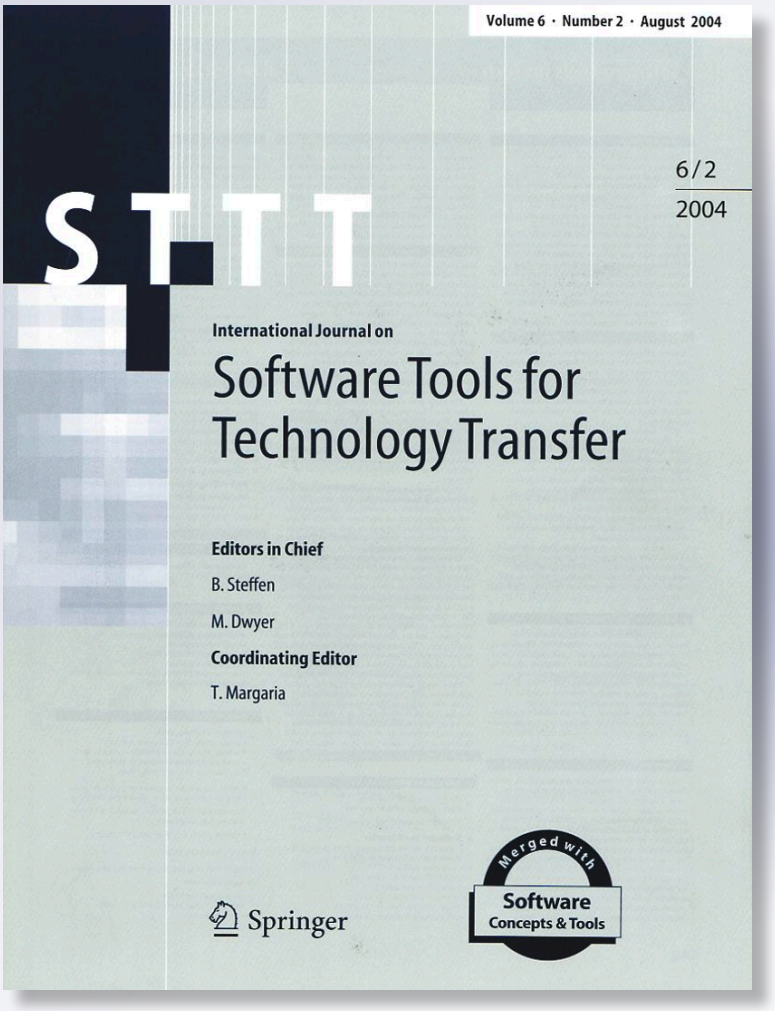
The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems

Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer & Rudolf Schlatte

International Journal on Software Tools for Technology Transfer

ISSN 1433-2779
Volume 14
Number 5

Int J Softw Tools Technol Transfer (2012) 14:567-588
DOI 10.1007/s10009-012-0250-1



Springer

Int J Softw Tools Technol Transfer (2012) 14:567–588
DOI 10.1007/s10009-012-0250-1

SW-DIVERSITY

The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems

Peter Y. H. Wong · Elvira Albert · Radu Muschevici · José Proença · Jan Schäfer · Rudolf Schlatte

Published online: 18 July 2012

Abstract Modern software systems must support a high degree of variability to accommodate a wide range of requirements and operating conditions. This paper introduces the Abstract Behavioural Specification (ABS) language and tool suite, a comprehensive platform for developing and analysing highly adaptable distributed concurrent software systems. The ABS language has a hybrid functional and object-oriented core, and comes with extensions that support the development of systems that are adaptable to diversified requirements, yet capable to maintain a high level of trustworthiness. Using ABS, system variability is consistently traceable from the level of requirements engineering down to object behaviour. This facilitates temporal evolution, as changes to the required set of features of a system are automatically reflected by functional adaptation of the system's behaviour. The analysis capabilities of ABS stretch from debugging, observing and simulating to resource analysis of ABS models and help ensure that a system will remain dependable throughout its evolutionary lifetime. We report on the experience of using the ABS language and the ABS tool suite in an industrial case study.

P. Y. H. Wong (✉)
Fredhopper B.V., Amsterdam, The Netherlands
e-mail: peter.wong@fredhopper.com

E. Albert
Complutense University of Madrid, Madrid, Spain

R. Muschevici · J. Proença
Katholieke Universiteit Leuven, Leuven, Belgium

J. Schäfer
University of Kaiserslautern, Kaiserslautern, Germany

R. Schlatte
University of Oslo, Oslo, Norway

Keywords Formal modelling and analysis · Concurrency · Tool support · Variability · Software product line · Feature modelling

1 Introduction

Diversity is prevalent in modern software systems in order to adapt to their application contexts [53]. Additionally, software systems must evolve to meet changing requirements over time. This may require substantial changes to the software and often results in quality regressions. After a change in a software system, typically some work is needed in order to regain the trust of its users. The HATS (Highly Adaptable and Trustworthy Software using Formal Models) project aims at developing a formal model-centric software development methodology [25,53] for engineering software systems that are subject to frequent changes.

The HATS approach is centred around the *Abstract Behavioural Specification* (ABS) modelling language [26] and an accompanying *ABS tool suite*. ABS allows the precise modelling and analysis of component-based distributed concurrent systems, focusing on their functionality while separating from the concerns such as concrete resources, deployment scenarios and scheduling policies. In particular, the language of ABS provides modelling concepts for specifying variability incrementally from the level of feature models down to object behaviour. This permits large-scale reuse and rapid product construction.

The contributions of this paper is twofold: (1) we present the ABS language and the ABS tool suite and guide this presentation via an industrial case study; and (2) we report on the experience of using the ABS language and the ABS tool suite in an industrial case study.

The ABS tool suite consists of the following tools for assist modeling in ABS:

Compiler front-end The ABS compiler front-end, which takes a complete ABS model of the software system as input, checks the model for syntactic and semantic errors and translates it into an internal representation. The front-end supports automatic product generation: variability of the software system can be resolved by applying the corresponding sequence of delta modules to its core ABS model at compile time.
Code generation There are various compiler back-ends that take the internal representation of ABS models and generate to either executable programs in implementation languages like Java and Scala, or rewriting systems in the language of Maude for simulation and analysis. In this paper, we focus on back-ends for Java and Maude.
ABS plugin There is an ABS plugin that extends the Eclipse integrated development environment (IDE) (http://www.eclipse.org) to provide an ABS IDE. The plugin offers an Eclipse perspective that integrates with the compiler front-end for navigating, editing, visualising, and type checking ABS models. It also integrates with the mentioned back-ends so that ABS models can be executed and simulated directly from the IDE.
ABSUnit During software development in ABS, unit tests are written to quickly validate the correctness of the method implementations and detect regressions due to changes. The ABS tool suite offers the ABSUnit testing framework for writing, managing and executing unit tests in ABS for ABS models.
Dependency management To support industrial collaborative development processes that focus on large-scale reuse, the ABS tool suite provides a package system and a dependency management system for ABS. A package system helps aggregating related ABS modules into platform-independent artifacts for efficient reuse, while the dependency management system helps building, deploying, and reusing such artifacts.
Foreign function interface In industrial software development environments, developers often use external software systems to enhance functionalities of software via a set of application programming interfaces (APIs). To leverage rich APIs and third party libraries offered by popular programming languages, the ABS tool suite offers a set of small extensions to the ABS language, known as the ABS foreign function interface (ABS-FFI), that allows ABS modules to interact with programs written in other programming languages. More importantly, ABS-FFI enables critical parts of a large software system to be formally analysed and verified. This can be achieved by modelling critical parts of the system as ABS modules and connect them to the rest of the system via ABS-FFI.
Visualisation Using the code generation back-ends, the ABS tool suite offers the facility to visualise and to debug running ABS models.
COSTABS One of the analysis techniques offered by the ABS tool suite is resource analysis: COSTABS is a static resource analysis tool for ABS [2]. It is an extension of the COSTA system [1]. COSTABS can be applied to analyse ABS models (and, in particular, to analyse a fragment of our case study) and infer precise information on their resource consumption.

We guide the presentation of the ABS language and the ABS tool suite via an on-going industrial case study based on the Fredhopper Access Server (FAS), a distributed web-based software system for Internet search and merchandising, developed by Fredhopper B.V. (http://www.fredhopper.com). In particular, we consider the *Replication System*; the Replication System ensures data consistency across the FAS deployment. Based on our experience with the case study, we consider the following requirements of ABS, discuss how the ABS language and the ABS tool suite fulfil these requirements and make suggestions on possible improvements. These requirements stem from the activity conducted within the HATS project [30]:

Usability We consider both the ABS language and the ABS tool suite with respect to their overall usability. This means software developers should be able to use the ABS language and tool suite with reasonable effort.
Reducing manual effort We consider how the ABS tool suite helps reducing manual effort and how some of the automated processes offered by the tool suite help reducing the errors that might have been caused by manual operations.
Integrated environment support We consider how the ABS language and tool suite are supported in an integrated environment; specifically, a well-supported set of tools in an integrated environment must have interoperable formats and common visual representation between tools' inputs and outputs. Moreover tools offered must be able to handle large code bases.

The paper is structured as follows: Sect. 2 introduces the Fredhopper Access Server, and in particular the Replication System that is used throughout this paper. Sects. 3 and 4 give an overview of the ABS language. Section 5 describes how to execute ABS models using the ABS tool suite. In this section, we also look at how to unit test ABS models and how to integrate ABS models with foreign languages. Section 6 describes the visualisation and resource analysis facilities offered by the ABS tool suite. Section 7 discusses the experience of using the ABS tool suite during the case study and on how the ABS language and tool suite fulfil the

