An extension of the ABS toolchain with a mechanism for type checking SPLs

An Extension of the ABS Toolchain with a Mechanism for Type Checking SPLs*

Ferruccio Damiani[1], Michael Lienhardt[1], Radu Muschevici[2], and Ina Schaefer[3]

[1] University of Torino, Torino, Italy
{ferruccio.damiani, michael.lienhardt}@unito.it
[2] Technische Universität Darmstadt, Darmstadt, Germany
radu@cs.tu-darmstadt.de
[3] Technische Universität Braunschweig, Braunschweig, Germany
i.schaefer@tu-braunschweig.de

Abstract. A Software Product Line (SPL) is a set of similar programs, called variants, with a common code base and well documented variability. Because the number of variants in an SPL can be large, checking them efficiently (e.g., to ensure that they are all well-typed) is a challenging problem. Delta-Oriented Programming (DOP) is a flexible approach to implement SPLs. The Abstract Behavioral Specification (ABS) modeling language and toolchain supports delta-oriented SPLs. In this paper we present an extension of the ABS toolchain with a mechanism for checking that all the variants of an SPL can be generated and are well-typed ABS programs. Currently we have implemented only part of this mechanism: our implementation (integrated in version 1.4.2 of the ABS toolchain and released in April 2017) checks whether all variants can be generated, however it does not check, in particular, whether the bodies of the methods are well-typed. Empirical evaluation shows that the current implementation allows for efficient partial type checking of existing ABS SPLs.

1 Introduction

Recent fundamental changes of deployment platforms (cloud, multi-core) together with the emergence of cyber-physical systems and the internet of things imply that modern software must support variability [22] and emphasize the need for modeling languages capturing system diversity.

The *Abstract Behavioral Specification* (ABS) [7] modeling language and toolchain has been designed to fill the gap between structural high-level modeling languages (e.g., UML) and implementation-close formalisms (including programming languages such as C/C++, C#, or Java). It facilitates the precise modelling

* This work has been partially supported by: EU Horizon 2020 project HyVar (www.hyvar-project.eu), GA No. 644298; ICT COST Action IC1402 ARVI (www.cost-arvi.eu); Ateneo/CSP D16D15000360005 project RunVar (runvar-project.di.unito.it); LOEWE initiative to increase research excellence in the state of Hesse, Germany as part of the LOEWE Schwerpunkt CompuGene.

Language	Role
Core ABS	Specifies base behavioural models
Micro Textual Variability Language Feature (μTVL)	Feature models
Delta Modelling Language (DML)	Modifications to base behavioural models
Product Line Configuration Language (CL)	Links features and delta modules, configures deltas with attributes

Fig. 1. Language definitions in ABS

of the behaviour of highly configurable distributed systems, and has been successfully used in industry [19,17,1].

Figure 1 illustrates the different languages comprised in ABS. The basis, *Core ABS*, is a strongly typed, abstract, object-oriented, concurrent, fully executable modeling language. The other three languages support the implementation of delta-oriented *Software Product Lines* (SPLs) of Core ABS programs.

An SPL is a set of similar programs, called *variants*, with a common code base and well documented variability [8]. *Delta-Oriented Programming* (DOP [21,5] and [2, Sect. 6.6.1]) is a flexible and modular approach to implement SPLs. A delta-oriented SPL comprises a *feature model*, an *artifact base*, and *configuration knowledge*. The feature model provides an abstract description of variants in terms of *features*: each feature represents an abstract description of functionality and each variant is identified by a set of features, called a *product*. The artifact base provides language dependent artifacts that are used to build the variants: it consists of a *base program* (written in the same language in which variants are written) and of a set of *delta modules* (*deltas* for short), which are containers of modifications to a program—for Core ABS programs, a delta can add, remove or modify classes, interfaces, fields and methods. Configuration knowledge connects the feature model with the artifact base by associating with each delta an *activation condition* over the features and specifying an *application ordering* between deltas. Once a user selects a product, the corresponding variant is derived by applying the deltas with a satisfied activation condition to the base program according to the application ordering. To avoid over-specification the application ordering can be partial—this opens the issue of ensuring *unambiguity* of the product line, i.e., for each product, any total ordering of the activated deltas that respects the partial ordering must generate the same variant.

With respect to the languages in Fig. 1: Core ABS is for writing base programs; μTVL is for feature models; *DML* is for deltas; and *CL* is for configuration knowledge.

As the number of variants in an SPL can be large, checking them efficiently (e.g., to ensure that they are all well-typed) is a challenging problem. Until the release of version 1.4.2 in April 2017, the ABS toolchain[4] did not officially provide any dedicated support for checking that an SPL is unambiguous (cf. the discussion above) and for *type checking an SPL* (i.e., to ensure that all its variants

[4] ABS language & tools: `http://abs-models.org/`

are generable[5] and well-typed).[6] In order to type check an SPL developers had to generate all its variants and type check each of them in isolation using the Core ABS type checker. Evaluation of ABS against industrial requirements [13,16] repeatedly identified lack of tool support for SPL unambiguity checking and SPL type checking as a major usability issue.

