5] in order to achieve a separation between an object's regular interface, determined by its type, and its reflective interface.

Future Mirrors Asynchronous messages to objects in remote cogs return a future to the caller and create a new process in the remote cog. The future is used to obtain the message's result after the message has been processed. A `FutureMirror` reflects on the future. It provides a set of operations that configure real-time attributes of the created process, such as setting the deadline and cost of the process that will supply the result to the future.

Object `Object` is the type of ABS objects. One can use reflective operations on objects by first using the function `reflect(object)` and then calling a reflective operation on the returned `ObjectMirror`. `ObjectMirror` provides a `getObject()` method that returns the `Object` it reflects upon.

Classes A `Class` type represents an ABS class. Its interface includes operations to add and remove methods.

Cog A `Cog` represents a concurrent object group (cog), which in ABS is the unit of concurrency and distribution. A cog has a processor which runs at most one process at any given time, a queue of processes waiting to execute, and a process scheduler, which determines which process from the queue will run at each scheduling point. The process scheduler determines the scheduling policy and is configurable.

Scheduler Schedulers provide a `schedule()` method which returns a `Process` from the cog's queue. It is possible to define a custom scheduler and attach it to a cog using `setScheduler()`.

Product Line A separate interface is dedicated to configuring the SPL attached to an ABS model. ABS supports dynamic SPLs, meaning that a running product can be reconfigured into another product dynamically. All a user needs to do is to obtain a handle to a `Productline` object by calling the `getPL()` function, and call `configureProduct()` with the name of the target as an argument. The user also needs to ensure that the system is in a "safe" state when reconfiguration occurs.

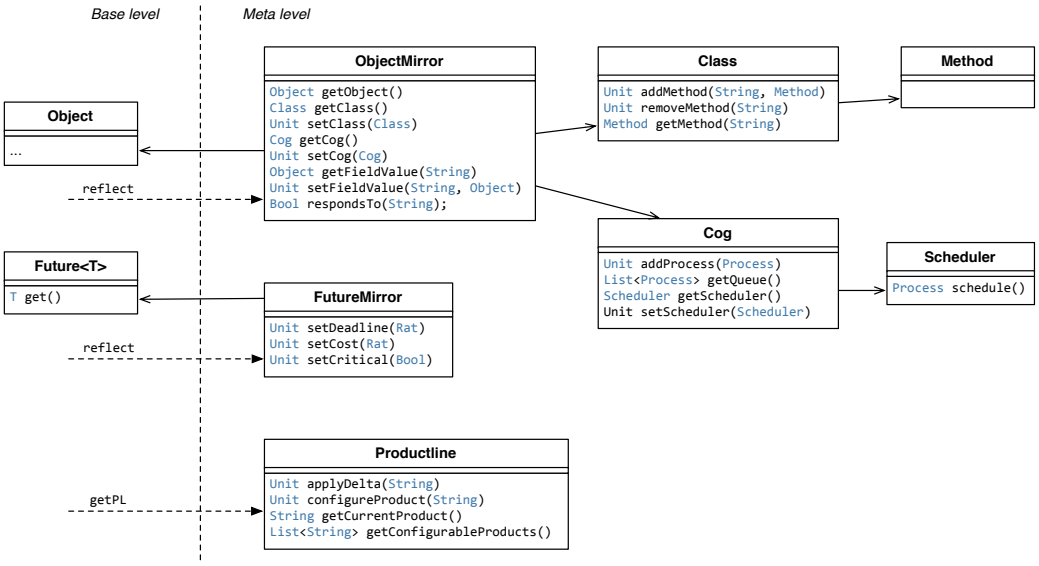

Figure 1: MetaABS interface. Labels on the dotted arrows denote the MetaABS functions that return an object of the type they point to.

3.2 Usage Example

The following example illustrates how reflective operations are accessed from an object mirror.

```
import * from ABS.Meta;
class C implements I { Unit foo() {...} }
{
 I obj = new C();
 ObjectMirror mir = reflect(obj);
 Class cls = mir.getClass();
 cls.removeMethod("foo");
}
```

4 User-Defined Process Schedulers and Real-time support

Task scheduling in ABS is by default non-deterministic. The ABS Java compiler back end provides a flexible configuration mechanism to define the scheduling strategies that are used during the execution of an ABS system [9]. We make this configuration mechanism accessible at runtime for ABS models that are compiled to Java.