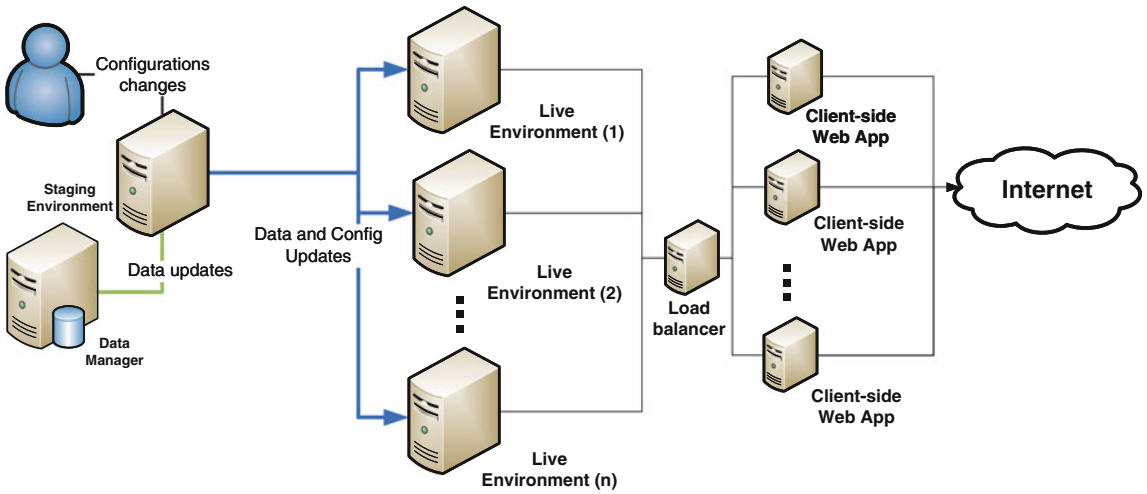


Fig. 1 Example of FAS deployment

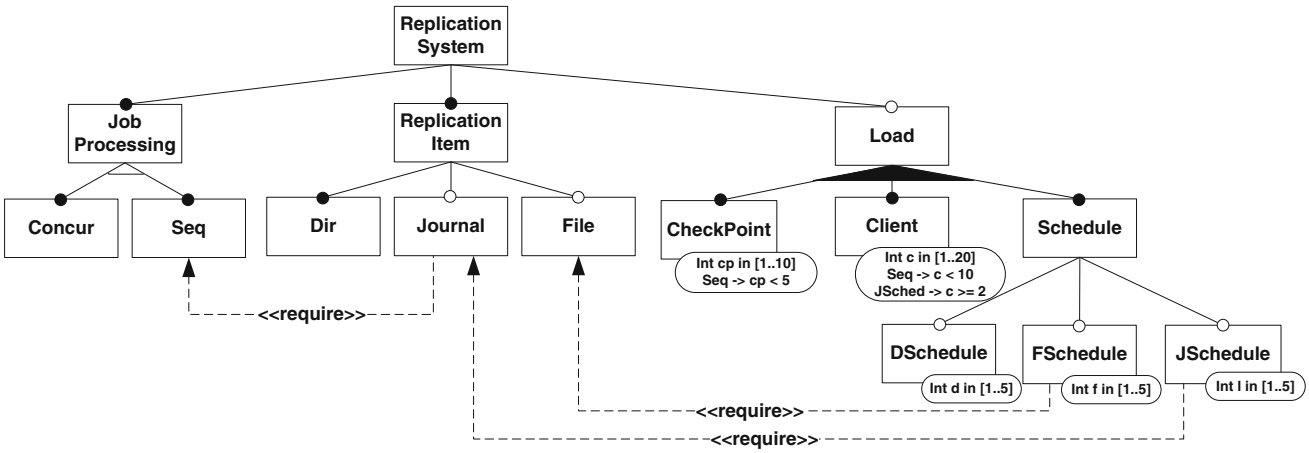


Fig. 2 Feature model of the Replication System

requirements identified in this section. Finally, Sects. 8 and 9 present related work and conclude this paper. A glossary of important terms can be found in Appendix A.

2 Fredhopper case study

The Fredhopper Access Server (FAS) is a distributed concurrent object-oriented system that provides search and merchandising services to e-commerce companies. Briefly, FAS provides to its clients structured search capabilities within the client's data. To minimise the possible disruption caused by data updates in a FAS installation, each FAS installation is deployed according to the FAS deployment architecture. Figure 1 shows an example setup. A FAS deployment consists of a set of "environments". In this case study we focus on two types of environments: live and staging. 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 [57]. The replication protocol is implemented by the *Replication System*. The Replication System consists of a SyncServer at the staging environment and one SyncClient for each live environment. The SyncServer determines the schedule of replication, as well as the items of replication, while each SyncClient is responsible for initiating communication with the SyncServer, receiving replication schedules from the SyncServer, and scheduling replication jobs according to those schedules to receive replication items.

2.1 Variability

There exist several variants of the Replication System. We express these variants by *features*. A feature is a product characteristic that is relevant to some stakeholder in the development project. Features are organised in a *fea-*

ture model [19,43], essentially a set of logical constraints expressing the dependencies between them. Feature models are usually represented graphically as feature diagrams. Figure 2 shows the feature diagram of the Replication System. Specifically, the feature diagram defines a set of legal feature combinations. They represent the set of valid Replication System variants that can be built from the given features. A feature diagram is a tree structure in which each node denotes a feature and is annotated with the name of that feature. Relationships between a parent and its child features are categorised as follows: child features with a black dot attached denote mandatory features, while child features with a white dot attached denote optional features. The root node denotes the root feature and is by default mandatory. If the edges from the parent to its child features are shaded in black then *one or more* of the child features must be selected if the parent feature is selected. If the edges are shaded in white then *exactly one* of the child features must be selected if the parent feature is selected. Otherwise, all of the child features must be selected. Dotted lines between features specifies orthogonal constraints between connected features. Going back to the case study, the feature diagram of the Replication System shown in Fig. 2 has three main features: `Job Processing`, `Replication Item` and `Load`. Feature `Job Processing` offers one of sequential and concurrent client job processing, feature `Replication Item` offers various supports for different types of replication items, for example sub-feature `File` supports replicating a file set, whose files' name matches a particular pattern, and feature `Load` offers supports to configure resources, replication schedules and the number of clients. Reader can find more details of individual features in [57].

3 The core ABS language

In the HATS methodology the ABS language is used to precisely define the behaviour of distributed software systems. ABS itself consists of a core called *Core ABS*, which is used to model single software systems, and Full ABS, an extension on top of the core to support variability modelling. This section describes Core ABS, whereas Full ABS is described in Sect. 4.

Core ABS (or simply ABS in the following) is a concurrent, multi-paradigm modelling language. Syntax-wise, ABS resembles standard programming languages like Java. In fact, the syntax of ABS was deliberately designed to be as close as possible to existing programming languages in order to lower the entry barrier to use ABS. 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.

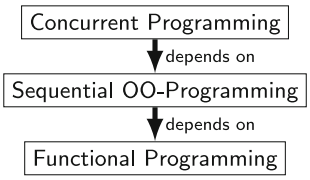


Fig. 3 Layered language design of ABS

ABS combines functional, object-oriented, and concurrent programming in a single language. The design follows a strict layered approach, where the properties from a lower layer cannot be invalidated by a higher layer (see Fig. 3). In addition, higher layers depend on lower layers but not vice versa.

In the remainder of this section we describe each language layer in more detail. A complete description of all ABS features can be found in [7,38].

3.1 Sequential programming

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. Having such a functional core makes it possible to describe large parts of a software system in a side-effect-free way to simplify reasoning. For brevity, we omit details of the functional sub-language and refer readers to [7,38] for its full descriptions.

ABS also supports class-based, object-oriented programming with standard imperative constructs. 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*, which are described in Sect. 4.2.

3.1.1 Interfaces

Interfaces define types for objects. They are nominal, that is, have a name, and define a set of method signatures, that is, the names and types of callable methods. Syntactically, ABS interfaces look like Java interfaces. ABS does not support type parameters for interfaces, that is, generic interfaces are not possible.

The following listings show some interfaces from the Replication System, namely a collection of interfaces to represent streams. `FB<A>` is an algebraic data type representing the result of reading from and writing to an I/O stream. Values of this type can be constructed by either the constant `Ok`, wrapping some value of type `A` using the constructor `Result` or defining an error message using the constructor `Error`.

```
data FB<A> = Ok | Result(A) | Error(String)
interface Reader { Int readInt(); }
interface Writer { Unit writeInt(Int i); }
interface Input { FB<Int> readInt(); }
interface Output { FB<Unit> writeInt(Int s); }
interface Stream extends Input, Output { ... }
```

Listing 1 Streams I/O

3.1.2 Classes

```
class ReaderImpl(InputStream i) implements Reader {
  Int readInt() {
    Fut<FB<Int>> ib = i!readInt();
    FB<Int> fb = ib.get;
    return result(fb);}}
```

Listing 2 Implementation of Reader

Classes define the implementation of objects. In contrast to Java, for example, classes do *not* define a type. Classes can implemented arbitrarily many interfaces, which then define the type of a new instance of that class. A class has to implement all methods of all its implementing interfaces. In addition, a class can define *private* methods which do not appear in any interface. Such methods can only be invoked on **this**. Instead of constructors, classes in ABS have *class parameters* and an optional *init block*.

Listing 2 shows the class `ReaderImpl` from the replication case study that implements the `Reader` interface.

3.1.3 Statements and expressions

ABS has standard statements and expressions known from languages such as Java, with identical syntax. Beside side-effect-free expressions on built-in data types, expressions can be method invocations (`x.m(a)`), object creation (**new** `C(a)`), and field and variable reads and assignments (`x =` **this**`.y`). There is also a conditional statement, a **while** loop, and the **skip** statement which does nothing.

3.2 Concurrent programming

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.

3.2.1 Concurrent object groups

A new COG is created by using the **new cog** expression. It takes as argument a class name and optional parameters and returns a reference to the initial object of the new COG. The following line of code from the Replication System case study shows the creation of a new `ServerImpl` COG. As `ServerImpl` is an active class (it has a `run` method), it also starts running concurrently to the creating COG.

```
new cog ServerImpl(...);
```

Location type system. To be able to statically distinguish references to objects of other COGs (*far references*) from references to objects of the same COG (*near references*), ABS provides a pluggable type extension that introduces *location types* (see [59] for details). The extension is realised by using *type annotations*, which is a built-in mechanism of ABS to extend the type system. A location type annotation is either `[Near]`, `[Far]`, or `[Somewhere]`, where `[Somewhere]` means that the reference is either near or far. An integrated type inference mechanism automatically infers most location type annotations.

3.2.2 Asynchronous method calls

Communication between COGs may solely be done 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 and return a future instead of a normal value (see Sect. 3.2.3). Listing 3 shows partially the definition of the class `ClientJobImpl` that models replication jobs and is used by the SyncClient for communicating asynchronously with the SyncServer. Note that we will refer back to the listing in Sect. 5.6.

3.2.3 Futures

Asynchronous method calls return *futures*. A future is a place-holder for the result of the method call. Initially, a future is *unresolved*. When the called method has terminated, the future will be *resolved* (automatically) with the result value of the call. The caller can thus later synchronise with the future and obtain the result value of the method call.

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. To obtain the value from a future, the **get**-expression (`f.` **get**) is used. It 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 between COGs can be simulated in ABS by performing an asynchronous method call and waiting for the resolved future using the **get**-expression. For example, the `setDb` method in Listing 3 blocks the COG until the database has been retrieved from the client.

```
class ClientJobImpl(Client c, ...) {
  DataBase db;
  Unit setDb() {
    Fut<DataBase> fd = c!getDataBase();
    db = fd.get; }
  Unit run() {
    this.setDb();
    Bool connected = this.connect();
    while (~connected) {
       await duration(10,10);
       connected = this.connect();
    } ... }
  Bool hasFile(Fn id) {
    Fut<Bool> he = db!hasFile(id); await he?;
    return he.get; }
  Maybe<Size> processFile(Fn id) {
    Maybe<Size> result = Nothing;
    Fut<Set<Fn>> ff = db!listFiles();
    await ff?; Set<Fn> fids = ff.get;
    if (contains(fids,id)) {
      Bool hf = this.hasFile(id);
      if (hf) {
        Fut<Content> fc = db!getContent(id);
        await fc?; Content c = fc.get;
        if (isFile(c))
           result = Just(content(c)); } }
    return result;}}
```

Listing 3 Partial implementation of `ClientJobImpl`

3.2.4 Cooperative multi-tasking

Each asynchronous method call results in a *task* in the COG of the target object. Tasks are scheduled *cooperatively* within the scope of a COG. Cooperative scheduling means that switching between tasks of the same COG happens only at specific *scheduling points* during program execution, which are apparent in the source code, and that at no point two tasks in the same COG are active at the same time. Hence, the state of a COG can never be accessed by two tasks at the same time and, in addition, interleaving points can be syntactically identified and analysed.