In this paper we present an extension of the ABS toolchain with an *SPL unambiguity and type checking mechanism* (*SPL checking mechanism*, for short). This mechanism is an adaptation to ABS of an approach formalized for Imperative Featherweight Delta Java (IFΔJ) [5,12], a minimal core calculus for delta-oriented SPLs where variants are written in IFJ [5], an imperative version of FJ [18]. Currently we have implemented only part of the checking mechanism: our implementation checks whether the SPL is unambiguous and whether all variants can be generated, however it does not check, in particular, whether the bodies of the methods are well-typed.

Empirical evaluation shows that the extended toolchain allows for efficient partial checking of existing ABS product lines, providing a significant performance increase with respect to generating and fully type checking each variant in isolation using the Core ABS type checker. This result raises our confidence that the (currently under development) implementation of the full SPL checking mechanism will remain similarly performant.

The paper is organized as follows. In Sect. 2 we provide an overview of ABS and recall its minimal fragment FDABS [9]. In Sect. 3 we illustrate the SPL checking mechanism by means of FDABS. In Sect. 4 we present the implementation of part of the SPL checking mechanism for the complete ABS language and its integration in the ABS toolchain. In Sect. 5 we show how the extended ABS toolchain is applied to check two SPLs developed in two different industrial modeling scenarios. In Sect. 6 we discuss related work and conclude in Sect. 7.

2 FDABS: a Minimal Language for Core ABS and Deltas

In this section we provide an overview of ABS product lines and of the Featherweight Delta ABS (FDABS) [9] language, the minimal fragment of ABS used in Sect. 3 to illustrate the SPL checking mechanism.

We illustrate ABS product lines by means of a version of the *Expression Product Line* (EPL) benchmark [20] (see also [5]) defined by the following grammar which describes a language of numerical expressions:

Exp ::= Lit | Add Lit ::= <non−negative−integers> Add ::= Exp "+" Exp

Each variant of the EPL contains an interface `Exp` that represents an expression equipped with a subset of the following operations: `eval`, which returns the value

[5] The generation of a variant fails whenever the application of an activated delta fails. The application of a DML delta to a Core ABS program fails, e.g., if the delta tries to add a class that is already present in the program, or tries to remove or modify a class that is not present in the program.

[6] The development of the SPL checking mechanism described in this paper started in 2015 and a prototypical version has been made available since June 2015.

```
// Feature model (written in μTVL)
productline EPL;
features Flit, Fadd, Feval, Fprint;
root EPL { group allof {
  Flit, opt Fadd, Feval, opt Fprint }}
```

```
// Configuration knowledge (written in CL)
delta Dadd when Fadd;
delta Dlit_NOprint when !Fprint;
delta Dadd_NOprint after Dadd
                    when Fadd && !Fprint;
```

```
1  // Base program (written in the FABS
2  // subset of Core ABS)
3  interface Exp {
4    Int eval();
5    String toString();
6  }
7  class Lit implements Exp {
8    Int val;
9    Exp set(Int x) {
10       this.val=x; return this;
11   }
12   Int eval() { return this.val; }
13   String toString() {
14     return this.val.toString();
15   }
16 }
```

```
17 // Deltas (written in the FDML subset of DML)
18 delta Dadd;
19 adds class Add implements Exp {
20   Exp a;      Exp b;
21   Exp set(Exp a, Exp b) {this.a=a; this.b=b; return this;}
22   Int eval() { return this.a.eval() + this.b.eval(); }
23   String toString() {
24     return this.a.toString() + "+" + this.b.toString(); }
25 }
26
27 delta Dlit_NOprint;
28 modifies interface Exp { removes toString; }
29 modifies class Lit { removes toString; }
30
31 delta Dadd_NOprint;
32 modifies class Add { removes toString; }
```

Fig. 3. ABS code for the EPL: feature model (top left, cf. the graphical representation in Fig. 2); configuration knowledge (top right); and artifact base (bottom)

of the expression as an integer; and `toString`, which returns the expression as a `String`.

Figure 2 shows the feature model of the EPL depicted as a feature diagram. The EPL has four products, described by four features: the mandatory features `Flit` and `Feval` correspond to the presence of literal expression (i.e., numbers) and the `eval` method, respectively; the optional features `Fadd` and `Fprint` provide the `Add` class (for sum expression) and the `toString` method, respectively.

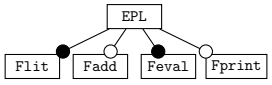

Fig. 2. EPL feature model

Figure 3 illustrates the ABS code implementing the EPL. Configuration knowledge (Fig. 3, top right) lists the names of the deltas and specifies when and how each delta must be applied to generate a given variant: the `when` clause declares the activation condition of a delta, while the `after` clauses specify the application ordering between deltas (cf. the description of DOP in Sect. 1). The artifact base comprises the base program and three deltas. The base program (Fig. 3, bottom left) declares the interface `Exp` and the class `Lit` (for literals)—note that both `Exp` and `Lit` declare the `toString` method. The delta `Dadd` (activated when feature `Fadd` is selected) adds the `Add` class for sum expression, with the methods `eval` and `toString`. The delta `Dlit_NOprint` (activated when feature `Fprint` is not selected) removes the `toString` method from the `Exp` interface and `Lit` class, while the delta `Dadd_NOprint` removes it from the `Add` class.

