Modelling Adaptable Distributed Object Oriented Systems using the HATS Approach – A Fredhopper Case Study [*]

Peter Y. H. Wong[1], Nikolay Diakov[1], and Ina Schaefer[2]

[1] Fredhopper B.V., Amsterdam, The Netherland
{peter.wong,nikolay.diakov}@fredhopper.com
[2] Technische Universität Braunschweig, Germany
i.schaefer@tu-bs.de

Abstract. The HATS project aims at developing a model-centric engineering methodology for the design, implementation and verification of distributed, concurrent and highly configurable systems. Such systems also have high demands on their dependability and trustworthiness. The HATS approach is centered around the Abstract Behavioural Specification modelling language (ABS) and its accompanying tools suite. The HATS approach allows precisely specifying and analyzing the abstract behaviour of distributed software systems and their variability. The HATS project measures its success by applying its framework not only to toy examples, but to real industrial scenarios. In this paper, we evaluate the HATS approach for modelling an industrial scale case study provided by the eCommerce company Fredhopper. In this case study we consider Fredhopper Access Server (FAS). We model the commonality and variability of FAS's replication system using the ABS language and provide an evaluation based on our experience.

Keywords: Variability modelling; Software product lines; Industrial case study; Formal modelling and specification; Evaluation

1 Introduction

Software systems evolve to meet changing requirements over time. Evolving software systems may require substantial changes to the software and often result 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 "Highly Adaptable and Trustworthy Software using Formal Models" (HATS) project aims at developing tools, techniques and a formal software product line (SPL) development methodology [10, 34] for rigorously engineering distributed software systems are subject to changes.

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

The HATS approach is centered around the Abstract Behavioural Specification (ABS) modelling language [23, 13], an accompanying ABS tool suite [11, 36] and a formal engineering methodology for developing SPL [10]. ABS facilitates to model precisely SPLs of distributed concurrent systems, focusing on their functionality, while providing the abstraction to express concerns, such as available resources, deployment scenarios and scheduling policies. In particular, the language of ABS provides modelling concepts for specifying SPL's variability from the level of feature models down to object behaviour. This permits large scale reuse within SPLs and rapid product construction during the application engineering phase of the SPL engineering methodolody [10].

In this paper, we evaluate the application of the HATS approach to an industrial SPL case study of the Fredhopper Access Server (FAS) product line. FAS, developed by Fredhopper B.V. (`www.fredhopper.com`), is a distributed service-oriented software system for Internet search and merchandising. In particular we consider FAS's *replication system*; the replication system ensures data consistency across the FAS deployment. We use this case study to evaluate the HATS approach with respect to the following criteria, derived during the requirement elicitation activity conducted at the beginning of the HATS project [17]:

Expressiveness We evaluate the ABS language with respect to its *practical* language expressiveness. We investigate from the user's perspective how readily and concisely ABS allows users to express program structures and behavior, and its capability to capture variability in SPLs.

Scalability We evaluate the ABS language with respect to the size and the complexity of the modelled system. It is important to provide mechanisms at the language level that permit separations of concerns, reuse and compositional development of SPLs.

Usability We evaluate the HATS approach with respect to its overall usability, focussing on the ease of adoption and learnability. We take into account the tool support, as well as the language's syntax and semantics.

The structure of this paper is as follows: Section 2 briefly presents the HATS approach; Section 3 describes the functionality of the Fredhopper Access Server (FAS) product line and its replication system; Section 4 considers how to model the replication system's commonality using ABS; Section 5 considers how to model the replication system's variability using ABS. We present an evaluation based on our experience using ABS in Section 6. We provide an overview of the existing approaches to model and analyse concurrent distributed systems with variabilities in Section 7 and a summary of this paper in Section 8.

2 HATS approach

The HATS approach is designed as a formal methodology for developing SPL [10]. The HATS methodology is a combination of the ABS language, a set of well-defined techniques and tool suite for ABS, and a formal methodology to bind them to specific steps in a SPL development process. ABS comprises a core language with specialised language extensions, each focusing on a particular aspect of modelling SPLs, while respecting the separation of concerns principle and encouraging reuse.

The *Core ABS* is a strongly typed, concurrent, object-based modelling language with a formal executable semantics and a type system [23]. The Core ABS consists of a functional and a concurrent object levels: The functional level provides a flexible way to model internal data in concurrent objects, while separating the concerns of internal computation from the model; this is an important language feature for scalability. The functional level supports user-defined parametric data types and functions with pattern matching. The concurrent object level is used to capture concurrent control flow and communication in ABS models; the concurrency model of the Core ABS is based on the concept of Concurrent Object Groups. A typical ABS model consists of multiple object groups at runtime. These groups can be regarded as autonomous, runtime components that are executed concurrently, share no state and communicate asynchronously. The core ABS's object-based model structure provides a good fit with UML modelling approaches, while its type system guarantee type safety at runtime for well typed core ABS models. The core ABS's executable semantics supports early verification and validation.