The **suspend** statement introduces an *unconditional* scheduling point, causing the running task to be suspended and another task of the COG to be scheduled. With the **await** statement, one can create a *conditional* scheduling point, where the running task is suspended until after the specified condition becomes true. The **await** statement can be used to suspend a task until a future becomes resolved (inter-COG synchronisation) or a Boolean condition over the object state becomes true (synchronisation between tasks in the same COG). While waiting for a future, other tasks can run. For example, the `processFile` method in Listing 3 defines a conditional scheduling point, suspending the running task to wait for the content of a file, while allowing other tasks to run in between.

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 (that is, establishes all the object invariants[1]). At all other points, objects are implicitly protected against concurrent modifications of their fields.

3.3 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*. 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.

The following listing shows a module header from the Replication System case study.

```
module ReplicationSystem.Streams;
export *; import * from Files; ...
```

4 Full ABS

The HATS methodology is geared towards modelling highly configurable systems. Such systems have a high degree of variability to accommodate different requirement and deployment scenarios.

ABS provides language constructs and tools for modelling variable systems following Software Product Lines (SPL) [48] engineering practices. The *Micro Textual Variability Language* μTVL [28] is used to model all products of an SPL by using features and feature attributes. A μTVL model hence describes a feature model. A *Product Selection* identifies individual products that are of particular interest.

[1] An object invariant is a property about values of the object's fields that must be satisfied when the object is created, and before and after any of its methods is invoked.

Fig. 4 Generation of a software product

Delta modules are reusable units of ABS code which can be applied incrementally to an ABS model to adapt its behaviour to conform to a particular product. Finally, the *Configuration* associates features to delta modules, enabling us to generate the ABS model for individual products by naming a product. The Core ABS language (cf. Sect. 3) enriched with the above extensions is called the *Full ABS*.

Figure 4 depicts the main steps required to build a software product using ABS. The developer first selects the desired features. This selection is then used to choose the relevant delta modules. Each of these delta modules is applied in a particular sequential order to the core model. The application of all relevant delta modules results in a software product with the desired features.

The remainder of this section details the ABS language constructs used for modelling software product lines. Section 4.1 introduces feature models and describes how to select valid products. The delta modelling approach is described in Sect. 4.2, and the connection between features and deltas is made in Sect. 4.3. We exemplify the approach, using the case study, in Sect. 4.4. Finally, we describe our platform and deployment model in Sect. 4.5. A complete reference of ABS, including the constructs for modelling variability, can be found in [7,28].

4.1 Feature model

Our example software product line of Replication Systems is represented graphically by the feature diagram in Fig. 2. Listing 4 shows how the underlying feature model is encoded in ABS using the textual variability language μTVL.[2]

A μTVL feature model is encoded as integer and boolean constraints over features and feature attributes. Each solution for these constraints represents a valid product of the feature model. ABS allows the developer to name products that are of particular interest, in order to easily refer to them later when the actual code needs to be generated. Listing 5 shows examples of valid products. For example the product `DS` selects the features `Dir` and `Seq`. The parent nodes of these two features are automatically included.

Both products mentioned here are valid since they satisfy the constraints associated with the feature model. In the product `DFSCCDF` the feature `Client` must include an assignment of all of its attributes; in this case `c` is initialised with the value 2. The μTVL checker that comes with the ABS tool suite can evaluate all product selections with respect to the feature model and warn about invalid products.

[2] μTVL is based on the textual variability language TVL [20].

```
root ReplicationSystem {
 group allof {
  JobProcessing {
   group oneof { Seq, Concur } },
  ReplicationItem {
   group [1..*] {
    Dir, opt Journal { require: Seq; }, opt File }}
  opt Load {
   group [1..3] {
    Client {
     Int clients in [1 .. 20];
     Seq -> (clients < 10); },
    CheckPoint { ... },
    Schedule {
     group [0..3] {
      DSChedule { ... },
      FSChedule { ... },
      JSChedule { ... }}}}}}}
```

Listing 4 Feature model of the Replication System

```
product DS(Dir, Seq);
product DFSCCDF(Dir, DSchedule{d=1}, Client{c=2},
 File, Seq, CheckPoint{cp=2}, FSchedule{f=1});
```

Listing 5 Product selections for the Replication System

4.2 Delta model

ABS supports delta-oriented programming [52], an approach that aims at developing a set of programs simultaneously from a single code base, following the SPL engineering approach [48]. In delta-oriented programming, features defined by a feature model (as described in Sect. 4.1) are associated with code modules that describe modifications to a *core* model. These modules are called *delta modules* (or *deltas* for short). Hence the implementation of a software product line is divided into a core model and a set of delta modules. The core model consists of the classes that implement a complete product of the corresponding product line. Delta modules describe how to change the core model to obtain new products. This is done by adding new classes, modifying existing ones, or even removing some classes. The choice of which delta modules to apply is based on a selection of desired features for the new product. Delta modules are associated with features through *application conditions*. Application conditions are logical expressions that evaluate to either true or false, given a particular feature and attribute selection. If a delta module's application condition is true, then the delta module is *applied*, meaning that the core model is modified according to the changes described by the delta module. By not associating delta modules directly with

features, a degree of flexibility is obtained, resulting in better reuse of code and the ability to resolve conflicts caused by delta modules modifying the code base in incompatible ways.

Delta modules' granularity is at the level of methods and fields. They can add new methods to classes, and remove or modify existing methods. The example below shows a delta module that modifies the class `Main`, which is assumed to have been declared in the core.

```
delta DSchedule(Int s);
modifies class Main {
 modifies List<Schedule> getSchedules() {
  List<Schedule> ss = original();
  return take(ss,s);}}
```

Listing 6 Delta module `DSchedule`

The `DSchedule` delta module provides a new implementation for the method `getSchedules()` in class `Main` by defining a so-called method modifier, introduced by the **modifies** keyword. Adding and removing methods is also supported using the **adds** and **removes** keywords. Calling the special method **original**`()` makes it possible to access the behaviour a method had before the application of the delta, similar to the use of `super` to access superclasses in Java. In addition to modifying object behaviour, ABS delta modules allow adding or removing fields. New fields are introduced using the **adds** keyword and removed using the **removes** keyword. Finally, ABS also supports manipulating the list of interfaces that classes implement by a similar addition and removal mechanism.

Delta modules define an optional list of parameters which can be used to pass the values of feature attributes defined in the product selections (Sect. 4.1) into the delta body. These parameters must be immutable objects, such as integers, booleans or strings. In the above example, any occurrence of the integer variable `s` inside the delta is replaced with the concrete value of the feature attribute `SSchedule.s` upon delta application.

4.3 Software product line configuration

The ABS *configuration* language links feature models, which describe the structure of an SPL (Sect. 4.1), to delta modules (Sect. 4.2), which implement its behaviour. This link is illustrated in Fig. 5. The configuration defines, for each selection of features satisfied by a product selection, which delta modules should be applied to the core model.

Features and delta modules are associated through *application conditions*, which are logical expressions over the set of features and attributes in a feature model. The collection of applicable delta modules is given by the application conditions that are true for a particular feature and attribute selection.

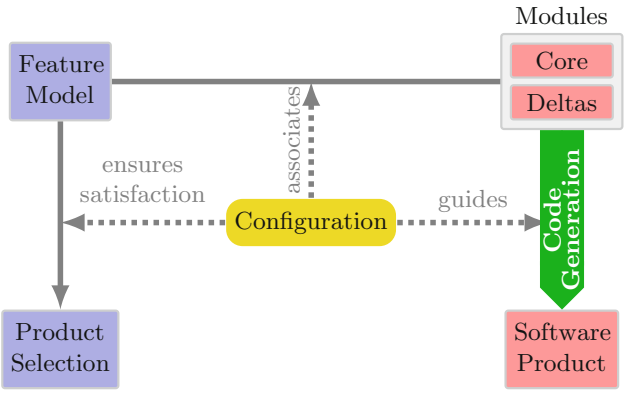


Fig. 5 Variability modelling framework of ABS

A configuration block specifies the name of the product line, the set of features it provides, and the set of delta modules used to implement those features.

```
productline ReplicationSystem;
features Dir, File, Journal, Seq, Concur, Client,
  Schedule, CheckPoint, DSchedule,
  FSchedule, JSchedule;
delta File when File;
delta Journal when Journal;
delta Concurrent when Concur;
delta Client(Client.c) when Client;
delta CP(CheckPoint.cp) when CheckPoint;
delta DSchedule(DSchedule.s) when DSchedule;
delta FSchedule(FSchedule.f)
  after DSchedule when FSchedule;
delta JSchedule(JSchedule.l)
  after FSchedule when JSchedule;
```

Listing 7 Product line configuration for replication items

The example in Listing 7 first names the set of **features** from the feature model in Fig. 1 used to configure this product line, followed by a set of **delta** configurations. Each delta configuration contains at least the name of the delta module to be applied and an application condition introduced by the **when** keyword. For example, the delta configuration "`Concurrent` **when** `Concur;`" states that the delta `Concurrent` is only applied if the feature `Concur` is selected. Deltas can be parametrised and their arguments can include attribute variables, as exemplified by the `Client` delta configuration.

Finally, delta configurations can also include an **after** clause to mediate *conflicts*, as shown in the configuration of the `FSchedule` and `JSchedule` deltas. Two deltas are in conflict if their specified modifications do not commute. An **after** clause resolves conflict by imposing an order of application for certain deltas. A more elaborate mechanism for reconciling conflicting feature functionality in ABS has been presented [34].

4.4 Product generation

```
class Directory(Fn q, DataBase db)
  implements Item { ... }

class SnapshotImpl(DataBase db, Set<Schedule> ss)
  implements Snapshot {
  Set<Item> items = EmptySet;
  ...
  // used by 'File'
  Unit item_original(Schedule s) { ... }
  // modified by 'File'
  Unit item(Schedule s) { item_original(); ... }}

// added by 'File'
class FileSet(Fn q, String p, DataBase db)
  implements Item { ... }

class Main {
  ...
  // added by 'File'
  Set<Schedule> files = ...
  // modified by 'Client' and 'CP', respectively
  Map<Int,String> getCids() { ... }
  Map<CP,Map<Fn,Content>>> getDatas() { ... }
  // modified by 'DSchedule' and later by 'FSchedule'
  Schedule getSchedules() { ... }}
```

Listing 8 Core ABS model for the product DFSCCDF

We now summarise the full process of generating a software product from our example SPL. The Replication System modelled in ABS defines a feature model, a set of delta modules, and a product line configuration. For each selection of features, a new Core ABS program is generated. In the following we select the product `DFSCCDF`, introduced in Listing 5, which includes the features `Dir`, `File`, `Seq`, `Client`, `CheckPoint`, `DSchedule` and `FSchedule`. The generation of the final software product proceeds as follows.