The abstract syntax of FDABS is given in Fig. 4. An FDABS product line L, see Fig. 4 (top left), consists of: a feature model $\mathcal{M}$, configuration knowledge $\mathcal{K}$, a base program P, and a (possibly empty) set $\overline{\Delta}$ of deltas—following Igarashi et al. [18], $\overline{X}$ denotes a finite (possibly empty) sequence of syntactic elements of kind X, and the empty sequence is denoted by $\emptyset$. In FDABS there is no syntax for feature models and configuration knowledge.

$L ::= \mathcal{M}\ \mathcal{K}\ P\ \overline{\Delta}$

Δ ::= `delta` `d`; $\overline{IO}\ \overline{CO}$
IO ::= `adds` ID | `removes` `I`; | `modifies` `I` [`extends` $\overline{\texttt{I}}$] { $\overline{HO}$ }
HO ::= `adds` HD; | `removes` `m`;
CO ::= `adds` CD | `removes` `C`; | `modifies` `C` [`implements` $\overline{\texttt{I}}$] { $\overline{AO}$ }
AO ::= `adds` FD | `removes` `f`; | `adds` MD | `removes` `m`; | `modifies` MD

$P ::= \overline{ID}\ \overline{CD}$

ID ::= `interface` `I` [`extends` $\overline{\texttt{I}}$] { $\overline{HD}$; }
CD ::= `class` `C` [`implements` $\overline{\texttt{I}}$] { $\overline{AD}$ }
HD ::= `I` `m`($\overline{\texttt{I x}}$)

AD ::= FD | MD
FD ::= `I` `f`;
MD ::= HD { $\overline{s}$ `return` e; }

Fig. 4. Syntax of FDABS (top left), FDML (top right) and FABS (bottom)

The language for base programs, Featherweight Core ABS (FABS), is given in Fig. 4 (bottom). The non-terminal P represents programs, ID interface declarations, CD class declarations, HD method headers, AD attribute declarations, FD field declarations, MD method declarations, e expressions, and s statements. To save space, we do not specify expression and statement syntax.

The language for deltas, Featherweight DML (FDML), is given in Fig. 4 (top right). A delta Δ comprises a name `d` and a set of operations IO on interfaces and operations CO on classes. These operations can add or remove interfaces and classes, or modify their content by adding or removing attributes. Moreover, these operations can also change the set of interfaces implemented by a class or extended by an interface by means of an optional `implements` or `extends` clause in the `modifies` operation, respectively. Finally, it is also possible to *modify* the body of a method with the `modifies` operation, where the new method may call the original implementation of the method using the keyword `original`.

3 The Checking Mechanism for FDABS Product Lines

SPL analysis approaches can be classified into three main categories [25]: *product-based* analyses work only on generated variants (or models of variants); *family-based* analyses work on the artifact base, without generating any variant or model of variant, by exploiting feature model and configuration knowledge to derive results about all variants; *feature-based* analyses work on the reusable artifacts in the artifact base (base program and deltas in DOP) in isolation, without using feature model and configuration knowledge, to derive results on all variants.

In this section we outline how the SPL unambiguity and type checking mechanism formalized for IFΔJ by Bettini et al. [5,12], can be reformulated for FABS. The type checking mechanism comprises three steps: i) a feature-based analysis step that extracts suitable constraints from the base program and the deltas; ii) a family-based step that builds a data structure, called the *product family generation trie* (PFGT) of the SPL, that is exploited (in the next step) to optimize generation and check of the constraints; and iii) a product-based step that uses the constraints extracted in the first step to generate and check, for each product of the SPL, constraints that are satisfiable if and only if the associated variant is well-typed.

3.1 Unambiguity Checking. In the formalization of DOP by Bettini et al. [5,12] the application ordering is specified by providing a totally ordered partition of the set of the deltas, which is interpreted as defining the partial ordering such that: *two deltas in the same set are not comparable, and two deltas in different sets are ordered according to the partition ordering.* Bettini et al. [5] pointed out that, if the application ordering is specified (as described above) by a totally ordered partition of the deltas, then unambiguity of the SPL is implied by a stronger condition, called *strong unambiguity*, which states that: i) if a delta in a set of the partition adds or removes a class/interface then no other delta in the same set adds, removes or modifies the same class/interface; and ii) the modifications of the same class/interface in different deltas in a same set are disjoint (i.e., there is at most one delta operation for each field, method, `implements` clause, method header, and `extends` clause). They also pointed out that strong unambiguity can be efficiently checked by only analyzing the *delta signature table* (DST) of a product line, which is a table that has an entry for each delta declared in the artifact base: it associates with each delta name a data structure, called *delta signature*, containing the information provided by the delta deprived of the bodies of its methods.