MetaABS provides operations to access and modify the cog of a particular object. The cog can be assigned a user-defined scheduler object. User defined schedulers need to implement the `Scheduler` interface, which provides a `schedule` method that returns a `Process` from the (builtin) `queue` data type. Schedulers can be given access to program state through class parameters.

Real-time ABS provided support for analysing real-time properties of computations in ABS on the Maude platform. Following **(author?)** [4]'s work we augment processes with a set of standard real-time attributes.

Arrival time is the time when the process is created in response to an asynchronous method call.

Cost is the processor time needed to complete the task without interruption.

Deadline is the time by which the process should complete, relative to the arrival time.

Criticality is a boolean parameter related to the consequence of missing the deadline.

Start time is the time when the process starts running.

```
class MyScheduler(Int x) implements Scheduler {
  Process schedule(List<Process> l) { return head(l); }
  Unit setX(Int x) { this.x = x; }
}
interface C { Unit m(); }
```

```
class CImpl implements C {
  Unit m() {...}
}
{
  C o = new cog CImpl();
  ObjectMirror om = reflect(o);
  Cog g = om.getCog();
  Scheduler s = new MyScheduler(0);
  ObjectMirror sm = reflect(s);
  sm.setCog(g);
  g.setScheduler(s);
```

```
class CImpl implements C {
  {
    ObjectMirror om = reflect(this);
    Cog g = om.getCog();
    Scheduler s = new MyScheduler();
    g.setScheduler(s);
  }
  Unit m() {...}
}
{
  C o = new cog CImpl();
```

```
  Fut<Unit> f = o!m();
  FutureMirror fm = reflect(f);
  fm.setDeadline(1/40);
  fm.setCost(1/100);
  fm.setCritical(false);
}
```

Figure 2: Implementation of a user-defined scheduler in MetaABS.

Finishing time is the time when the process is completed.

Arrival time, as well as start and finishing time are set automatically to the system's time at process creation, start and completion. *Cost*, *Deadline* and *Criticality* are process parameters settable by the user. To set these attributes, the MetaABS future mirror is used.

Implementing support for real-time processes in the ABS Java back end means we can measure time using Java's standard clock. Previous implementations of Real-time ABS execute on the Maude rewrite engine and uses a clock that runs from 0 to 100 by default and only advances (from the point of view of a process) when the process suspends, awaits, executes a get expression or a statement that has a cost.

The ABS code example in Figure 2 shows two ways how the scheduling strategy of a cog can be customised. The `MyScheduler` class implements a `schedule` method, which returns the `Process` to be scheduled for execution. The state of the program at runtime can also play a role in scheduling through class parameters, represented here by the parameter `Int x`.

To assign a `Scheduler` object to a cog, one has to obtain a handle on the

`Cog` object from an object that is defined inside that cog. Then one uses to the `setScheduler` operation on that cog object to change the scheduler.

In the variant on the left side, the `MyScheduler` object is created outside the cog containing object `o`. This would cause a runtime error when synchronously calling `schedule()`. Therefore it is necessary to move the scheduler object `s` to the cog whose processes it should schedule. MetaABS allows to change the cog of an object, effectively moving the object from one processor to another. Alternatively, a scheduler can be defined directly inside class `C`'s *init* block, effectively placing it inside the same cog as any instance of class `C`. This implies that the scheduler is tied to the class inside which it is defined. In effect, it sets the scheduler of any cog inside which an instance of the class is created. In the first variant, the scheduler can be assigned to any object (of any class). This variant is shown in the right column.

Since the object `o` is in a remote cog, its method is invoked asynchronously. This creates a process that will be scheduled using the user-defined scheduling algorithm. To set the real-time parameters of the process, we `reflect` on the future and use its mirror to set deadline, cost and criticality.

5 Dynamic Software Product Re-configuration

MetaABS and a runtime model that supports runtime adaptation are useful for various scenarios in which the ABS model needs to change dynamically. One such scenario is modelling dynamic SPLs, which introduce the ability of *reconfiguring* products at runtime. Runtime reconfiguration is understood as the transformation of a product into another valid product defined by the SPL, all without the need to re-compile and deploy the system. Adding this facility to ABS complements the static SPL modelling capability of ABS. Static product generation introduced support for configuring a particular SPL product at compile time by taking an ABS *core* model and a set of *delta modules* and "flattening" them to obtain an executable core ABS model of that single product. We add support for runtime product reconfiguration to ABS by adding a dynamic representation of product specifications and delta modules and by deferring the flattening process to the runtime. Product reconfiguration takes the runtime representation of a product and applies a set of dynamic deltas to obtain a different product of the SPL.