1. Check if the product is valid with respect to the feature model. In this case the product `DFSCCSF` is valid.
2. Find all applicable delta modules. In this case the deltas modules with an application condition that holds are `File`, `Client(2)`, `CheckPoint(2)`, `DSchedule(1)` and `FSchedule(1)`.
3. Linearise the application of deltas. The restriction here is over the delta module `FSchedule`, which has to follow the `DSchedule` delta. In this case, a valid order of application is: `File`, `Client(2)`, `CheckPoint(2)`, `DSchedule(1)` and `FSchedule(1)`.
4. Apply the deltas sequentially. We start by applying the `File` delta to the Core ABS model. We then apply the `Client(2)` delta to the result of the previous application, and proceed until all selected delta modules have been applied.

The resulting software product is the Core ABS program shown in Listing 8.

4.5 Platform and deployment modelling

ABS, being a modelling language, can be used to express both functional and non-functional properties of systems. In distributed systems, an important concept is that of *locality*, that is, the physical location of parts of the system. This is important, among other things, because communication cost between objects, server load, memory use and runtime behaviour depend on the deployment configuration of the system.

ABS offers the notion of *deployment components*, which is a way of expressing both the location of a COG and an abstract view of the resources offered by that location. A deployment component is created like a normal ABS object, but has no methods or active behaviour. Hence, its presence does not influence the functional behaviour of the model. However, a new COG can be associated with a deployment component; this models that this part of the system runs "on" that component.

```
class System(Map<CP,Map<Fn,Content>> is,
  List<Schedule> ss, Map<Id,String> cids) {

  Unit run() {
    DeploymentComponent s =
      new DeploymentComponent("s1",
        set[URL("http://s1/"),CPUCapacity(8)]);

    [DC:s] new cog ServerImpl(is,ss,keys(cids));

    List<Id> keys = keyList(cids);
    while (length(keys) > 0) {
      Int id = head(keys); keys = tail(keys);

      DeploymentComponent c =
        new DeploymentComponent(intToString(id),
          set[URL(lookup(cids,id)),CPUCapacity(4)]);

      [DC:c] new cog ClientImpl(id,sp); }}}
```

Listing 9 `System` class

The example shown in Listing 9 is part of the Replication System case study and defines a class that sets up a runtime structure modelling the distributed Replication System running on machines with the specified amount of resources. The SyncServer runs on a 8-core processor, while each SyncClient requires an 4-core processor.

One of the code generation back-ends provided by the ABS tool suite is the Maude back-end (c.f. Sect. 5.2). The Maude back-end supports simulation of models under CPU resource constraints, where the cost of executing a statement (as specified by the modeller via annotations) influences the time that is needed for the simulation to complete. This allows

reasoning about deployment architectures, such as the influence of a faster machine for a part of the system on the performance of the whole system. More formal descriptions of this work can be found in [12,41].

5 Tool support for ABS

Figure 6 gives an overview of the current ABS compiler framework. The ABS compiler takes an ABS model as input, which includes the feature model and product selection over the feature model (Sect. 4.1), delta modules (Sect. 4.2), a product line configuration (Sect. 4.3) as well as a Core ABS model (Sect. 3).

Two different back-ends translate ABS models into either Maude [24] or Java, which allow ABS models to be executed and analysed on these platforms. The abstract syntax tree (AST) is the cornerstone of tool integration: it is the internal representation for ABS models, and tools will typically reason about one or more such models or produce them. All tools will work on a common representation of the AST, which has the following benefits:

- *Reduced implementation costs* individual tools need not know about concrete model syntax and type-checking.
- *Easier integration* the output from one tool can be the input to another, or several tools can cooperate to produce results in the same format.

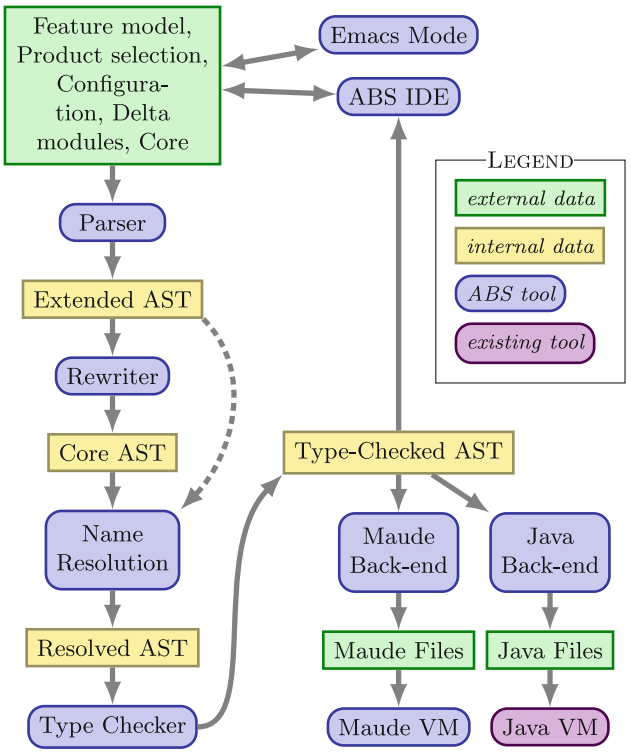


Fig. 6 Overview of the ABS compiler framework

For simplicity Fig. 6 is not exhaustive; for example, it does not describe our tools for resource analysis or debugging of type-checked ASTs, discussed in the next section.

5.1 Modelling with the ABS tool suite

The ABS tool suite offers the ABS Eclipse IDE that integrates most of our ABS-related tools. This plugin can be installed as a bundle via its Eclipse update site tools.hats-project.eu/update-site.

The ABS *compiler front-end* translates textual ABS models into an internal representation and checks the models for syntax and semantic errors. This internal representation is used by the analysis tools described in Sect. 6. The representation of the feature model is further used to search valid combinations of features and to validate the existing product selections. The *compiler back-end* generates code from the models, targeting some suitable execution or simulation environment.

In this section we describe how to compile and run ABS models, how to package ABS projects, and how to perform unit tests. ABS models are compiled into Maude for simulation and verification purposes, or into Java for integration with existing systems. Furthermore, we explain how Java programs generated from ABS models using the Java back-end can interact with other existing Java libraries using a ABS-FFI.

5.2 The Maude back-end

The ABS tool suite is able to translate ABS code to the format of rewriting logic, which is executable on the Maude engine. The AST of an ABS model is converted into Maude terms that are then used for simulation by an interpreter written in Maude. The translation can be done from the command line or from within Eclipse or Emacs. The rationale behind the Maude back-end is to have an execution platform that

1. allows for easy inspection of object and process states, messages and COGs;
2. has an output that is accessible by visualisation and other tools;
3. is easily extensible with additional semantics, e.g., for simulating time, resource consumption or runtime variability;
4. has a formal semantics that is amenable to tool-based analysis.

The Maude interpreter is designed for dynamic adaptation of ABS models. This will allow to convert Full ABS models, which include product selections, configurations and deltas , into Maude terms. The Maude interpreter will support the runtime reconfiguration of models, that is, the ability

to reconfigure a model to represent a different product of a given SPL, or to dynamically add and remove deltas from a running product.

The result of running an ABS model on the Maude interpreter is a plain-text representation of the state of the entire model comprising classes, futures, COGs and object instances with their state and process queue. This output can be used for visualisation or analysis purposes.

5.3 The Java back-end

The Java back-end takes the type checked AST of an ABS model and produces a collection of Java classes that encode the ABS model. The rationale behind having a Java back-end is to have an execution platform for ABS models that:

1. is more scalable to handle the execution and the testing of very large ABS models;
2. is highly configurable for testing and applying different scheduling strategies;
3. allows for easy observation and for integration of additional tools written in standard Java;
4. makes it possible to directly use generated code from ABS models in systems that are written in standard Java.

In contrast to Maude code (Sect. 5.2), the generated Java code cannot be easily used for analysis purposes, but it includes hooks for tracing, scheduling and debugging, and its execution speed is higher.

For every main block that exists in an ABS model, a corresponding `Main` class that contains a standard Java `main` method is generated. Thus, the generated Java code can be executed like any other standard Java code. The generated Java code relies on a runtime library (included in the `absfrontend.jar`), which must be provided when executing the system.

5.4 Foreign function interface

In industrial software development environments, developers often use external software systems to enhance functionalities of software via a set of application programming interfaces (APIs). External software systems include third party libraries and existing in-house software components.

Similarly, the Replication System integrates with external software systems to provide the following functionalities:

– Real-time job scheduling,
– Distributed execution of SyncClients and SyncServer,
– Communication between SyncClient and SyncServer via stream sockets,
– Read and write to physical file store,
– Read and write to standard output.

While it is possible to develop standard APIs to realise these functionalities at the level of ABS, it is not possible, for example, to anticipate formal specification of existing third party libraries using ABS. As a result, the ABS tool suite offers a set of small extensions to the ABS language, known as the ABS ABS-FFI, that allows ABS modules to interact with software systems written in other existing programming languages. In addition, ABS-FFI enables critical parts of a large software system to be formally analysed and verified. This can be achieved by modelling critical parts of the system in ABS and connect the ABS definitions to the rest of the system via ABS-FFI. Currently ABS-FFI supports communication between ABS and Java classes.

```
interface In { FB<Int> readInt();}
interface Out{ FB<Unit> writeInt(Int s);}
```

Listing 10 `Input` and `Output`

As an example, we show how to use ABS-FFI to enable stream sockets based communication between SyncClient and SyncServer. Listing 10 shows two simple interfaces `In` and `Out` for reading an integer from and writing an integer to a stream.