The ABS toolchain uses the `after` clauses (cf. Sect. 2) to compute a totally ordered partition of the set of deltas, that we call the *canonical partition*, which in turn defines the application ordering as described above—we call it the *canonical application ordering*. Therefore, we can directly exploit the result on strong unambiguity [5]. The canonical partition is computed by: building the `after` graph (i.e., a direct graph where the nodes are the names D of the deltas, and there is an edge from D_1 to D_2 iff there is a clause D_2 `after` D_1); checking whether the `after` graph is acyclic; computing for each delta the length l of the longest path from a source node in the `after` graph; and putting in the same set all the deltas that have the same l.

3.2 Type Checking Step (i): Extracting Constraints. The constraints extracted from a FABS program, called *program constraints*, encode its typing requirements. Figure 5 lists the constraints extracted from method bodies (i.e., from expressions and statements). These constraints are associated with the corresponding method and class via `with` clauses. Consider, for instance, the base program in Fig. 3. The method `set` (line 9) of class `Lit` has return type `Exp`. It updates the field `val` with the variable `x` of type `Int`, and returns `this`. Therefore *in class* `Lit` *the body of method* `set` *requires that field* `val` *must exist and be of type* `Int`*, and class* `Lit` *must implement (either directly or indirectly) interface* `Exp`. This requirement is encoded by the constraint:

`Lit with {set with {field(Lit, val, Int), subtype(Lit, Exp)}}`

Program constraints can be straightforwardly checked against the *signature* of the corresponding program, which is a data structure containing the information

`class(C)`	*class* `C` *must be defined*
`interface(I)`	*interface* `I` *must be defined*
`subtype(C, I)`	`C` *must implement (directly or not)* `I`
`subtype(I, I′)`	`I` *must extend (directly or not)* `I′`
`field(C, f, I)`	`C` *must have field* `f` *of type* `I`
`meth(C, m, Ī → I′)`	`C` *must have method* `m` *of type* `Ī → I′`
`meth(I, m, Ī → I′)`	`I` *must have method* `m` *of type* `Ī → I′`

Fig. 5. Constraints for expressions and statements

provided by the program deprived of the bodies of its methods.

The constraints extracted from a delta, called *delta constraints*, contain delta operations on program constraints. Consider the deltas in Fig. 3. The delta `Dadd` in line 18 *adds the* `Add` *class*. This is encoded by the constraint:

$$\texttt{adds}(\texttt{Add}\ \texttt{with}\ \mathbf{C}) \tag{1}$$

where $\mathbf{C}$ are the constraints extracted from the body of the class `Add`. They also contain the following constraint:

$$\texttt{toString}\ \texttt{with}\ \{\texttt{meth}(\texttt{Exp}, \texttt{toString}, \emptyset \rightarrow \texttt{String})\} \tag{2}$$

stating that *body of method* `toString` *requires that method* `toString` *of type* $\emptyset \rightarrow$ `String` *must be defined in interface* `Exp`. The delta `Dadd_NOprint` in line 27 *removes the method* `toString` *from the class* `Add`, this is encoded by the constraint:

$$\texttt{modifies}(\texttt{Add}\ \texttt{with}\ \{\texttt{removes}(\texttt{toString})\}) \tag{3}$$

The *abstraction* of a program consists of its signature and constraints. The *delta abstraction table* (DAT) maps each delta name to a data structure that contains the signature and the constraints of the delta. For each product, the abstraction of the corresponding variant (recall that a variant is a FABS program) can be straightforwardly generated from the abstraction of the base program and the DAT: the signature of the variant is generated by applying the signatures of the activated deltas to the signature of the base program; and the constraints of the variant are generated by applying the constraints of the activated deltas to the constraints of the base program. Therefore, each variant can be type checked, without being generated, by generating and checking its abstraction (i.e., by checking its constraints against its signature). For instance, consider the product $\{\texttt{Flit}, \texttt{Fadd}\}$ of the EPL (in Fig. 3): the deltas `Dadd`, `Dlit_NOprint` and `Dadd_NOprint` are activated and the constraints extracted from them are applied to the constraints extracted from the base program. Because, e.g., the constraint (3) is applied after the constraint (1), the constraint (2) is removed from the programs constraints generated for the variant. The resulting constraints are thus validated against the signature of that variant, in which neither `Exp` nor `Lit` have the method `toString`.

3.3 Type Checking Step (ii): Building the PFGT. Given a strongly unambiguous SPL (cf. Sect. 3.1), any total ordering of the deltas that is compatible with the canonical application ordering can be used to generate the variants. Each of these total orderings determines a set $\mathbf{S}$ that contains, for each product, the ordered sequence of the deltas activated by the product (note that different total orderings may generate the same set). The trie [15] for $\mathbf{S}$, called the *product family generation trie* (PFGT), is a tree, where each edge is labeled by a delta, that represents all sequences in $\mathbf{S}$ by factoring out the common prefixes—the structure of the trie for a set of sequences $\mathbf{S}$ is uniquely determined by $\mathbf{S}$. Moreover, each node of the PFGT that corresponds to a sequence in $\mathbf{S}$ is labeled by the associated product (by construction, these nodes include all the leaves).