A runtime product specification includes a set of *adaptations* that describe how the product can evolve at runtime. An adaptation contains the names of the products that the current product can be transformed into, and a *state update* specification, which describes the transformation of the model's state when the product is adapted.

The conceptual difference to static product configuration is that, while with static product configuration we always start with the base product (represented by a *core* ABS model) and apply a sequence of deltas until we obtain any of the products specified by the product line, dynamic product reconfiguration starts with any product and applies a set of deltas to obtain a different product from a subset of the set of specified products. The premise is that any particular product can be only reconfigured at runtime into a different product from a well-defined set of products. Figure 3 illustrates this idea.

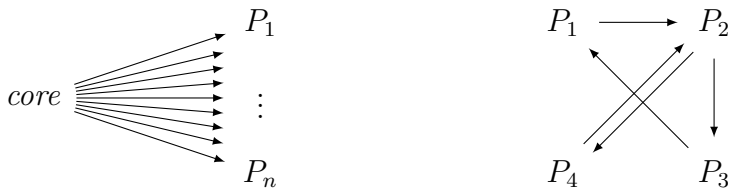

Figure 3: Static product configuration transforms a core into any product of the SPL (left). With dynamic product reconfiguration (right), only certain transitions between products are allowed at runtime.

MetaABS provides a `Productline` interface that gives access to operations that adapt the running software product by applying a sequence of delta modules. A `Productline` represents the set of products P that are available at runtime. Each of these products can be *reconfigured* into another product from a subset of $P' \subseteq P$. P' is defined statically in the product configuration.

```
1  product P1 (F1, F2) {
2     P2 by Upd1_2;
3     P3 by UPd1_3;
4  }
5  product P2 (F1, F2, F3);
6  product P3 (F1, F4);
7
8  productline PL {
9     delta D1 when F1 && F2;
10    delta AddF3 from F1 && F2 && ~F3 to F1 && F2 && F3;
11    delta AddF4RemoveF2 from F1 && F2 && ~F4 to F1 && ~F2 && F4;
12 }
```

Figure 4: Product specification example for dynamically re-configurable product.

Figure 4 shows and example of a product specification. A product `P1` contains features `F1` and `F2`. This product can be reconfigured at runtime

into products `P2` or `P3` by applying a set of delta modules as specified by *delta clauses* in the product line configuration (line 8), and the *updates* `Upd1_2` or `Upd1_3` (lines 2–3) respectively.

With compile-time product configuration we always start with a program *core* and apply a sequence of delta modules. At run-time, products can be *re*-configured, meaning that we start with a product and obtain another product. The syntax of delta clauses has been extended to allow two application conditions: one that is evaluated against the current feature selection (the product before reconfiguration), introduced by the **from** keyword and one which specifies the feature selection after reconfiguration, introduced by the **to** or **when** keywords (lines 10–11).

The **by** keyword introduces a *state update*, which specifies how to transfer the objects' state when transitioning from a product to another product, and also specifies synchronisation conditions that define when objects can be updated safely.

6 A Dynamic Back End for ABS

A back end that fully implements the MetaABS API has been contributed to the ABS compiler tool chain. The key idea behind its design is to use dynamic structures in the target language (Java) to represent ABS language elements. The main difference from the standard ABS Java back end is how language elements are represented when translated to Java. Whereas the standard Java back end (cf. Chapter 2 of Deliverable 1.4 [10]) represents ABS classes, functions and data types as Java classes and ABS interfaces as Java interfaces, the dynamic Java back end uses Java (singleton) objects to represent interfaces, classes, methods, objects, object fields, cogs, data types, functions, etc. Such a representation trades execution performance for fully malleable ABS models. This section details the design of the dynamic ABS back end and illustrates the code generation process by examples.

6.1 Design

Figure 5 shows the main types used to represent ABS model elements in Java. When compiling a model using the dynamic ABS Java backend, MetaABS operations (Figure 1) are mapped to this interface.