We provide a base implementation in ABS that models a stream. This is implemented by ABS classes `Input` and `Output` shown in Listing 11, which uses the class `Stream` that encapsulates a `List` to model the underlying implementation of a stream. This base implementation is also used at runtime when no foreign implementation is provided, for example in the Maude back-end.

```
module Stream; import * from ABS.FFI;

[Foreign] class Input(In s) implements In {
 FB<Int> readInt() { return s.readInt(); }}

[Foreign] class Output(Out s) implements Out {
 FB<Unit> writeInt(Int i) {
   return s.writeInt(i);}}

class Stream implements In, Out {
  List<Data> buffer = Nil;
  FB<Int> readInt() {
    FB<Int> fb = Error("Cannot read string");
    if (isJust(int(last(buffer)))) {
      Data e = last(buffer);
      buffer = front(buffer);
      fb = Result(fromJust(int(nt))); }
    return fb; }

  FB<Unit> writeInt(Int s) {
    buffer = Cons(Int(s),buffer); return OK;}}
```

Listing 11 The `Stream` module

The ABS classes `Input` and `Output` are annotated with a `Foreign` annotation. This means a foreign implementation can be provided. To provide a foreign implementation in the Java back-end, we subclass the generated Java classes `Input_c` and `Output_c` and override the methods whose name have the prefix `ffi_`, as shown in Listing 12.

```
public class Input extends Input_c {
  public FB<ABSInteger> ffi_readInt() {
    try {
      ABSInteger i = fromInt(stream.readInt());
      return new FB_Result<ABSInteger>(i);
    } catch (IOException e) {
      ABSString s = fromString(e.getMessage());
      return new FB_Error<ABSInteger>(s); }}}

public class Output extends Output_c {
  public FB<ABSUnit> ffi_writeInt(ABSInteger s) {
     try {
       stream.writeInt(s.toInt());
       return new FB_OK<ABSUnit>();
     } catch (IOException e) {
       ABSString s = fromString(e.getMessage());
       return new FB_Error<ABSUnit>(s); }}}
```

Listing 12 Java Implementation of `Input` and `Output`

5.5 Dependency management

At the language level, ABS supports incremental development and reuse via a combination of object composition, module systems and delta modelling. As a result ABS is very suitable for modelling adaptable and evolvable software systems such as the Replication System. In order to support industrial collaborative development processes that focus on large scale reuse, the ABS tool suite also provides a package system and a dependency management system for ABS code. A package system helps aggregating related ABS modules into a platform-independent artifact for efficient reuse, while the dependency management system helps building, deploying, and reusing artifacts.

5.5.1 ABS package

The ABS package format (APK), based on the Java Archive (Jar) format, is an open source, platform-independent file format. APK aggregates many (related) ABS modules into a single unit and represents a single point of reference to that unit. For example, we package the SyncClient-related ABS modules containing classes such as `ClientImpl` and `ClientJobImpl` into a APK file `client.jar`, while we package the SyncServer-related ABS modules containing classes such as `ServerImpl` and `AcceptorImpl` into a APK file `server.jar`.

5.5.2 Package dependency

The Apache Maven tool (http://maven.apache.org) can be employed to help managing dependencies between FAS components as well as dependencies of third party libraries.

Maven is an open source Java-based tool that consists of a project object model (POM), a set of standards, a project lifecycle, and an extensible dependency management and build system via plug-ins. To support industrial collaborative development processes that are characterised by large scale reuse, the ABS tool suite provides a Java-based plugin that extends Maven to support the ABS language.

5.6 ABSUnit: unit testing with ABS

During the development of ABS models, unit tests are written to quickly validate the correctness of class methods and detect regressions. A unit test exercises a unit of functionality of a system under test (SUT), which is usually at the level of public class methods, and makes assertions about the state of that system after the unit's execution.

The ABS tool suite comes with the ABSUnit testing framework for writing unit tests for ABS. This framework is based on the original xUnit architecture [21] that inspires many unit test frameworks such as JUnit (http://junit.org). It consists of an ABS package and a test runner generator.

The ABS package contains modules that provide the necessary mechanism to define fixtures, test cases, checks and test suites. ABS annotations are used to declare which classes define the fixture, which methods define test cases and which methods define test data.

Listing 13 shows the interface `ClientJobTest`. The interface itself is annotated with `Fixture`, which denotes that the implementations of this interface are to be treated as fixtures and test cases. Methods of interface `ClientJobTest` are also annotated. Implementations of methods annotated with `Test` are to be executed as test cases while implementations of methods annotated with `DataPoint` return a list of test data that serve as input to test methods that take input of the same type.

```
type Data = Map<Fn,Maybe<Size>>;
[Fixture] interface ClientJobTest {
  [Test] Unit testProcessFile(Data ds);
  [DataPoint] List<Data> getDatas();}
```

Listing 13 Interface `ClientJobTest`

Listing 14 shows a part of the class `Suite` that implements the methods `testProcessFile` and `getDatas` from the interface `ClientJobTest`. The method `testProcessFile` defines a test case on the method `processFile(Fn)` of `ClientJobImpl`, which was shown in Listing 3 (Page 572).

```
interface Job {
  Maybe<Size> processFile(Fn id);
  Unit setDB(DataBase db);}

[Suite] class TestImpl implements ClientJobTest {
  Set<Data> datas = ... ABSAssert aut = ...
  Set<Data> getDatas() { return datas }
  Job getCJ(DataBase db) { return null; }
  Unit testProcessFile(Data ds) {
    DataBase db = new cog TestDataBase(ds);
    Job job = this.getCJ(db);
    List<Fn> ids = keyList(ds);
    while (length(ids) > 0) {
      Fn i = head(ids); ids = tail(ids);
      Maybe<Size> s = job.processFile(i);
      Comparator cmp = ...;
      aut.assertEquals(cmp); }}}
```

Listing 14 Defining test cases using `ABSUnit`

The class `TestImpl` uses a number of features from module `AbsUnit` to assist defining test oracles. Specifically, `TestImpl` has an instance `ABSAssert` from `AbsUnit` that provides the method `assertEquals(Comparator)`. This method takes a comparator (`Comparator`), which knows how to compare two instances of a specific kind.

Some well-known problems are usually encountered when trying to unit-test existing objects, namely inaccessible object state and, in the case of ABS, uncontrollable active behavior. Mock objects and test-first development are used to solve these issues, but the delta mechanism of ABS can additionally be used to make an object testable.

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

Listing 15 Delta module `JobTestDelta`

Listing 15 shows the delta module `JobTestDelta` that is used to modify the SUT such that `ClientJobImpl` now implements an additional interface `Job` that provides a method `setDB(DataBase))` to access the field `db`. Moreover the delta also removes `run()`. The delta module `JobTestDelta` also modifies `getCJ(Database)` method such that the method returns an instance of `ClientJobImpl` with method's input `db` being set to the instance field `db`.

The test runner generator of the ABSUnit framework is built into the ABS front-end. It takes the ABS model for the SUT and the test suites, parses the files and type checks the ABS definitions, and returns an ABS module `AbsUnit.TestRunner` containing a main block that executes all test cases from the test suites concurrently.

6 Visualising, debugging and analysing ABS

This section describes tools for visualising and analysing ABS models.

6.1 Visualisation and observation

As ABS is a modelling language, it has no built-in I/O support. When not using the ABS-FFI (cf. Sec. 5.4), executing an ABS model produces no visual output. Depending on the used back-end, there are different ways for visualising and observing ABS model executions. When using the Maude back-end, it is possible to observe the final (or intermediate) system states by looking at the Maude representation of the system. When using the Java back-end, however, there is no such representation. Instead, the Java back-end offers a flexible plugin mechanism for writing ABS observers in Java. The ABS tool suite comes with a number of predefined observers. For example, the `ConsoleObserver` prints a log trace to the standard output, which can be used for debugging. A more interesting observer is the UML sequence diagram generator. During debugging or execution of the Java program generated from an ABS model, The UML sequence diagram generator can be started from Eclipse as a separate application. The generator observes the execution and generates UML sequence diagrams of the high-level communication between COGs.

6.2 Interactive debugging

The ABS Eclipse plugin provides a debugging perspective that can be used to interactively debug an ABS system. The perspective offers several views for observing the current state of the system, that is, the COGs and the stack trace of each task. The user has full control over all scheduling decisions, so that any possible execution path can be simulated. Scheduling decisions can also be stored to a file to be replayed later.

6.3 Resource analysis: COSTABS

One of the most important characteristics of a program is its cost or resource consumption. The notion of resource is generic and it can refer to time, number of executed instructions, memory usage, calls to a specific method, etc. Resource analysis [58] aims at automatically bounding the resource consumption of executing programs statically, that is, without actually having to run them. The results of the analysis are then valid for any possible input data value. *Upper* bounds on the resource consumption provide guarantees that the program will never exceed the amount of resources the analysis infers. *Lower* bounds estimate the cost in the best case for all possible executions. They can be used to decide whether it is

worth to execute a task remotely, as the costs of requesting remote execution can actually be higher than just executing the task locally.

Research on cost analysis to-date has mainly focused on sequential programming languages. The COSTA system [1] is a state-of-the-art cost and termination analyser for sequential Java (bytecode) programs. A recent extension of COSTA, called COSTABS [2], extends the functionalities of COSTA to analyse Core ABS models.[3] The resulting extension is the first static cost analyser for a concurrent and distributed language. Clearly, cost is often affected by the fact that concurrent computations can be suspended and resumed along an execution. This is because the execution state can change when the processor is released (e.g., class fields can be modified) and this in turn can influence the associated cost (e.g., a loop can perform after the change a larger number of iterations). The techniques implemented in COSTABS for handling such concurrent behaviour of ABS models are described in detail in [2].

6.3.1 Cost models

The user of COSTABS must first select the *cost model* of interest. The cost model determines the type of resource, among the following ones:

- *Termination* it is the simplest cost model which simply ensures that the resource consumption is bounded, but does not provide a concrete bound.
- *Steps* it tries to approximate the number of executed instructions, including both instructions of the imperative and the functional part of the model.
- *Memory* the memory consumption estimates the size of the terms constructed in the functional part of the language. This is because objects are meant to be the concurrency units while the data structures are constructed using terms.
- *Objects* it counts the total number of objects created along the execution. This provides an indication of the amount of parallelism that might be achieved.
- *Task level* it estimates the number of tasks that are spawned along an execution. This can be useful for finding optimal deployment configurations.

6.3.2 Cost centres

The next option of COSTABS is whether the cost is split into *cost centres*. Cost centres represent the different distributed components of the system and allow us to obtain the cost per component rather than a single cost expression which accumulates the resource consumption of all components, as in traditional analysers for sequential languages. The current implementation of COSTABS assumes that objects of the same type belong to the same cost centre, that is, they share the processor. In future work, we plan to automatically infer to which component each object belongs and then assign the computational cost performed on each object to its corresponding cost centre.

There is a third option which allows the user to choose the size abstraction used in the analysis, which can either be *Term Size* or *Term Depth*. We do not go into the details of these two options here; interested readers can find out more in [2]. Once all options have been set up, we can proceed to analyse the selected method (or function). COSTABS analyses it as well as all code reachable from it.

As an example, let us show the results that COSTABS produces on a fragment of the Replication System case study. Recall that the `processFile` method of `ClientJobImpl` shown in Listing 3 on Page 572.

```
def Bool hasf(Dir d, Fn id) =
  case snd(d) { Entry(e) => isJust(ffind(e,id));};

def Maybe<Either<File,Dir>> ffind(Entry f,Fn id)
    =
 case contains(keys(f),id) {
  True => case lookup(f,id) {
   Cnt(s) => mvalue(True,id,Cnt(s));
   Entry(e) => mvalue(False,id,Entry(e)); };
  False => case f {
   InsertAssoc(Pair(i,Cnt(_)),fm) =>
       ffind(fm,id);
   InsertAssoc(Pair(i,Entry(g)),fm) =>
    case ffind(g,id) {
     Nothing => ffind(fm,id);
     r => qualify(r,i); };
   EmptyMap => Nothing; }; };
```

Listing 16 Implementation of function `hasf`

We want to analyse the method `hasFile` of `DataBase` that is asynchronously invoked by `processFile`. The implementation of the method `hasFile` is provided by the class `DataBaseImpl`. The method `hasFile` takes a file identifier `i` of type `Fn` and checks if there exists a file with identifier `i` in the database. The Replication System ABS model abstracts the physical implementation of a data base using ABS algebraic data types. Specifically, the method `hasFile` evaluates the expression `hasf(r,i)`, where `r` is a `Dir` value representing the top level directory of the client's file system. Listing 16 shows the definition of `hasf`.