The ABS toolchain generates the variants by applying the activated deltas according to a total ordering, that we call the *topological application ordering*, which is a topological sorting of the direct graph describing the canonical appli-

cation ordering (this graph contains at least all the edges of the `after` graph, cf. Sect. 3.1). Our implementation of the SPL type checking mechanism (illustrated in Sect. 4) builds the PFGT for (the set **S** of sequences of deltas determined by) the topological application ordering.

3.4 Type Checking Step (iii): Checking All Variants. All the variant program abstractions can be efficiently generated and checked by traversing the PFGT in depth-first order and marking each node N with a program abstraction [12].

- The root node is marked with the abstraction of the base program.
- Each non-root node is marked with the program abstraction obtained by applying the abstraction of the delta that labels the edge between N and it parent to the program abstraction that marks N's parent.
- If N represents a product, the program abstraction generated for marking N is checked, thus establishing whether the associated variant is well-typed.
- If the generation of the program abstraction for N fails (i.e., the application of the delta abstraction that labels the edge entering N to the program abstraction marking N's parent fails), then the subtree with N as root is pruned, and an error message informs that the variants associated with the products in the subtree cannot be generated.

4 Integration into the ABS Toolchain

Until the release of version 1.4.2 in April 2017, the ABS toolchain was structured as a pipeline of three components:

1. The *Parser* component takes in input an ABS SPL (i.e., a set of files written in the different languages listed in Table 1) and produces the Extended Abstract Syntax Tree (E-AST) representing the artifact base—it also checks whether the `after` graph is acyclic and, if so, it computes the topological application ordering (cf. the explanation in last paragraph of Sect. 3.3).
2. The *Rewriter* component generates the variant corresponding to a given product: it applies the activated deltas to the base program to produce the Core AST (C-AST) of the variant.
3. The *Semantic Analysis and Backend* component supports analyzing the C-AST by using the different tools developed for the Core ABS language [6]. The first of the analysis to be performed is type checking.

In order to check that all variants can be generated and are well-typed Core ABS programs, the user had to generate all the variants and type check each of them in isolation using the Core ABS type checker. Moreover, there was no support for checking the unambiguity of the SPL.

Currently we have implemented part of the SPL checking mechanism illustrated in Sect. 3. Our implementation is available as a novel component in version 1.4.2 of the ABS toolchain. The novel component, called the *SPL Checking*

component, is inserted into the pipeline between the Parser and the Rewriter components. We stress that, unlike in Sect. 3, the implementation is based not merely on FDABS, but on the complete ABS language.

4.1 An Overview of the Novel Component. The SPL Checking component performs the following steps:

1. A feature-based step that computes the signature of the base program and the DST (i.e., the signature of each delta)—cf. Sect. 3.2. Some errors in the artifact base can already be detected and reported during this step.
2. A step that, by using only the DST and the canonical partition, checks whether the SPL is strongly unambiguous—cf. Sect. 3.1.
3. A family-based step that (by using only configuration knowledge) builds the PFGT for the topological application ordering—cf. Sect. 3.3.
4. A product-based step that (by using only the signature of the base program, the DST and the PFGT) efficiently generates all the variant signatures (thus checking whether all the variants can be generated)—cf. Sect. 3.4.

It is worth observing that the signature of a variant contains enough information to check whether, e.g.: each interface occurring in some `implements` or `extends` clause, or in some method header or fields declaration is declared in the variant; the `extends` relation is acyclic; each interface has no (defined or inherited) incompatible method headers; each class implements the methods of the interfaces listed in its `implements` clause—we are currently working on extending step 4 above to perform these checks.

Moreover, we are also working on fully implementing the SPL type checking mechanism, i.e., on replacing steps 1 and 4 above by:

1′ A feature-based step that computes the abstraction of the base program and the DAT (i.e., the abstraction of each delta)—cf. Sect. 3.2. Some errors in the artifact base can already be detected and reported during this step.

4′ A product-based step that (by using only the abstraction of the base program, the DAT, and the PFGT) efficiently generates and checks all the variant abstractions (thus checking whether all the variants can be generated and are well-typed)—cf. Sect. 3.4.

4.2 On Building the PFGT. The PFGT, which caches and keeps track of common delta application sequences, is used to improve the efficiency of generating variant signatures (cf. Sect. 3.4). In order to build the PFGT we need to enumerate all the products of the SPL (cf. Sect. 3.3). μTVL supports feature attributes (Booleans and integers) as a way of permitting more fine-grained product specifications. An established approach [4] to determine all the products of a feature model with Boolean and integer attributes is to express it as a constraint satisfaction problem (CSP) over Boolean or integer variables. The

solutions to the CSP are all the attributed products, i.e., all the attributed feature combinations allowed by the attributed feature model. The ABS toolchain[7] uses an off-the-shelf CSP library[8] for this task.