The ABS then extends the Core ABS with the following specialised extensions [13].

- The *Micro Textual Variability Language* (μTVL), based on Classen et al.'s TVL [14], expresses the variability of SPL at the level of feature models during the family engineering phase of the SPL engineering process.
- The *Delta Modeling Language* (DML), based on delta modelling [33], models variability of SPL at the level of object behavior during the family engineering phase of the SPL engineering process. The variability at the behavioral level is represented by a set of delta modules that contain modifications of the ABS model of software artifacts in SPL, such as additions, modifications and removals of model entities. Delta modules provides the capability to define generic software artifacts in SPL.
- The *Product Line Configuration Language* (CL) connects the variability of SPL from feature models down to object behavior by specifying the relationship between features and delta modules. Each delta module is associated with one or more feature in the feature model, thereby allowing reuse delta modules across features in the SPL.
- The *Product Selection Language* (PSL) specifies individual products in the SPL by providing a particular feature selection along with its initialization code.

The ABS tools suite [36] includes an ABS compiler front end, which takes a complete ABS model of the SPL as input, checks the model for syntax and semantic errors and translates it into an internal representation. The front end supports automatic product generation, variability of the SPL can be resolved by applying the corresponding sequence of delta modules to its core ABS model at compile time; variability resolution is one of the core activities during the

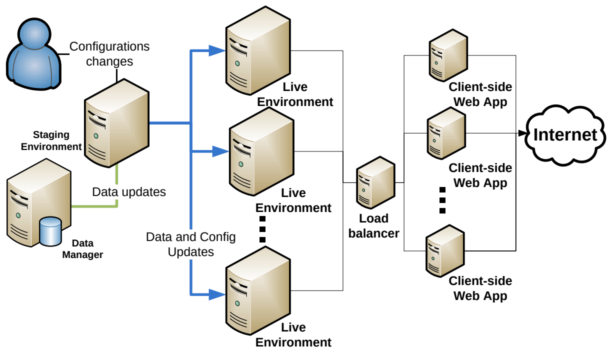

Fig. 1. An example of a FAS deployment

application engineering phase of the SPL. Different back ends translate the internal representation into Maude or Java, allowing ABS models to be executed and analyzed. The tools suite also includes a plug-in for the Eclipse IDE (`www.eclipse.org`). The plugin provides an Eclipse perspective for navigating, editing, visualizing, and type checking ABS models, and an integration with the back ends, so that ABS models can be executed or simulated directly from the IDE.

3 Fredhopper Access Server

The Fredhopper Access Server (FAS) is a component-based and service-oriented distributed software system. It provides search and merchandising services to e-Commerce companies such as large catalogue traders, travel agencies, etc. Each FAS installation is deployed to a customer according to the FAS deployment architecture. Figure 1 shows an example setup. A detailed presentation of FAS's individual components and its deployment model can be found in the HATS project report [18].

A FAS deployment consists of a set of live and staging environments. A live environment processes queries from client web applications via web services. FAS aims at providing a constant query capacity to client-side web applications. A staging environment is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments according to the replication protocol.

Implementations of the replication protocol are provided by the *replication system*. A replication system consists of a set of computation nodes; one of which is the synchronization server residing in a staging environment, while all other nodes are synchronization clients residing in the live environments. The synchronization server takes care of determining the schedule of replication, as well as the content of each replication item. The synchronization client is responsible for receiving data and configuration updates. A replication item is a set of files and represents a single unit of replicable data.

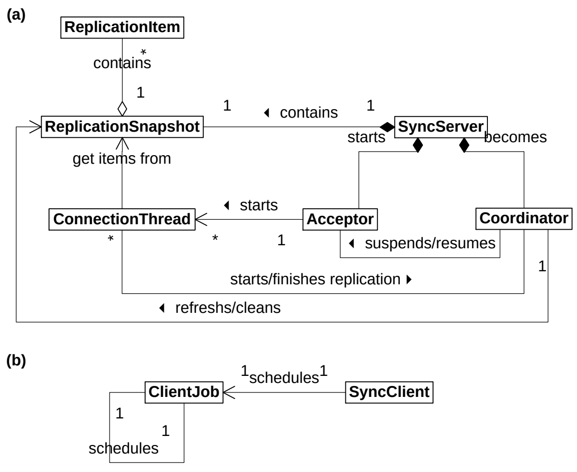

Fig. 2. Class diagram of (a) synchronisation server and (b) synchronisation client

The synchronization server communicate to clients via connection threads that serve as the interface to the server-side of the replication protocol. On the other hand, synchronization clients schedule client jobs to handle communications to the client-side of the replication protocol. In our ABS model, both connection threads and client jobs belong to separate concurrent object groups [23] (a mechanism to structure the object heap into separate units) and communicate via asynchronous method invocations. Cooperative multitasking and strict data encapsulation between the concurrent object groups prevent deadlocks and race conditions.