ABS classes are represented as objects of type `ABSDynamicClass`, which provide operations to set or modify the class name, initialisation block (constructor), methods, fields and class parameters. Methods and constructors of classes are represented as objects of type `ABSClosure`. `ABSClosure` is an

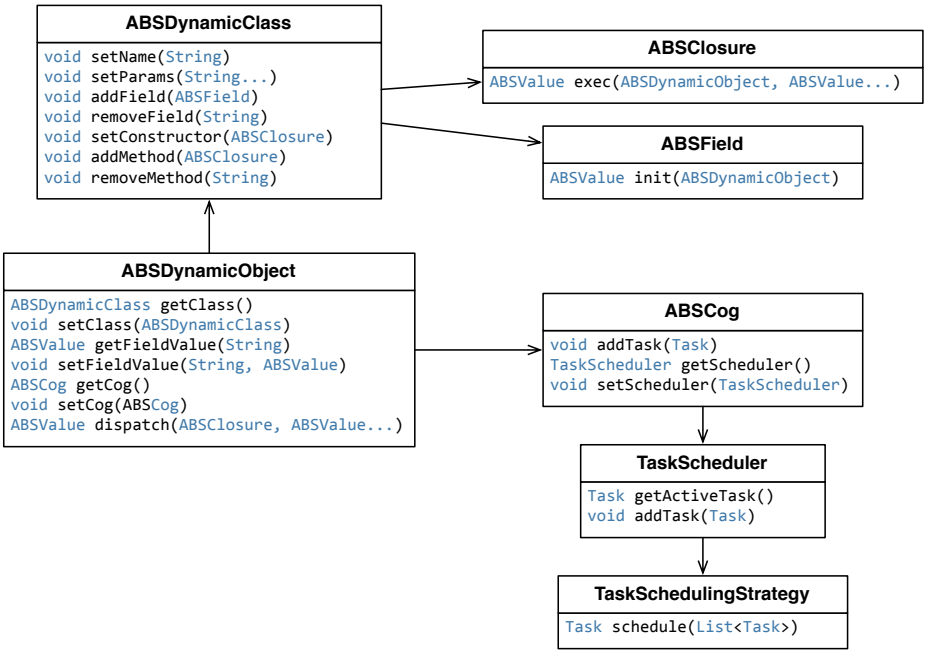

Figure 5: Dynamic ABS Java back end interface (partial view).

abstract class whose `exec` method serves as a placeholder for each method's specific behaviour. To create a method, a concrete subclass of `ABSClosure` overriding `exec` needs to be provided. Fields are represented as objects of type `ABSField`. Concrete fields inherit from `ABSField` and provide a specific initialisation expression by overriding the `init` method. ABS objects are instances of `ABSDynamicObject`, which offers an interface through which it is possible to modify their class and cog associations, update their fields and call methods. `ABSCog` objects are associated to objects and mainly control the scheduling of tasks. By modifying a cog's `TaskScheduler` one can configure its scheduling policy, implemented as a `TaskSchedulingStrategy`.

6.2 Usage

To generate Java bytecode using the dynamic ABS Java back end, the ABS compiler is invoked using the `-dynamic` switch. For example, the following command generates Java code for the ABS program `PeerToPeer.abs` (provided as an example in Deliverable 1.2 [9]) into the `javagen` directory:

```
java -cp absfrontend.jar abs.backend.java.JavaBackend -d
   javagen -dynamic PeerToPeer.abs
```

To execute the code generated from the PeerToPeer.abs example, one can use the following command line:

```
java -cp javagen:absfrontend.jar PeerToPeer.Main
```

These tasks can be performed equally using the Eclipse IDE with the ABS plugin installed.

6.3 Code Generation

To illustrate code generation for the dynamic ABS Java back end, we show how a few simple ABS code examples (blue boxes) compile to Java using the dynamic Java back end (green boxes), and compare it to the code generated for the regular, static Java back end (grey boxes).

```
class C(Int x) implements I {
   Int getX() { return x; }
}
```

```
public final class C_c extends ABSObject implements ABSClass, I_i {
   private ABSInteger x;
   public C_c(ABSInteger) {...}
   public final ABSInteger getX() {...}
}
```

```
1  public final class C_c {
2     private static ABSDynamicClass c;
3     public static ABSDynamicClass singleton() {
4        if (c == null) {
5           c = new ABSDynamicClass();
6           c.addField("x", new ABSField() { /* Override init */ });
7           c.addMethod("getX", new ABSClosure() { /* Override exec */ });
8        }
9        return c;
10    }
11 }
```

Figure 6: Code generation example: class declaration.