Even though ABS support feature attributes, we omit these when enumerating all solutions to the attributed feature model in order to build the PFGT, as they are irrelevant for type checking variants. This is due to the fact that, for each product with attributes, the set of the activated deltas is determined by the selected features alone, i.e., it is not possible to specify the activation of a delta based on the values of feature attributes. Feature attributes only influence the configuration to the extent of assigning concrete values to variables in the ABS source code—this is achieved by passing feature attributes as attributes to the activated deltas. The delta abstraction is built by abstracting away from these. The correct use of attributes can be checked separately, for each delta (together with its `after` clause and `when` clause) in isolation. A convenient side-effect of abstracting away attributes is that it may significantly reduce the number of products—consider, for example that the introduction of a single, unbound integer attribute multiplies the number of products by up to 2^{32}.

5 Case Studies and Evaluation

We tested the SPL Checking component on two industrial SPLs implemented in ABS.

The first case study is provided by the Fredhopper Access Server (FAS), a distributed web-based software system for Internet search and merchandising, developed by Fredhopper B.V. (now ATTRAQT). In particular, we considered the Replication System, a subsystem for ensuring data consistency across the FAS deployment. This system has been described in more detail previously [26]. The version we tested includes a feature model with 8 features and 5 integer feature attributes. These are implemented with a core ABS program and 8 deltas, totaling about 2000 lines of ABS code. The feature model has 16 products when feature attributes are omitted. If feature attributes are considered, the number of products is 5500.

The second case study is provided by the FormbaR project [19], an ongoing effort to build a comprehensive model of railway operations for Deutsche Bahn AG. The current version of this ABS model includes a feature model with 5 features, a core program and 5 deltas, totaling about 3300 lines of code. The feature model is comparatively simple and only has 5 products; each product has one feature and the associated variant is generated by applying a single delta to the base program. The source code is available from the FormbaR project website.[9]

[7] ABS toolchain available at `https://github.com/abstools/abstools`
[8] Choco Solver: `http://www.choco-solver.org/`
[9] FormbaR project: `http://formbar.raillab.de/`

5.1 Error Reporting. If the ABS compiler (`absc`) detects a product line declaration in a given ABS source project, it automatically performs the SPL unambiguity checking and the SPL partial type checking. These checks capture two kinds of errors, respectively (cf. Sect. 4.1). First, in case SPL unambiguity cannot be ruled out, an error similar to the following is reported:

```
replication.abs:1419:0:The product line RS is potentially ambiguous:
    Deltas JCD and CD both target class RS.RSMain, method
    getSchedules, but their application order is undefined.
productline RS;
^
```

Second, in case the generation of the signature of a variant fails, an error message, such as the following, points to the problem.

```
replication.abs:1396:4:Field iterator could not be found. When
    applying delta JCD on top of CD >> JD >> RD >> core, while
    building product {RS,Client,JSched}.
   removes Int iterator;
---^
```

5.2 Performance. We measured the performance of the SPL unambiguity and partial type checking mechanism implementation and compared it against the performance of generating and fully type checking all variants of the respective SPL in sequence. The results of the experiments are summarized in Fig. 6—we used a 2013 laptop machine (Intel core i7 CPU at 2 GHz, 16 GB RAM). We found that SPL partial checking had little noticeable impact on the performance of the ABS compiler. For the FAS Replication System SPL it took just 0.643 seconds, of which 0.61 s were spent computing all solutions to the constraint satisfaction problem (i.e., to enumerate all the products), and 0.033 s were spent checking unambiguity and generating the signatures of all the 16 variants.

By comparison, generating a single variant of the FAS Replication System SPL took on average slightly over 5 seconds (we timed only the process of transforming the E-AST by applying deltas and subsequently type checking the C-AST). For all 16 different products obtainable by disregarding attributes, this process took 82 seconds. Before implementation of the SPL Checking component, no tool infrastructure existed to automatically type check the entire product line. Moreover no option to exclude feature attributes from variant generation was provided. It would have required the developer to manually build and subsequently check all 5500 products, which takes around 8 hours.

For the FormbaR SPL, currently a quite simple product line, where each product has a single feature and only one delta is applied to generate the associated variant, it took 0.1 s to generate all 5 variants. If we include the time to solve the CSP, then the whole process took 0.64 s. This is slightly less efficient than employing the SPL partial checking mechanism, which took 0.566 s (including the time to solve the CSP).

5.3 Discussion. According to the numbers above the SPL partial checking mechanism exhibits a significant performance advantage (with respect to gen-

	FAS Replication System	FormbaR
Solve CSP	0.610	0.540
Check unambiguity & build variant PSTs	0.033	0.026
SPL partial checking (total)	**0.643**	**0.566**
Build one variant (avg.)	5.130	0.020
Build variants	82.080	0.100
Solve CSP & build variants (total)	**82.690**	**0.640**

Fig. 6. Case studies: performance (numbers in seconds)

erating all variants and fully type checking each of them in isolation) for all but very simple SPLs. Our evaluation also shows that the most performance-critical task of SPL partial checking is solving the CSP in order to enumerate all products. For a product line with hundreds or thousands of products, this task could potentially take too long to be practical. We plan to approach this problem from two angles. First, we see potential for optimization of the CSP, and will consider the alternative of SAT (which has shown promising performance according to related studies [24,14]), as this has not been a focus of our attention so far. Second, we will explore solving simplified CSPs corresponding to partially configured feature modes in case of too large solution spaces.