As part of the FAS product line, the replication system defines variability on the types of replication items, the coordination policy of replication and the resource consumption during replication. This allows members of the product line to be tailored for FAS deployments with specific data requirement and platform resource constraints.

4 Modelling Commonality

In this section, we present how to model the replication system's commonality using the Core ABS. Figures 2(a) and (b) show the UML class diagram of the synchronization server and client respectively. The synchronization server consists of an acceptor, several connection threads, a coordinator, a SyncServer and a replication snapshot. The synchronization client consists of a SyncClient and one or more client jobs. Listing 1.1 shows the ABS interfaces for the core components of the synchronization server and clients. For brevity, we have omitted ABS class definitions.

The Acceptor component is responsible for accepting connections from the synchronization clients and is specified by the interface Acceptor. The interface provides a method for a client job to obtain a reference to a connection thread, as well as methods to enable and disable the synchronization server to accept a new client job connection.

```
interface ConThread { Unit command(Command c); }
interface Acceptor {
  ConThread getConnection(Job job);
  Bool isAcceptingConnection();
  Unit suspendConnection();
  Unit resumingConnection(); }

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

interface SyncServer {
  Acceptor getAcceptor();
  Coordinator getCoordinator();
  Snapshot getSnapshot();
  DB getDataBase(); }

interface Snapshot {
  Unit refreshSnapshot(Bool r);
  Unit clearSnapshot();
  Set<Item> getItems();}

interface Item {
  FileEntry getContents();
  ItemType getType();
  Id getAbsoluteDir();
  Unit refresh();
  Unit cleanup();}

interface SyncClient {
  Acceptor getAcceptor();
  DB getDataBase();
  Unit becomesState(State state);
  Unit setAcceptor(Acceptor acceptor);}

interface Job {
  Bool registerItems(CheckPoint checkpoint);
  Maybe<FileSize> processFile(Id id);
  Unit processContent(File file);
  Unit receiveSchedule();}
```

Listing 1.1. ABS interfaces of the replication system

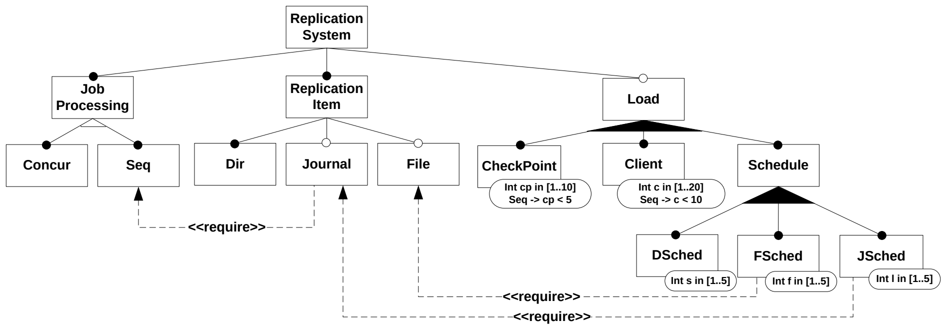

Fig. 3. Feature model of the Replication System

The connection thread and the client job are specified by interfaces ConThread and Job, respectively. Each connection thread is instantiated by the Acceptor component. After the Acceptor receives a connection from a client job, it instantiates a ConThread to carry out the replication protocol. A connection thread is specified by the interface ConThread. The ConThread component has a single method command(), which is asynchronously invoked by Job objects to determine the current state of a replication. The Coordinator component is responsible for coordinating the connections that the Acceptor accepts from synchronization clients. It also provides methods for preparing and clearing replication items before and after replication sessions. The SyncServer component starts the Acceptor and the Coordinator components. It also keeps a reference to the relevant replication snapshot, i.e., the data that is currently being replicated.

Listing 1.1 also shows the ABS interfaces of the components that are part of the synchronization clients. A SyncClient communicates with the SyncServer via job scheduling. At initialization time, the SyncClient schedules a client job to acquire a replication schedule from the server. Using this schedule, the client job creates a new client job for performing the actual replication. Each client job, thereafter, is responsible to request replication schedules and to set up the subsequent jobs for further replication. Each client job receives replication items from a connection thread and updates the synchronization client's files (configuration and data). The client job is specified by interface Job.