After selecting the *Instructions* cost model, enabling the *Cost centres* option and using the *Term size* abstraction, COSTABS infers that the number of executed instructions is asymptotically quadratic on the size of the directory which is traversed, that is, $O(Dir^2)$. The upper bound is quadratic because in function `ffind` invoked from `hasf` (which is in turn invoked from method `hasFile`), we perform two

[3] The extension to Full ABS is currently under development.

recursive calls to `ffind` which, as expected, lead to a quadratic complexity.

Assuming that the option cost centre is enabled, as no computation is performed when executing the analysed method on objects of types `DeploymentComponent` or `main`, their cost is zero. In the cost centre for objects of type `ClientJobImp`, we have performed just 4 computation steps (which are due to the steps executed in method `hasFile` of `DataBase`). The remaining cost is performed in an object of type `DataBaseImpl`, as it appears at the bottom.

Our example has shown that after selecting a method for analysis, in order to approximate its cost, COSTABS analyses the method as well as all methods and functions invoked from it in a uniform and precise way.

6.3.3 Final remarks

The results obtained by COSTABS have been used to strengthen the results obtained by the Maude simulator [12]. The main idea here is to analyse the functional part of the model statically by using COSTABS. As the upper bounds obtained consider the worst-case cost of the execution, they strengthen the results obtained by the simulator which runs the concurrent part of the model for a particular input. Besides, we have also studied [4] the automatic generation of proofs by using the Key verification tool, which formally verifies that the results obtained by the COSTABS tool are correct (since the implementation could contain bugs).

The resource analysis currently is not able to efficiently handle variability. Each extension requires a whole re-analysis of the code. Incremental resource analysis [8] aims at re-analysing only the fragments of the code whose resource consumption is affected by the variation, instead of having to re-analyse the whole program. However, the extension of incremental resource analysis to the concurrent objects paradigm still has not been studied, and it is subject to future work.

Also, we are studying the use of the results gathered by a may-happen-in-parallel analysis [11] in order to improve the bounds obtained by COSTABS. In particular, if we know that two methods cannot be executing in parallel, we do not have to consider potential interleaving in their executions. This can greatly improve the quality of the bounds obtained by COSTABS since otherwise all potential interleaving (maybe useless) need to be considered by the analysis.

7 Discussion

In this section we discuss how the ABS language and the ABS tool suite fulfil the requirements of usability, reducing manual effort and supporting integrated environments. We also provide suggestions where improvements can be made.

Table 1 Metrics about case study

Metrics	Java	ABS
No. of lines of code	6,400	5,000
No. of classes	44	40
No. of interfaces	2	43
No. of user-defined functions	N/A	80
No. of user-defined data types	N/A	17
No. of features	N/A	15
No. of deltas	N/A	15
No. of products	N/A	96

7.1 ABS language

The case study described in this paper considered the Replication System, which is part of the Fredhopper Access Server (FAS). The current Java implementation of FAS has over 150,000 lines of code.

Table 1 shows some metrics about the existing implementation and the ABS model of the Replication System. In particular, with 15 features, the Replication System generates 96 products (if feature attributes are ignored). We see that the number of lines of the existing Java implementation and of the ABS model are not very different. This is because the ABS model describes additional model-level information such as deployment components and simulations of external inputs, which the Java implementation lacks. Also, the ABS model includes scheduling information, file systems, and data bases, while the Java implementation leverages third party libraries and the Java API. These additional model-level information accounts for over 1,000 lines of code.

On the other hand, some initial results from another case study (currently under review for publication) point to the effects of the variability constructs of ABS on existing systems. In that case study, an existing model written in Creol [40] is being re-written using ABS. The original system exhibits feature variability along multiple axes and uses a preprocessor to extract a desired product from common code in a somewhat ad-hoc fashion. Feature selection is implemented via setting pre-processor values and `#ifdef` constructs in the code. The corresponding ABS model (currently a work in progress, but mostly feature-complete) is about half the size of the old model, as measured in lines of code, and the code is subjectively easier to understand since different features are separated cleanly.

7.1.1 Variability modelling

We consider the usability aspect of the ABS language by considering its practical expressiveness in modelling problem space variability.

ABS offers μTVL for modelling variability at a design level. μTVL was used to specify the feature model of the Replication System. The language provides a compositional hierarchical view of the dependencies between features in terms of sub-feature relationships, optional and mandatory selections, and constraints between features and attribute specifications of features. We found that μTVL provides the necessary expressiveness to model the design space variability of the Replication System. Using the ABS product selection language, we could precisely express the feature selection for each relevant product. In comparison, the existing Java implementation does not have an explicit, well-defined feature model that captures the relationships and constraints between system features. Features are informally described in textual documents or declared through Java preference files. Unlike the ABS model, features are not modelled explicitly and compositionally. As a result, we had to manually harvest parts of the functionalities of the Replication System and translate these functionalities into features.

At the object behaviour level, variability modeling in ABS relies on delta modules, as explained in Sect. 4.2. The unit of variability at that level a method, that is, it is not possible to selectively modify parts of a method. In practice, this level was not found to be too restrictive if the code followed good object-oriented design criteria. The need to change only a part of a method usually indicated that a method had more than one purpose or mixed different abstraction levels. Applying the standard extract method refactoring technique led to both cleaner code and to a delta that could be cleanly applied.

7.1.2 Behavioural modelling

We consider usability aspects of the ABS language by considering its practical expressiveness in modelling solution space variability:

Expressiveness The Delta modelling language offers the expressiveness to specify variability at the level of object behaviour. Together with product line configuration, product selection and feature modelling facilities, the ABS language offers a holistic approach to expressing variable aspects of a program as features and relating them to object behaviour.

Configuration We were able to use ABS to incrementally and compositionally develop the Replication System that yields members that are well-typed and valid with respect to the system's variability. We were able to systematically and in a top-down fashion implement all features in the feature model to obtain the Replication System. The existing Java implementation provides no explicit relationships between features. Configuration and selection of features are defined in terms of Java preference data. Moreover, preference data only has an explicit connection to qualified Java packages and class names, and not to object behaviour. As a result it is difficult to ensure that all combination of features are considered every time a change has to be made to the implementation.

Modularity The ABS module system allowed us to model both the commonality and the variability of the Replication System separately and incrementally. As future work we aim to perform compositional analysis on delta definitions.

Reusability The delta modelling language provides a mechanism to express variability at the level of behaviour. Together with functional and object composition, the ABS language thus provides a wide range of mechanisms for code reuse. In particular, the combination of object composition and delta modelling allows us to achieve code reusability similar to that of class inheritance. In addition, the ABS module system also allows more generic definitions such as data types and functions to be reused across the ABS model of the product line. The existing Java implementation achieves code reuse via class inheritance and object compositions.

Testability Using the delta modelling language and the ABSUnit framework, we were able to define unit tests in ABS. We were able to systematically modify existing methods of the SUT in a type safe manner, and add setter methods to obtain the initial state that satisfy the precondition of the method to be tested. Moreover, as the objects in ABS can only be typed by their interfaces, we were also able to construct mock implementations as inputs for test cases. In the existing Java implementation, neither of these are possible in general. It is not possible to modify final or private methods in a type safe manner[4]. Also objects may be typed by their classes and this makes mock object construction complex and some time impossible.

7.2 ABS front-end

7.2.1 Type checking

We have found that on-the-fly type checking provided by the combination of the ABS type checker and the ABS eclipse plugin removes the need to manually determine type correctness of our ABS model. This reduces manual effort considerably.

7.2.2 Location type checking

We use the ABS location type checker through the Eclipse plugin to perform on-the-fly type checking if our model erroneously defines synchronous method invocations between

[4] Unsafe modification of final or private methods in Java can be achieved using the Java Reflection API.

objects residing in separate concurrent object groups. This reduces manual effort considerably.

7.2.3 Product selection

We have found that the product selection allows us to generate products from the Replication System through the Eclipse plugin run configurations. Product selection reduces manual effort as it supports automatic application of delta modules to ABS models. The ABS Eclipse plugin conveniently provides the generation facility at the same level as the run configuration of Eclipse Java Development Tool. Moreover, the current ABS front-end provides information about the sequence of deltas applied during product selections, which helps debugging and other analyses.

7.2.4 Package dependency

We have found that the combination of Apache Maven, ABS package system and tight Eclipse integrations eases the code management of the Replication System ABS model as the size of the model increases. Moreover, it enables us to pull other ABS packages, such as those for ABS-FFI and ABS-Unit, so that they can be used in the case study. It also enables us to conduct our case study collaboratively.

7.2.5 Test runner generation

We have found the automatic test runner generation facility indispensable for quickly simulating concurrent unit tests. These are specified using ABSUnit in the same manner as with other unit testing frameworks for sequential testing, such as JUnit. Without the test runner generator, we would have to write a test runner for every combination of unit test cases we wanted to simulate.

7.3 Back-ends

7.3.1 Maude back-end

The Maude back-end is very useful for debugging runtime error such as deadlocking. This reduces manual effort considerably as it allows us to investigate the complete terminating state of a Maude simulation. This would be extremely difficult for the existing Java implementation as there is no simulation support and the distributed setting in a real-time execution makes it hard to access the complete terminating state.

7.3.2 Java back-end

The Java back-end generates Java source code from ABS models that can be executed in real time against physical platforms. We have found that the Java back-end provides the mechanism to specify choices in the presence of non-determinism during the execution of the generated Java code using the Eclipse debugger. This is useful because we are able to explore many more execution paths, which help to reveal unforeseen errors in the models.

7.3.3 Foreign function interface

We have used the combination of the Java back-end and the ABS-FFI to simulate the Replication System in a distributed setting. The existing implementation of the Replication System is deployed according to FAS deployment architecture, and communication between SyncClient and SyncServer is via IP sockets. By using ABS-FFI we need not implement I/O and IP sockets directly at the level of ABS, but rather use the existing support provided by Java.

7.4 Eclipse plugin

We have found that the ABS Eclipse plugin dramatically increases the usability of the tools provided by the ABS tool suite for modelling. We were able to leverage heavily on the syntax highlighting, content completion and code navigation provided by the ABS editing perspective. We were also able to modularly organise our ABS model into separate ABS artifacts and use the Maven support for ABS provided by the ABS Eclipse plugin to connect these artifacts. This allowed us to achieve a much better separation of concerns within the ABS model. We believe the combination of this Eclipse plugin and Maven support increases ABS's applicability in the industry where collaborative software development is prevalent and third party libraries are heavily used.