6 Related Work

SPL implementation approaches can be classified into three main categories [22]: *annotative approaches* expressing negative variability; *compositional approaches* expressing positive variability; and *transformational approaches* expressing both positive and negative variability.

DOP is a transformational approach. Notably, it is an extension of *Feature-Oriented Programming* (FOP, see [3] and [2, Sect. 6.1]), a compositional approach, where deltas are associated one-to-one with features and have limited expressive power: they can add and modify program elements, however, they cannot remove them. In annotative approaches all variants are included within the same model (often called a 150% model). A prominent example of SPL annotative implementation mechanism is represented by C preprocessor directives (`#define FEATURE` and `#ifdef FEATURE`).

We refer to [25] for a survey on SPL analyses (cf. the brief discussion at the beginning of Sect. 3). Here we discuss a type checking mechanism for FOP SPLs that has been implemented for the AHEAD Tool Suite [24], and a type checking mechanism for delta-oriented SPLs that we have recently proposed [10].

The AHEAD Tool Suite [3] supports FOP SPLs of Java programs. Thaker et al. [24] illustrates the implementation of a family-based approach for type checking AHEAD SPLs. The approach comprises: i) a family-feature-based step that computes for each class a stub (which contains a stub declaration for each field or method declaration that could appear in that class) and compiles each feature module in the context of all stubs; and ii) a family-based step that infers a

set of constraints that are combined with the feature model to generate a formula whose satisfiability implies that all variants can be successfully generated and compiled. The first step requires that all the field and method declarations that could appear in a class `C` in some variant must be "type compatible", e.g., for each field name `f` all declarations of field with name `f` that may appear in `C` in some variant must have the same type. The type checking approach presented in Sect. 3 do not suffer of this limitation. However, it involves an explicit iteration over the set of products, which becomes an issue when the number of products is huge (a product line with n features can have up to 2^n products). The approach by Thaker et al. [24] does not require an explicit iteration over the set of products, however it requires to check the validity of a propositional formula (which is a co-NP-complete problem). Thaker et al. [24,14] report that the performance of using SAT solvers to verify the propositional formulas (a SAT solver can be used to check whether a propositional formula is valid by checking whether its negation is unsatisfiable) for four non-trivial product lines was encouraging and that, for the largest product line, applying the approach was even faster than generating and compiling a single product. The empirical evaluation (cf. Sect. 5) of our partial implementation for ABS (cf. Sect. 4) of the feature-product-family approach for delta-oriented SPLs originally formalized for IFΔJ [5,12] (that is reformulated for FDABS in Sect. 3) exhibits a similar performance increase with respect to generating and checking each variant in isolation.

Delaware et al. [14] provide a formal foundation for the approach implemented by Thaker et al. [24]. They formalize a feature-family-based type checking approach for the Lightweight Feature Java (LFJ) calculus, which models FOP for the Lightweight Java (LJ) [23] calculus. The approach, which does not suffer of the "type compatible" limitation of the approach by Thaker et al. [24], comprises: i) a feature-based step that uses a constraint-based type system for LFJ to analyze each feature module in isolation and infer a set of constraints for each feature module; and ii) a family-based step where the feature model and the previously inferred constraints are used to generate a formula whose satisfiability implies that all variants are well-typed.

Recently, Damiani and Lienhardt [10] proposed a feature-family-based type checking approach that provides an extension to DOP of the two steps of the approach by Delaware et al. [14] together with a preliminary step, called partial typing, which provides early detection of some errors by analyzing each delta with respect to the class, field and method declarations occurring in the whole artifact base.[10] The approach has been designed to take advantage of automatically checkable DOP guidelines that make an SPL more comprehensible and type checking more efficient (see also [11]). Both the approach and the guidelines are formalized by means of IFΔJ.

[10] Partial typing guarantees that variants that can be generated and have their inner dependencies satisfied are well-typed, thus providing early detection of some errors—however, because it does not use feature model and configuration knowledge, it cannot guarantee that each variant can be generated and do not contain references to classes, fields or methods that are not defined in the variant.