5 Modelling Variability

As part of FAS product line, the replication system defines variability on the types of replication item and replication strategy. We capture this variability using the ABS language extensions described in Section 2.

Figure 3 shows the feature diagram of the replication system and Listing 1.2 shows the corresponding μTVL model. Specifically, the replication system has three main features: JobProcessing, RepItem and Load.

The feature JobProcessing requires an alternative choice between the two sub features Seq and Concur, capturing the choice between sequential and concurrent client job processing, respectively.

The feature RepItem allows choosing between three replication item types represented by the features Dir, File and Journal. The Dir feature is mandatory, that is, all versions of the replication system support replicating complete file directories. The File feature is optional and is selected to support replicating a file set, whose files' name matches a particular pattern. the Journal feature is optional and is selected to support replicating database journal. In particular, the Journal feature requires the feature Seq which means that variants of the replication system that support database journal replication may only schedule client jobs sequentially.

```
root ReplicationSystem {
 group allof {
  JobProcessing { group oneof { Seq, Concur }},
  RepItem { group [1..*] { Dir, opt File, opt Journal { require: Seq; }}},
  opt Load {
   group [1..3] {
    Client { Int c in [1 .. 20]; Seq -> c < 10; },
    CheckPoint { Int cp in [1 .. 10]; Seq -> cp < 10; },
    Schedule {
     group [1..3] {
      SSchedule { Int s in [1 .. 5]; },
      FSchedule { Int f in [1 .. 5]; require: File; },
      JSchedule { Int l in [1 .. 5]; require: Journal; }
     }}}}}}
```

Listing 1.2. Feature model of the replication system in μTVL

The feature Load is an optional feature that configures the load of the replication system. It offers sub features Client, CheckPoint and Schedule. The feature Client configures the number of synchronisation clients, and defines the constraint such that if client job processing is sequential, the number of clients must be less than ten. The feature CheckPoint configures the number of updates allowed per execution, defines the constraint such that if the client job processing is sequential, the number of updates must be less than five. The feature Schedule configures the number of locations in the file system at which changes to different replication item types are monitored. It is an optional feature that offers sub-features SSchedule, FSchedule and JSchedule to record the number of locations for directory, file set, and journal replication respectively. Note that FSchedule and JSchedule cannot be selected unless features File and Journal are selected respectively.

The basis replication system supports sequential client job processing. This functionality is implemented by the active class JobImpl. A partial ABS class definition of JobImpl is shown in Listing 1.3. Each instance of the JobImpl class initialises the Boolean field newJob to False and invokes its run method. This method in turn invokes scheduleNewJob() asynchronously. The method scheduleNewJob() waits for field newJob to become True before creating a new instance of Job. Setting newJob to True at the end of the run method ensures that each client job is scheduled sequentially.

The lower half of Listing 1.3 defines the delta module Concurrent. A delta module specifies changes to a basis ABS model, such as the addition, modification or removal of classes, in order to define the shape of the model in another system variant. The delta module Concurrent specifies a class modifier for the class JobImpl that contains a method modifier. The method modifier removes the await statement from the method scheduleNewJob() such that a new instance of the class Job is created as soon as the current Job instance releases the lock of this object group. This allows scheduling client jobs concurrently.

```
class JobImpl(SyncClient c, JobType job) implements Job {
  Bool nj = False;
  Unit newJob() { await nj; new JobImpl(this.c,Replication); }
  Unit run() { .. this!newJob(); .. nj = True; .. }
  Unit state(State state) { .. } ..
}

delta Concurrent {
  modifies class JobImpl {
    modifies Unit newJob() { new JobImpl(this.c,Replication); }}}
```

Listing 1.3. Modeling job processing

```
class Dirs(Id q, DB db) implements Item { .. }

class SnapshotImpl(DB db, Schedules ss)
implements Snapshot {
  Set<Item> items = EmptySet;
  Unit item(Schedule s) {
    if (isSearchItem(s)) {
     Item item = new Dirs(left(item(s)),this.db);
     this.items = Insert(item,this.items);}..}}
```

Listing 1.4. Partial implementation of replication item

Listing 1.4 shows a partial definition of the classes DirectoryItem and SnapshotImpl. The class DirectoryItem defines a replication item for a complete file directory. The class SnapshotImpl represents a replication snapshot. The method item defined in the class SnapshotImpl takes a replication schedule, creates a corresponding Item object and adds it to the set of replication items. By default, this method only handles replication schedules for complete file directories.