7.5 Debugging and visualisation

We have used the ABS debugging perspective provided by the ABS Eclipse plugin to step through Java execution of the Replication System ABS model. We have found it particularly useful when debugging failed unit tests defined using the ABSUnit framework. Unlike the Maude simulation, using the debugging perspective we were able to pinpoint the cause of failed tests much quicker. Using the debugger, we were also able to record and replay the failed test run to ensure that the regression is fixed.

7.6 Resource analysis

We have found that resource analyses using COSTABS gave us a better idea about the complexity of our model. For example, the method `hasFile` analysed in Sect. 6.3 is a method which is invoked for every file entry of a replication item.

This is a hot spot and has a quadratic complexity. These results will allow us to (1) optimise our ABS model in continuation of the case study, (2) monitor changes to resource bounds during the evolution of the ABS model (changes over time), and (3) utilise the upper bound information to specify resource usage when investigating the Replication System's underlying platform variability such as those introduced in Sect. 4.5.

7.7 Summary

In this section we have provided a discussion on how the ABS language and the ABS tool suite satisfy the requirements of usability, manual effort reduction and integrated environments support.

8 Related work

The idea of using formal models, written down in formal modelling languages, to describe software systems, goes back a long time. One of the earliest examples of such a formal language is VDM [22], which was originally (then under the name VDL) developed in order to specify the semantics of the IBM PL/I language and compiler [18]. VDM proved to be useful beyond its original intended field and is still in active use [44] and being extended with object-oriented and real-time features [56]. Other formal modelling languages in a similar, state-based style as ABS are Z [54] and B [6], respectively, their object-oriented extensions.

Due to language features like asynchronous method calls and COGs, ABS has a stronger focus on modelling distributed and parallel systems than the above languages. VDM++, for example, implements a conventional threading model where more than one process can execute within an object at a time, relying on mutex annotations on atomic operations to manually prevent race conditions. VDM-RT [36] adds static deployment scenarios and asynchronous calls, albeit without return values. ABS processes, in contrast, never leave the scope of their object, create new processes upon procedure calls, and synchronise with each other via explicit synchronisation points. Hence, ABS methods are not necessarily executed atomically, but processes cooperate within an object in a transparent way that is obvious from the program source. Seen from this perspective, the semantics of ABS are nearer to the Actor model [33] and to process calculi like CSP [35] and the Pi-calculus [45] (the asynchronous messages of ABS can be implemented via processes in these calculi) than to languages derived from a sequential programming model. Due to this simpler concurrency model, proof theories for ABS (and its precursor Creol [40], which shares the same concurrency model) exist [29,31] and have been prototypically implemented [9,10] to allow verification of multi-threaded ABS programs.

Although ABS is not currently used as a language for application-level programming, Erlang [16] (itself inspired by the Actor model) has shown that distributed systems consisting of large numbers of active, message-passing entities can be used at scale. ABS objects conceptually have the same role as Erlang processes, once the well-known problems of distributed systems [60] can be addressed in ABS—work towards a distributed exception handling and recovery system is outlined in [39].

Most existing approaches to express variability in modelling and implementation languages can be classified into two main categories [42,55]: annotative and compositional.

Annotative approaches consider one model representing all products of the product line, implementing features as some form of annotations in the source code. Variant annotations, e.g., using UML stereotypes in UML models [32] or presence conditions [27], define which parts of the model have to be removed to derive a concrete product model. The orthogonal variability model (OVM) proposed in Pohl et al. [48] models the variability of product line artifacts in a separate model where links to the artifact model take the place of annotations. Similarly, decision maps in KobrA [5] define which parts of the product artifacts have to be modified for certain products. The architecture description language EAST-ADL2 presents a compositional approach [50], aligning features closely with the software architecture of the product line (but the need to break modularity in the feature model is acknowledged in the paper).

Compositional approaches, such as delta modelling [51], associate distinct model fragments or modules with product features. These distinct entities are composed for a particular feature configuration. A prominent example of this approach is AHEAD [23], which can be applied on the design as well as on the implementation level. In AHEAD, a product is built by stepwise refinement of a base module with a sequence of feature modules. Design-level models can also be constructed using Aspect-oriented programming techniques [37,47,55].

Delta modelling has similarities with Aspect-oriented programming as a concept of organising the various features and concerns of a software system. Aspects provide a finer-grained mechanism to arbitrarily insert code into existing methods than delta modeling, whereas deltas support a more extensive but coarser-grained mechanism, where classes and methods can be added, modified and even removed. Moreover, the feature and delta modelling facilities of ABS provide a mechanism to model software product lines following established principles. A more detailed comparison of the usage of aspects and feature models has been presented by Apel and Batory [3].

In feature-oriented software development (FOSD) [23], features are considered on the linguistic level by feature

modules. Apart from Jak [23], there are various other languages using the feature-oriented paradigm, such as FeatureC++ [15], FeatureFST [13], or Prehofer's feature-oriented Java extension [49]. In [13,46], combinations of feature modules and aspects are considered. In [14], an algebraic representation of FOSD is presented. Feature Alloy [17] instantiates feature-oriented concepts for the formal specification language Alloy.

9 Conclusion

This paper presented the ABS language and its tool suite for modelling distributed, adaptable, object-oriented software systems. ABS is a statically typed, multi-paradigm language with functional and object-oriented features. Its concurrency model is based on concurrent object groups (COGs) which form the unit of distribution in ABS. ABS has built-in support for software product line engineering. Delta modules allow the modeller to express variability on the ABS level. The feature model of a product line is described using the μTVL language. Product line configuration and product selection then connect the feature model to the delta model to gain a fully automatic way for generating ABS products from feature models.

The ABS language is supported by the ABS tool suite that offers a comprehensive set of tools for developing, executing, testing, debugging, analysing and visualising ABS models. This includes a standard compiler front-end for parsing and type-checking ABS models. This front-end is integrated into an Eclipse plugin that supports standard features like syntax highlighting, error reporting, outline views, auto-completion and a debugging perspective. In addition, the plugin supports the execution of other ABS tools directly from the IDE. There is no direct interpreter for the ABS language; instead ABS models are translated to other languages like Java, Scala and Maude. Unit testing in ABS is supported by a unit testing framework based on annotations. ABS can interact with software modules written in other languages (currently only Java) by using the ABS-FFI of ABS (ABS-FFI). ABS models can be packed into ABS packages for managing module dependencies by using Maven. This is especially interesting in the context of software product lines where dependencies between different product artifacts can be automatically managed. Product line engineering is further supported by tools for automatic consistency checking of feature models written in μTVL and product generation from product line configurations and product selections. The ABS language and its tools are being developed in tandem with several case studies of which the largest one, the Replication System, is presented in this paper. The language and the tools are thus constantly evaluated and improved when needed.

Acknowledgments We gratefully thank the anonymous referees for many useful comments and suggestions that greatly helped to improve this article. This work was funded in part by the Information and Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 and PRI-AIBDE-2011-0900 projects, by UCM-BSCH-GR35/10-A-910502 grant and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

Appendix A

Glossary

μTVL See Micro textual variability language.

ABS Abstract Behavioral Specification language. An executable class-based, concurrent, object-oriented modeling language based on Creol, created for the HATS project.

ABS-FFI See ABS foreign function interface.

ABS foreign function interface A set of small extensions to the ABS language to allows ABS models to interact with programs written in other existing programming languages such as Java.

ABS package The ABS package, based on the Java Archive (Jar) format, is an open source, platform-independent file format. APK aggregates many (related) ABS modules into a single unit and represents a single point of reference to that unit.

ABSUnit A unit testing framework for defining, managing and executing unit tests for ABS models.

APK See ABS package.

Application condition A logical expression over features and feature attributes, that evaluates to a Boolean value with respect to a given feature selection.

COG See Concurrent object group.

Concurrent object group A unit of concurrency in ABS. Consists of one or more objects. Within a cog, only one process can be active at any one time.

Core A set of classes to which a delta is applied, resulting in a new core.

Core ABS The behavioural functional and object-oriented core of the ABS modeling language

Cost analysis Static analysis techniques which over approximates the cost of executing a program for any value of its input data.

Cost center A cost center represents a distributed component to which its cost will be assigned.

Cost model A cost model defines the type of resource of interest (e.g., time, steps, memory).

COSTABS A cost and termination analyser for ABS programs.

Delta A specification of modifications to core ABS classes and interfaces.

Delta module see Delta.

Deployment The act of running a real system on a machine or group of machines.

Deployment component A model of a location able to execute (parts) of a system. Deployment components can be modelled with meta data describing the amount of resources (e.g., CPU, memory) available to COGs executing on a specific deployment component.

Deployment modelling The act of modelling non-functional properties of a system, specifically the location of parts of a system and the resources available at these locations.

FAS See Fredhopper access server

Fredhopper access server Fredhopper access server is a component-based, service-oriented and server-based software system, which provides search and merchandising IT services to e-Commerce companies such as large catalog traders, travel booking, managers of classified, etc.

Feature A characteristic of a software system that is relevant to some stakeholder in the development project

Feature attribute A way to characterise a feature more precisely. A feature attribute has a type, which defines the set of values it can assume

Feature model A model that specifies the feature combination that are valid

Feature selection A set of features

Full ABS Core ABS language plus extensions supporting SPL engineering

Future See Future variable.

Future variable A handle on an ABS process, also the value of an asynchronous method call expression. Futures are used to synchronise with and get the result from a process. Futures are first-class values and can be passed as parameters.

Live environment A live environment in the FAS deployment architecture is responsible for processing queries from client web applications via the Web Services technology.

Lower bound Under-approximation of the best case cost

Maven An open source Java-based tool that consists of a project object model (POM), a set of standards, a project lifecycle and extensible dependency management, and build systems via plug-ins.

Maude A rewriting logic interpreter and toolkit that is used to specify the semantics of ABS and to simulate execution of ABS models.

Micro textual variability language The HATS Variability Modeling Language that expresses variability on the level of feature models.

Modifier A fragment of a delta that specifies how a class, method, or field should be modified

Module A language construct in ABS that introduces a name space. Modules define the scope of unqualified identifiers.

Process A unit of execution in ABS. Processes are created in response to asynchronous method calls and run in the scope of one object.

Product A single software system

Product selection A set of products that is of particular interest to the developer

Replication system The Replication System in the FAS deployment architecture synchronises the configurations and data from the staging environment to multiple live environments. Specifically the Replication System consists of the synchronisation server (SyncServer) and one or more clients (SyncClient).

Resource analysis See Cost analysis

Scheduling point A location in the source code where a COG is allowed to suspend the running process and choose another enabled process to run. Scheduling points can be unconditional (via the suspend statement) or conditional (via the await statement).

SPL See Software product line

SPL configuration A set of directives associating features with delta modules

Software product line A set of software systems that share some commonality, but also differ in some well-defined ways. The systems of an SPL are developed together from the same code base.

Staging environment A staging environment in the FAS deployment architecture is responsible for receiving client data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments using the Replication System. See Replication System.