The example in Figure 6 shows the code generation process for class declarations. The generated static Java code is very similar to the original ABS code: the generated Java class `C_c` corresponds to ABS class `C`, and the generated method `ABSInteger getX()` corresponds to `Int getX()`. In the dynamic setting, ABS classes are represented as singleton instances [13] of class `ABSDynamicClass`. The static method `C_c.singleton` (line 3) creates an

`ABSDynamicClass` object (line 5) and adds class `C`'s fields and methods. These are represented as subclasses of `ABSField` and `ABSClosure`. For each method, a new class inheriting from `ABSClosure` is created, which, by overriding the `exec` method, encodes the method's specific behaviour. Similarly, `ABSField` is subclassed for each field, with the overriding `init` method defining the field's specific initialisation expression. Instances of these classes are passed as arguments to `addField` (line 6) and `addMethod` (line 8).

```
C object = new C(42);
Int x = object.getX();
```

```
C_c object = new C_c(42);
ABSInteger x = object.getX();
```

```
1 ABSDynamicObject object = new ABSDynamicObject(c, 42);
2 ABSInteger x = (ABSInteger)object.dispatch("getX");
```

Figure 7: Code generation example: class instantiation and method call.

The example in Figure 7 shows object creation and method calling. In the static setting code generation is straightforward. In the dynamic setting, an object of the predefined class `ABSDynamicObject` is created with a reference to our `ABSDynamicClass` object representing class `C` (line 1). Calling `C`'s method then amounts to calling the `dispatch` method on the `ABSDynamicObject` with the name of the method as an argument (line 2). The `dispatch` method returns a generic `ABSValue` that needs to be cast to the method's specific return type.

6.4 Code generation for dynamic software product lines

The ABS compiler, which translates ABS source code to either Java code or Maude rewriting rules, supports static SPL deployment. At compile time, the user chooses a specific product, which is defined in the model as a set of features together with specific values assigned to feature attributes. Upon code generation, the set of delta modules that are applicable for this product are applied to the core ABS model in a process known as *flattening* of the model. Flattening removes all variability from a model, producing a runnable software product out of the set of products represented by the product line.

The goal of the dynamic ABS back end is to allow the modification of a large number of model elements and aspects at runtime. *Dynamic* SPLs (DSPLs) are executable models of SPLs that can transform their structure

and behaviour from one product to another at runtime. DSPLs are a popular application of dynamically adaptable software and since ABS already has extensive support for static SPLs, extending that support to the dynamic version was a logical development. To that end, a runtime representation of product specifications and delta modules is needed.

The code generated process for ABS product specifications is illustrated in Figure 8 using the example from Figure 4.

```
package products;
public class P1_prod {
   private static ABSDynamicProduct prod;
   public static ABSDynamicProduct singleton() {
      if (prod == null) {
         ABSDynamicProduct prod = new ABSDynamicProduct();
         prod.setName("P1");
         prod.setFeatures(Arrays.asList("F1", "F2", "F3"));
         prod.setUpdate("P2", "Upd1_2");
         prod.setUpdate("P3", "Upd1_3");
         prod.setDeltas("P2", Arrays.asList("AddF3"));
         prod.setDeltas("P3", Arrays.asList("AddF4RemoveF2"));
      }
      return prod;
   }
}
```

Figure 8: Product specification example for dynamically re-configurable product.

The list of deltas that is passed to `setDeltas` when creating the runtime instance of product `P1` is determined at compile time from the product configuration in the same manner as it is done when generating a static product. When generating a static model, these delta modules are applied already at compile time in a flattening process, whereas for the dynamic model, a valid sequence of deltas is simply recorded so that it can be applied at runtime.

The configuration of an SPL product – whether it occurs at compile-time or at run-time – follows the delta-oriented programming approach [22]. This means that in order to obtain the code for a specific product, a list of delta modules is applied in sequence to a program. Delta modules prescribe code modifications such as adding or removing classes, methods, fields, functions and data types. In ABS, delta modules are associated with features in delta clauses using application conditions, as part of the product line configuration [6]). Figure 9 illustrates the representation of delta modules in generated dynamic Java. Note that there is no equivalent representation in generated

```
delta D;
modifies class C {
   adds Int y;
   adds method Int getY() {...}
}
```

```
package delta.D;
public class C_mod {
   public static void apply() {
      ABSDynamicClass cls = C_c.singleton();
      cls.addField("y", new ABSField() { /* Override init */ });
      cls.addMethod("getY", new ABSClosure() { /* Override exec */ });
   }
}
```

Figure 9: Code generation example: delta modules.

static Java code, as the static Java back end applies and then discards deltas prior to code generation. Each class modifier defined by a delta is represented as a class with a static `apply` method that uses the API provided by the dynamic Java back end to apply the modifications to the current representation of the respective class.