In Listing 1.5, two delta modules are shown that contain the necessary functionality and modifications to handle other types of replication items. The delta module FileDelta is applied for file set replication and has three class modifiers. The first modifier adds the class FItem, implementing the interface Item, for handling replication file sets that match a regular expression. The second class modifier changes the class SnapshotImpl by updating the method item to handle replication schedules with file sets; here the statement **original**(s) calls the previous version of the method SnapshotImpl.item(s). The third class modifier changes the initial Main by introducing a new instance field files that records a list of schedules for file set replications. Similarly, the delta module JournalDelta contains the necessary modifications for handling database journal replication. It has three class modifiers to add a new implementation of interface Item, to update the method item to handle replication schedules with data base journals and to add new instance field logs that records a list of schedules for database journal replications.

```
delta FileDelta {
  adds class FItem(Id q, String p, DB db)
  implements Item { .. }

  modifies class SnapshotImpl {
    modifies Unit item(Schedule s) {
      original(s);
      if (isFileItem(s)) {
        Pair<Id,String> it = right(item(s));
        Item item = new FItem(fst(it),snd(it),this.db);
        items = Insert(item,items);
      }}}

  modifies class Main { adds List<Schedule> files = ... }
}

delta JournalDelta {
  adds class Journals(..) implements Item { .. }
  modifies class SnapshotImpl { .. modifies Unit item(..) ..}}
  modifies class Main { adds List<Schedule> logs = ... }
```

Listing 1.5. Deltas for replication items

Listing 1.6 shows the configuration of the replication system product line using the product line configuration language CL. The product line configuration links the modifications contained in the listed delta modules to product features and determines for which feature configurations the modifications have to be applied. In the considered example, the features Dir and Seq are the features provided by the core system. The application condition for delta module FileDelta states that this delta module is applied if feature File is selected, while the application condition for the delta module FSched states that the module is applied if feature FSchedule is selected and that it must be applied after delta module SSched should its corresponding feature be selected.

Listing 1.6 also shows two example product selections for the replication system product line specified in the product selection language PSL. Product P1 defines the basis variant of the replication system that supports the basic set of features, and product P2 that supports both directory and file set replication, concurrent client job scheduling and deploys three SyncClients for receiving replications.

```
productline ReplicationSystem {
  features Dir, File, Journal, Seq, Concur, Client, Schedule, CheckPoint,
           SSchedule, FSchedule, JSchedule;

  delta FileDelta when File;
  delta JournalDelta when Journal;
  delta Concurrent when Concur;
  delta CD(Client.c) when Client;
  delta CP(CheckPoint.cp) when CheckPoint;
  delta SSched(SSchedule.s) when SSchedule;
  delta FSched(FSchedule.f) after SSched when FSchedule;
  delta JSched(JSchedule.l) after FSched when JSchedule;
}

product P1 (Dir, Seq);
product P2 (Dir, File, Concur, Client({c=3}));
```

Listing 1.6. Product configuration and selection

6 Evaluation

This section presents an evaluation of the HATS approach with respect to the ABS language's expressiveness, scalability and usability. The specific criteria have been derived from the HATS project's requirement elicitation activity [17].

6.1 Expressiveness

We evaluate the practical expressiveness and the modeling capabilities of the ABS language. Specifically we investigate 1) how readily and concisely the ABS language expresses various kinds of program structures and behaviours, and 2) its capabilities to express variabilities behaviourally.

Data types Using ABS's algebraic data type, we were able to provide a high-level model of the replication system that abstracts from the underlying physical environment such as operating system, file storage and data base.

Functions Using the combination of ABS's algebraic data types and functions, we were able to use abstract data types such as lists, sets and maps, and subsequently define data types to abstract from the underlying environment such as file storage. We have also found functions to be useful as they guarantee to be free of side-effects and are more amenable to formal reasonsing. However, the ABS language does not support higher order functions. This means we cannot abstract certain behaviour, limiting reusability of function definition. This also implies that we cannot pass functions as parameters to methods.

Polymorphism ABS's algebraic data types and functions support parametric polymporhism, allowing data types and functions to be data-independently defined. However, this ABS classes and methods are not parametrically polymorphic, hence one has to specialise the types of method parameters, reducing the reusability of method implementations.

Syntactic Sugaring To model communications between active objects in ABS we often define the sequence of statements `Fut<A> f = o!m(); await f?; A v = f.get;` in an active object to model invoking method `m()` of object `o` asynchronously, yielding the thread control of its object group and blocking its own execution until the method call returns. We believe the usability of the language could be improved by providing syntactic sugaring to this kind of patterns of behaviours, and at the time of writing we know this types of sugaring are being added to the ABS language.