Upper bound Over-approximation of the worst case cost

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.n, Zanardini, D.: COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In: Proceedings of FMCO'07, vol. 5382 LNCS, pp. 113–133. Springer, Berlin (2008)
2. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Cost analysis of concurrent OO programs. In: The 9th Asian Symposium on Programming Languages and Systems (APLAS'11). Springer, Berlin, December (2011) (to appear)
3. Apel, S., Batory, D.: When to use features and aspects? A case study. In: International Conference on Generative Programming and Component Engineering, GPCE '06, pp. 59–68. ACM Press (2006)
4. Albert, E., Bubel, R., Genaim, S., Hähnle, R., Puebla, G., Román-Díez, G.: Verified resource guarantees using COSTA and KeY. In: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'11). ACM Press (2011)
5. Atkinson, C., Bayer, J., Muthig, D.: Component-based product line development: The KobrA approach. In: SPLC (2000)

6. Abrial, J.-R.: The B-Book—Assigning programs to meanings. Cambridge University Press, Cambridge (2005)
7. The ABS language specification. (2011) http://tools.hats-project.eu/download/absrefmanual.pdf
8. Albert, E., Correas, J., Puebla, G., Román-Díez, G.: Incremental resource usage analysis. In: ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2012), pp. 25–34. ACM Press, January 2012
9. Ahrendt, W., Dylla, M.: A verification system for distributed objects with asynchronous method calls. In: Breitman, K., Cavalcanti, A. (eds.) Formal methods and software engineering, international conference on formal engineering methods (ICFEM'09), vol. 5885, Lecture Notes in Computer Science, pp. 387–406. Springer, Berlin (2009)
10. Ahrendt, W., Dylla, M.: A system for compositional verification of asynchronous objects. Sci. Comput. Program. (2011)
11. Albert, E., Flores-Montoya, A., Genaim, S.: Analysis of May-Happen-in-parallel in concurrent objects. In: 14th IFIP WG 6.1 International Conference FMOODS 2012 and 32nd IFIP WG 6.1 International Conference FORTE 2012, Stockholm, Sweden, pp. 13–16 June, 2012, Proceedings, IFIP-LNCS. Springer, June 2012
12. Albert, E., Genaim, S., Gómez-Zamalloa, S., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Simulating concurrent behaviors with worst-case cost bounds. In: Butler, M., Schulte, W. (eds.) Proceedings of 17th International Symposium on Formal Methods (FM 2011), vol. 6664, Lecture Notes in Computer Science, pp. 353–368. Springer, Berlin (2011)
13. Apel, S., Lengauer, C.: Superimposition: A language-independent approach to software composition. In: Software Composition, vol. 4954, Lecture Notes in Computer Science, pp. 20–35. Springer, Berlin (2008)
14. Apel, S., Lengauer, C., Möller, B., Kästner, C.: An algebraic foundation for automatic feature-based program synthesis. Sci. Comput. Program. (SCP) **75**(11), 1022–1047 (2010)
15. Apel, S., Leich, T., Rosenmüller, M., Saake, G.: FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In: GPCE, vol. 3676, Lecture Notes in Computer Science, pp. 125–140. Springer, Berlin (2005)
16. Armstrong, J.: Erlang. Commun. ACM, **53**(9), 68–75 (2010)
17. Apel, S., Scholz, W., Lengauer, C., Kästner, C.: Detecting Dependences and interactions in feature-oriented design. In: IEEE International Symposium on Software Reliability Engineering (ISSRE) (2010)
18. Bekic, H., Bjørner, D., Henhapl, W., Jones, C.B., Lucas, P. : On the formal definition of a PL/I subset (selected parts). In: Jones, C.B. (ed.) Programming Languages and Their Definition, vol. 177, Lecture Notes in Computer Science, pp. 107–155. Springer, Berlin (1984)
19. Batory, D., Benavides, D., Ruiz-Cortes, A.: Automated analysis of feature models: challenges ahead. Commun. ACM **49**(12), 45–47 (2006)
20. 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
21. Beck, K.: Simple smalltalk testing: with patterns. Smalltalk Report, 4(3), October 1994
22. Bjørner, D., Jones, C.B. (eds.) The Vienna Development Method: The Meta- Language, vol. 61, Lecture Notes in Computer Science, Springer, Berlin (1978)
23. Batory, D.S., Sarvela, J.N., Rauschmayer, Axel.: Scaling step-wise refinement. IEEE Trans. Softw. Eng. 30(6) (2004)
24. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude—A high-performance logical framework, how to specify, program and verify systems in rewriting logic, vol. 4350, Lecture Notes in Computer Science. Springer, Berlin (2007)
25. Clarke, D., Diakov, N., Hähnle, R., Johnsen, E.B., Puebla, G., Weitzel, B., Wong P.Y.H.: HATS: A Formal software product line engineering methodology. In: Proceedings of International Workshop on Formal Methods in Software Product Line Engineering, September 2010
26. Clarke, D., Diakov, N., Hähnle, R., Johnsen, E.B., Schaefer, I., Schäfer, J., Schlatte, R., Wong, P.Y.H.: Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In: Bernardo, M., Issarny, V. (eds.) Formal methods for eternal networked software systems, vol. 6659, Lecture Notes in Computer Science, pp. 417–457. Springer, Berlin (2011)
27. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Softw. Process Improv. Pract. **10**(1), 7–29 (2005)
28. Clarke, D., Muschevici, R., Proença, J., Schaefer, I., Schlatte, R.: Variability modelling in the ABS language. In: Formal Methods for Components and Objects, vol. 6957 of LNCS. Springer, Berlin (2011)
29. Din, C.C., Dovland, J., Johnsen, E.B., Owe, O.: Observable behavior of distributed systems: Component reasoning for concurrent objects. J. Logic Algebr. Program. (2012) (to appear)
30. Requirement Elicitation, August (2009) Deliverable 5.1 of project FP7-231620 (HATS). http://www.hats-project.eu
31. Dovland, J., Johnsen, E.B., Owe, O. : Observable behavior of dynamic systems: Component reasoning for concurrent objects. In: Goldin, D., Arbab, F. (eds.) Proceedings of Workshop on the Foundations of Interactive Computation (FInCo'07) vol. 203 of Electronic Notes in Theoretical Computer Science., pp. 19–34. Elsevier, Amsterdam (2008)
32. Gomaa, Hassan: Designing Software Product Lines with UML. Addison Wesley, Boston (2004)
33. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: IJCAI, pp. 235–245 (1973)
34. Helvensteijn, M., Muschevici, R., Wong, P.Y.H. : Delta modeling in practice: a Fredhopper case study. In: Eisenecker, U.W., Apel, S., Gnesi, S. (eds.) International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)., pp. 139–148. ACM Press, New York (2012)
35. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
36. Hooman, J., Verhoef, M. : Formal semantics of a vdm extension for distributed embedded systems. In: Dams, D., Hannemann, U., Steffen, M. (eds.) Concurrency, Compositionality, and Correctness, vol. 5930, Lecture Notes in Computer Science, pp. 142–161. Springer, BErlin (2010)
37. Heidenreich, F., Wende, C.: Bridging the gap between features and models. In: Aspect-Oriented Product Line Engineering (AOPLE'07) (2007)
38. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig B., de Boer F.S., Bonsangue M.M. (eds.) Proceedings of 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010), vol. 6957 of LNCS, pp. 142–164. Springer, Berlin (2011)
39. Johnsen, E. B., Lanese, I., Zavattaro, G.: Fault in the future. In: Meuter W.D., Roman G.-C. (eds.) Proceedings of 13th International Conference on Coordination Models and Languages (COORDINATION 2011), vol. 6721, Lecture Notes in Computer Science, pp. 1–15. Springer, Berlin (2011)
40. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Softw. Syst. Model. **6**(1), 35–58 (2007)
41. Johnsen, E.B., Owe, O., Schlatte, R., Tapia Tarifa, S.L.: Validating timed models of deployment components with parametric

concurrency. In: Beckert, B., Marché, C. (eds.) Proceedings of International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'10), vol. 6528 Lecture Notes in Computer Science, pp. 46–60. Springer, Berlin (2011)
42. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: ICSE pp. 311–320 (2008)
43. Kang, K.C., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon University Software Engineering Institute (1990)
44. Larsen, P.G., Battle, N., Ferreira, M.A., Fitzgerald, J.S., Lausdahl, K., Verhoef, M.: The Overture initiative integrating tools for VDM. ACM SIGSOFT Softw. Eng. Notes. **35**(1), 1–6 (2010)
45. Milner, R.: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press, Cambridge (1999)
46. Mezini, M., Ostermann, K.: Variability management with feature-oriented programming and aspects. In: SIGSOFT FSE, pp. 127–136. ACM, New York (2004)
47. Noda, N., Kishi, T.: Aspect-Oriented Modeling for Variability Management. In: SPLC (2008)
48. Pohl, K., Böckle, G., Van Der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Berlin (2005)
49. Prehofer, C.: Feature-oriented programming: A fresh look at objects. In: European Conference on Object-Oriented Programming (ECOOP'97), vol. 1241, Lecture Notes in Computer Science, pp. 419–443. Springer, Berlin (1997)
50. Reiser, M.-O., Kolagari, R.T., Weber, M.: Compositional variability—concepts and patterns. In: HICSS, pp. 1–10. IEEE Computer Society (2009)
51. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Proceedings of 14th Software Product Line Conference (SPLC 2010), September 2010
52. Schaefer, I., Bettini, L., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Proceedings of the 14th International Conference on Software Product Lines: Going Beyond, SPLC'10, pp. 77–91. Springer, Berlin (2010)
53. Schaefer, I., Hähnle, R.: Formal methods in software product line engineering. IEEE Comput. **44**(2), 82–85 (2011)
54. Spivey, J.M.: The Z notation—a reference manual. Prentice Hall, Englewood Cliffs (1989)
55. Völter, M., Groher, I.: Product line implementation using aspect-oriented and model-driven software development. In: SPLC, pp. 233–242 (2007)
56. Verhoef, M., Larsen, P.G., Hooman, J. : Modeling and validating distributed embedded real-time systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM, vol. 4085, Lecture Notes in Computer Science, pp. 147–162. Springer, Berlin (2006)
57. Wong, P.Y.H., Diakov, N., Schaefer, I.: Modelling distributed adaptable object-oriented systems using hats approach: A fredhopper case study. In: Beckert, B., Damiani, F., Gurov, D. (eds.) 2nd International Conference on Formal Verification of Object-Oriented Software, vol. 7421 of LNCS. Springer, Berlin (2012)
58. Wegbreit, B.: Mechanical program analysis. Commun. ACM **18**(9) (1975)
59. Welsch, Y., Schäfer, J.: Location types for safe distributed object-oriented programming. In: 49th International Conference on Objects, Models, Components and Patterns (TOOLS Europe 2011), LNCS, pp. 194–210. Springer, June 2011
60. Waldo, J., Wyant, G., Wollrath, A., Kendall, S.C.: A note on distributed computing. In: Vitek, J., Tschudin, C.F. (eds.) Mobile Object Systems, vol. 1222, Lecture Notes in Computer Science, pp. 49–64. Springer, Berlin (1996)