7 Conclusion

This paper presents MetaABS, a reflective interface for the ABS language, and a dynamic back end, both designed together to facilitate tasks concerned with runtime model analysis and adaptation. Examples of such tasks, which are detailed in the paper, are user-controlled scheduling policies and dynamic software product lines that enable dynamic product reconfiguration. Other tasks, including the development of safe concurrent and dynamic software updates, are currently being investigated and left as future work.

References

[1] *The ABS Language Specification*, 2011. available at `http://tools.hats-project.eu/download/absrefmanual.pdf`.

[2] David Benavides. *Actor-oriented Metaprogramming*. PhD thesis, University of California, Berkeley, 2004.

[3] Joakim Bjørk, FrankS. Boer, EinarBroch Johnsen, Rudolf Schlatte, and S.Lizeth Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, pages 1–15, 2012.

[4] Joakim Bjørk, Frank de Boer, Einar Johnsen, Rudolf Schlatte, and S. Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, pages 1–15, 2012.

[5] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *ACM International Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[6] Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. Variability modelling in the ABS language. In *Formal Methods for Components and Objects*, volume 6957 of *LNCS*. Springer, 2011.

[7] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, August 2002.

[8] Ferruccio Damiani and Ina Schaefer. Dynamic delta-oriented programming. In *International Software Product Line Conference*, SPLC '11, pages 34:1–34:8. ACM, 2011.

[9] Full ABS Modeling Framework, March 2011. Deliverable 1.2 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[10] ABS system derivation and code generation, September 2012. Deliverable 1.4 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[11] Analysis, September 2012. Deliverable 1.3 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[12] Gianluigi Ferrari, Eugenio Moggi, and Rosario Pugliese. Metaklaim: meta-programming for global computing. In *Proceedings of the 2nd international conference on Semantics, applications, and implementation of program generation*, SAIG'01, pages 183–198. Springer, 2001.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns*. Addison-Wesley, November 1994.

[14] Michiel Helvensteijn. Dynamic delta modeling. In Eduardo Santana de Almeida, Christa Schwanninger, and David Benavides, editors, *16th International Software Product Line Conference, SPLC, Salvador, Brazil, Volume 2*, pages 127–134. ACM, 2012.

[15] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2011.

[16] Einar Broch Johnsen, Marcel Kyas, and Ingrid Chieh Yu. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In Ana Cavalcanti and Dennis Dams, editors, *Proc. 16th International Symposium on Formal Methods (FM'09)*, volume 5850 of *Lecture Notes in Computer Science*, pages 596–611. Springer-Verlag, November 2009.

[17] Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Dynamic resource reallocation between deployment components. In J. S. Dong and H. Zhu, editors, *Proc. International Conference on Formal Engineering Methods (ICFEM'10)*, volume 6447 of *Lecture Notes in Computer Science*, pages 646–661. Springer-Verlag, November 2010.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, 1997.

[19] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[20] Michael Lienhardt, Mario Bravetti, and Davide Sangiorgi. An object group-based component model. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece*, volume 7609 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2012.

[21] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[22] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proc. of 14th Software Product Line Conference (SPLC 2010)*, September 2010.

[23] Ina Schaefer, Lorenzo Bettini, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proceedings of the 14th international conference on Software product lines: going beyond*, SPLC'10, pages 77–91. Springer, 2010.

[24] Jean-Guy Schneider and Markus Lumpe. A metamodel for concurrent, object-based programming. In Christophe Dony and Houari A. Sahraoui, editors, *LMO*, pages 149–166. Hermès, 2000.

[25] Tim Sheard and Simon L. Peyton Jones. Template meta-programming for haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.

[26] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In Alex Aiken and Greg Morrisett, editors, *POPL*, pages 26–37. ACM, 2003.

[27] Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[28] Erwann Wernli, Mircea Lungu, and Oscar Nierstrasz. Incremental dynamic updates with first-class contexts. In *Technology of Object-Oriented Languages and Systems*, volume 7304 of *Lecture Notes in Computer Science*, pages 304–319, 2012.