Concurrency We were able to model the replication system's concurrent behaviour in terms of asynchronous method invocation, this ABS model provides a high level view of the communication between synchronisation server and clients, thereby separating the concerns of the physical communication layers between them and hence reducing the complexity of the model considerably. Another advantage of the cooperative scheduling offered by ABS's concurrency model is that we can safely define a method that modifies a state of an object without the need to explicitly enforce mutual exclusion on that state. Nevertheless, due to inherent nondeterministic scheduling between COGs as well as active objects within a COG [23], it is not possible to enforce fairness over competing active objects when simulating an ABS model. At the time of writing an implementation of a real-time extension of the ABS language is being developed [24]. This would provide the mechanism to specify schedules on the asynchronous method invocations, and allow us to enforce orders of execution to avoid starvation.

Variability Delta Modelling Language offers the expressivity to specify variability at the level of object behaviour. Together with Product Line Configuration, Product Selection and μTVL Language, the ABS language offers a holistic approach to expressing variabilities as features and relating them to object behaviour. We were able to use ABS to incrementally and compositionally develop the replication system product line that yields members that are well-typed and valid with respect to the product line's variability. Nevertheless, the current implementation of DML does not support modification of functions and data types, and this means we cannot capture their variabilities in the same way as classes and interfaces. At the time of writing, we know the implementation of DML is being improved to support functions and data types.

6.2 Scalability

We evaluate the ABS language with respect to scalability and reusability.

Data types Using ABS's algebraic data types, we separate the concern of the replication system's physical environments such as operation system, file storage and data bases from its ABS model. This allows us to scale the replication system product line model, such as increasing the number of SynClients, without being constrained by the physical environment.

Modularity The module system allows us to model both the commonality and the variability of the replication system separately and incrementally. Specifically, we started modeling the commonality of the product line independent from the product line's variability and individual components of the replication system commonality are modeled in separate modules (and files). Moreover, we modeled the product line's variability in terms of delta modules, this allows variation to be modeled incrementally while dividing delta modules in terms of the components which variations are to be resolved.

Code reusability DML provides the mechanism to express variability at the level of behaviour. This together with functional and object composition, the ABS language provides a wide range of mechanism 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. Nevertheless, while the current implementation of DML supports **original**() in method modifiers, which invokes that method's previous implementation, it does not support **original**() of a specific implementation, this has reduced code reusability when resolving conflict [12]. At the time of writing, DML is being improved to support delta-specific **original**().

Timing and resource information The current ABS semantics does not take time and the environment's resources into consideration. This separation of concerns allows one to focus on functional and partial correctness. Nevertheless, at the time of writing, an implementation of a real-time extension of the ABS language is being developed [24]. This would allow ABS models to express behavioural constraints due to timing information as well as the resources of the environment. This would enable one to analyse an ABS model with specific environment constraints such as process speed, memory etc.

6.3 Usability

In this section we evaluate ABS with respect to its overall usability, focusing on the ease of adoption and learnability, and taking into account the ABS tool suite as well as the language's syntax and semantics.

Note that the case study has been conducted in tandem with the development of the ABS language and tool suite, the case study, which has been conducted over the span of 14 months, has consequently led to many enhancement and fixes. Specifically, the HATS project employs an open source ticket tracking system (`http://trac.edgewall.org/`) to track bugs and feature requests, and the case study has brought about ten enhancements and over sixty fixes. As the ABS tool suite has been at development stage during the case study, our evaluation of usability would take this into account.

Syntax and semantics ABS has been designed to be as easy to learn as possible by building on language constructs well known from mainstream programming languages. Both functional and sequential imperative fragments of ABS can be easily acquainted by users with a working knowledge of any functional and object-oriented languages. However, it seems not as easy at first to learn the concurrent fragment of ABS, especially for those who are used to the multithreaded concurrency model. We believe this issue is remedied in twofold: 1) the availability of literature such as the technical papers [23, 13], the tutorial chapter [11] and case studies [18], and 2) the support of the ABS tool suite [36].

Compiler front end The ABS tool suite comes with a front end that takes an ABS model, performs parsing and type checking, and outputs the model's Abstract syntax tree (AST). The design of the ABS language and the availability of the front end guarantees that the ABS model constructed in the case study is well-typed. Moreover, the front end allows product derivation based on the the ABS model's product line configuration and product selection.

Maude/Java back ends The current version of the ABS tool suite comes with a back end for Maude and Java: The Maude back end takes the type checked AST of an ABS model and outputs the corresponding Maude model that can be then simulated using the Maude engine (`maude.cs.uiuc.edu`). Using both the front end and the Maude back end, we were able to quickly simulate multiple versions of the replication system. The ABS tool suite also provides a Java back end that takes the type checked AST of an ABS model and outputs the corresponding Java source codes that can be compiled and executed independently. We have found the Java back end to be particularly useful when used in conjunction with the ABS debugger.