7 Conclusion

We presented an extension of the ABS toolchain with an SPL unambiguity and type checking mechanism. Currently we have implemented only part of this mechanism (cf. Sect. 4.1). Our implementation is integrated in version 1.4.2 of the ABS toolchain, released in April 2017. We are currently working on fully implementing the SPL type checking mechanism and improving the efficiency of enumerating all the product of the SPL. Our evaluation, that used two industrial case studies, showed significant performance and usability advantages over an entirely product-based type checking approach that involves building all SPL variants. We found that 95% of the performance cost of the implemented SPL checking comes from solving the CSP necessary to enumerate all products. The actual unambiguity and partial type checking implementation is very efficient and takes only few milliseconds. This result raises our confidence that an implementation of the full SPL checking mechanism will remain similarly performant. We also plan to extend the SPL Checking component of the ABS toolchain by implementing the feature-family type checking mechanism and the automatically checkable DOP guidelines that we have recently proposed [10,11] (see Sect. 6). Our goal is to tame complexity of SPL type checking by tool support for DOP guidelines enforcement and orchestration of different type checking approaches.

Acknowledgments. We thank Eduard Kamburjan for help with the FormbaR case study, and the anonymous reviewers for comments and suggestions.

References

1. E. Albert, F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, and P. Y. H. Wong. Formal modeling and analysis of resource management for cloud architectures: an industrial case study using real-time ABS. *Service Oriented Computing and Applications*, 8(4):323–339, 2014.
2. S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.
3. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30:355–371, 2004.
4. D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *Proc. of CAiSE 2015*, volume 3520 of *LNCS*, pages 491–503. Springer, 2005. DOI: 10.1007/11431855_34.
5. L. Bettini, F. Damiani, and I. Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Informatica*, 50(2):77–122, 2013.
6. R. Bubel, A. Flores Montoya, and R. Hähnle. Analysis of executable software models. In *Executable Software Models: 14th Intl. School on Formal Methods*, volume 8483 of *LNCS*, pages 1–27. Springer, 2014.
7. D. Clarke, N. Diakov, R. Hähnle, E. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Wong. Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language. In *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *LNCS*, pages 417–457. Springer, 2011.
8. P. Clements and L. Northrop. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., 2001.

9. F. Damiani, R. Hähnle, E. Kamburjan, and M. Lienhardt. A unified and formal programming model for deltas and traits. In *Proc. of FASE 2017*, volume 10202 of *LNCS*, pages 424–441. Springer, 2017. DOI: 10.1007/978-3-662-54494-5_25.
10. F. Damiani and M. Lienhardt. On type checking delta-oriented product lines. In *Proc. of iFM 2016*, volume 9681 of *LNCS*, pages 47–62. Springer, 2016. DOI: 10.1007/978-3-319-33693-0_4.
11. F. Damiani and M. Lienhardt. Refactoring delta-oriented product lines to enforce guidelines for efficient type-checking. In *Proc. of ISoLA 2016*, volume 9953 of *LNCS*, pages 579–596, 2016. DOI: 10.1007/978-3-319-47169-3_45.
12. F. Damiani and I. Schaefer. Family-based analysis of type safety for delta-oriented software product lines. In *Proc. of ISoLA'12*, volume 7609 of *LNCS*, pages 193–207. Springer, 2012. DOI: 10.1007/978-3-642-34026-0_15.
13. F. de Boer, D. Clarke, M. Helvensteijn, R. Muschevici, J. Proença, and I. Schaefer. Final Report on Feature Selection and Integration, Mar. 2011. Deliverable 2.2b of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.
14. B. Delaware, W. R. Cook, and D. Batory. Fitting the pieces together: A machine-checked model of safe composition. In *Proc. of ESEC/FSE 2009*. ACM, 2009. DOI: 10.1145/1595696.1595733.
15. E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, Sept. 1960.
16. R. Hähnle. The Abstract Behavioral Specification Language: A Tutorial Introduction. In *Proc. of FMCO 2012*, volume 7866 of *LNCS*, pages 1–37. Springer, 2013. 10.1007/978-3-642-40615-7_1.
17. M. Helvensteijn, R. Muschevici, and P. Y. H. Wong. Delta modeling in practice: a Fredhopper case study. In *Proc. of VAMOS'12*, pages 139–148. ACM, 2012. DOI: 10.1145/2110147.2110163.
18. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
19. E. Kamburjan and R. Hähnle. Uniform modeling of railway operations. In *Proc. of FTSCS 2016*, volume 694 of *CCIS*, pages 55–71. Springer, 2017. DOI: 10.1007/978-3-319-53946-1_4.
20. R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating support for features in advanced modularization technologies. In *Proc. of ECOOP 2005*, volume 3586 of *LNCS*, pages 169–194. Springer, 2005. DOI: 10.1007/11531142_8.
21. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proc. of SPLC 2010*, volume 6287 of *LNCS*, pages 77–91. Springer, 2010. DOI: 10.1007/978-3-642-15579-6_6.
22. I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer*, 14(5):477–495, 2012.
23. R. Strniša, P. Sewell, and M. Parkinson. The Java module system: core design and semantic definition. In *Proc. of OOPSLA 2007*, pages 499–514. ACM, 2007.
24. S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proc. of GPCE '07*, pages 95–104. ACM, 2007. DOI: 10.1145/1289971.1289989.
25. T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 2014.
26. P. Y. 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. *Journal on Software Tools for Technology Transfer*, 14:567–588, 2012.