Eclipse plugin – The ABS Eclipse plugin provides syntax highlighting, content completion and code navigation similar to those provided by the Eclipse JDT (`www.eclipse.org/jdt`) for Java. The plugin also integrates the front end and back ends as a singe source technology such that construction, compilation and simulation of ABS models can be carried out directly via the Eclipse IDE. We have found the availability of the IDE greatly increases the scalability of the HATS approach during modeling. The ABS Eclipse plugin can be installed as a bundle via its Eclipse update site `tools.hats-project.eu/update-site`. A recent version of the ABS Eclipse plugin provides the capabilities to import and navigate ABS packages; ABS packages are JAR files containing ABS source codes. We believe this feature increases ABS's applicability in the industry where collaborative software development is prevalent and third party libraries are heavily used.

Debugger The ABS Eclipse plugin offers a debugging perspective for debugging ABS models. The novelty of this perspective is that users can define explicitly the order in which asynchronous method invocations are executed within a concurrent object group. The debugger also offers the option to save and replay histories of asynchronous method invocations. These facilities greatly ease our task of debugging and reproducing bugs during the case study.

Visualization Through the ABS Eclipse plugin's debugging perspective, the ABS tool suite offers a visualization tool that generates UML sequence diagrams of asynchronous communications between concurrent object groups during the debugging session. Sequence diagrams provide high-level views of the communications between components in the replication system, and this increases our understanding of the system's concurrent behaviour.

7 Related work

In this section we consider related work in the context of abstract behavioural, variability modelling, and evaluating SPL engineering methodologies.

The ABS language is a modelling language that aims to close the gap between design-level notations and implementation languages. The concurrent object model of ABS based on asynchronous communication and a separation of concern between communication and synchronization is part of a trend in programming languages today, due to the increasing focus on distributed systems. For example, the recent programming language Go (`http://golang.org`, promoted by Google) shares in its design some similarities with ABS: a nominal type system, interfaces (but no inheritance), concurrency with message passing and non-blocking receive. The internal concurrency model of concurrent objects in ABS stems from the intra-object cooperative scheduling introduced in Creol [16] This model allows active and reactive behavior to be combined within objects as well as compositional verification of partial correctness properties [1].

Existing approaches to express variability in modelling and implementation languages can be classified into two main categories [35, 25]: annotative and compositional. As a third approach, model transformations are applied for representing variability mainly in modelling languages.

Annotative approaches consider one model representing all products of the product line. Variant annotations, e.g., using UML stereotypes in UML models [19] or presence conditions [15], 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. [31] 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 [7] define which parts of the product artifacts have to be modified for certain products.

Compositional approaches, such as delta modelling [33], associate model fragments with product features that are composed for a particular feature configuration. A prominent example of this approach is AHEAD [9], 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 composition techniques [21, 35, 30]. Apel et al. [2] apply model superposition to compose model fragments.

In feature-oriented software development (FOSD) [9], features are considered on the linguistic level by feature modules. Apart from Jak [9], there are various other languages using the feature-oriented paradigm, such as FeatureC++ [3], FeatureFST [4], or Prehofer's feature-oriented Java extension [32]. In [29, 4], combinations of feature modules and aspects are considered. In [5], an algebraic representation of FOSD is presented. Feature Alloy [6] instantiates feature-oriented concepts for the formal specification language Alloy.

Model transformations are used to represent product variability mainly on the artifact modelling level. The common variability language (CVF) [20] represents the variability of a base model by rules describing how modelling elements of the base model have to be substituted in order to obtain a particular product model. In [22], graph transformation rules capture artifact variability of a single kernel model comprising the commonalities of all systems.

There have been interests to evaluate SPL methodologies using case studies [26, 27]. In Lopez-Herrejon et al. work [26], for example, they propose the Graph Product Line (GPL) as a standard problem to implement for evaluating SPL methodologies. In this work they compare qualities such as performance and lines of code between implementations of GPL using the GenVoc SPL methodology [8]. There are also evaluation strategies that focus on other concerns such as tool support. For example, Matinlassi [28] compares several SPL methodologies with respect to qualities such as tool support, guidance and application domains.

8 Summary

In this paper, we presented an evaluation on the HATS approach by conducting a case study on an industrial-scale software product line of a distributed and highly configurable software system using the ABS language and its accompanying ABS tools suite. We modelled the replication system's commonality using Core ABS and the replication system's variability using the Full ABS. At the time of writing the replication system product line ABS model consists of 5000 lines of code, and defines 40 classes, 43 interfaces, 15 features, 8 deltas and 12108 products. Based on the case study we provided an evaluation of the ABS language with respect to practical expressiveness and modeling capabilities, scalability and usability.

By performing the case study in tandem the development of the HATS approach, we were able to provid timely feedback on language expressiveness and the applicability of the modeling tools. The work on the case study has influenced decisions in the design of the ABS language, and in both the enhancements and fixes to the tools suite. As the HATS approach continues to mature, we aim to extend the replication system case study to capture further variabilities and evolution scenarios, and to conduct formal validation and verification of the replication system using the formal analysis tools developed for ABS models.

References

1. W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programming*, 2011. In Press. To appear.

2. S. Apel, F. Janda, S. Trujillo, and C. Kästner. Model Superimposition in Software Product Lines. In *International Conference on Model Transformation (ICMT)*, 2009.
3. S. Apel, T. Leich, M. Rosenmüller, and G. Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, 2005.
4. S. Apel and C. Lengauer. Superimposition: A language-independent approach to software composition. In *Software Composition*, volume 4954 of *Lecture Notes in Computer Science*, pages 20–35. Springer-Verlag, 2008.
5. S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming*, 75(11):1022 – 1047, 2010.
6. S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Detecting Dependences and Interactions in Feature-Oriented Design. In *IEEE International Symposium on Software Reliability Engineering*, 2010.
7. C. Atkinson, J. Bayer, , and D. Muthig. Component-Based Product Line Development: The KobrA Approach. In *SPLC*, 2000.
8. D. Batory and B. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, Feb. 1997.
9. D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.*, 30(6), 2004.
10. D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, G. Puebla, B. Weitzel, and P. Y. H. Wong. HATS: A Formal Software Product Line Engineering Methodology. In *Proceedings of International Workshop on Formal Methods in Software Product Line Engineering*, Sept. 2010.
11. D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Y. H. Wong. Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *LNCS*, pages 417–457, June 2011.
12. D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract Delta Modeling. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 13–22, New York, NY, USA, Oct. 2010. ACM.
13. D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte. Variability modelling in the ABS language. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, Lecture Notes in Computer Science. Springer-Verlag, 2011. To appear.
14. A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, Nov. 2010.
15. K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
16. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, 2007.
17. Requirement Elicitation, Aug. 2009. Deliverable 5.1 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.
18. Evaluation of Core Framework, Aug. 2010. Deliverable 5.2 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

19. H. Gomaa. *Designing Software Product Lines with UML*. Addison Wesley, 2004.
20. Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In *SPLC*, 2008.
21. F. Heidenreich and C. Wende. Bridging the Gap Between Features and Models. In *Aspect-Oriented Product Line Engineering (AOPLE'07)*, 2007.
22. P. K. Jayaraman, J. Whittle, A. M. Elkhodary, and H. Gomaa. Model composition in product lines and feature interaction detection using critical pair analysis. In *MoDELS*, pages 151–165, 2007.
23. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, Lecture Notes in Computer Science. Springer-Verlag, 2011. To appear.
24. E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In B. Beckert and C. Marché, editors, *Proc. International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'10)*, volume 6528 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2011.
25. C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE*, pages 311–320, 2008.
26. R. E. Lopez-Herrejon and D. S. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *International Conference on Generative and Component-Based Software Engineering (GCSE'01)*, volume 2186 of *LNCS*, pages 10–24, 2001.
27. R. E. Lopez-Herrejon, D. S. Batory, and W. R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *European Conference on Object-Oriented Programming (ECOOP'05)*, volume 3586 of *Lecture Notes in Computer Science*, pages 169–194. Springer-Verlag, 2005.
28. M. Matinlassi. Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA. In *International Conference on Software Engineering (ICSE'04)*, pages 127–136. IEEE Computer Society, 2004.
29. M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT FSE*, pages 127–136. ACM, 2004.
30. N. Noda and T. Kishi. Aspect-Oriented Modeling for Variability Management. In *SPLC*, 2008.
31. K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, Heidelberg, 2005.
32. C. Prehofer. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer-Verlag, 1997.
33. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proc. of 15th Software Product Line Conference (SPLC 2010)*, Sept. 2010.
34. I. Schaefer and R. Hähnle. Formal methods in software product line engineering. *IEEE Computer*, 44(2):82–85, Feb. 2011.
35. M. Völter and I. Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *SPLC*, pages 233–242, 2007.
36. P. Y. H. Wong, E. Albert, R. Muschevici, J. Proença, J. Schäfer, and R. Schlatte. The ABS Tool Suite: Modelling, Executing and Analysing Distributed Adaptable Object-Oriented Systems, Sept. 2011. Submitted for publication.