A unified and formal programming model for deltas and traits

A Unified and Formal Programming Model for Deltas and Traits*

Ferruccio Damiani[1], Reiner Hähnle[1,2], Eduard Kamburjan[2], and Michael Lienhardt[1]

[1] University of Torino
Torino, Italy
{ferruccio.damiani, michael.lienhardt}@unito.it
[2] Technical University Darmstadt
Darmstadt, Germany
{haehnle, kamburjan}@cs.tu-darmstadt.de

Abstract. This paper presents a unified model for two complementary approaches of code reuse: Traits and Delta-Oriented Programming (DOP). Traits are used to modularly construct classes, while DOP is a modular approach to construct Software Product Lines. In this paper, we identify the common structure of these two approaches, present a core calculus that combine Traits and DOP in a unified framework, provide an implementation for the ABS modelling language, and illustrate its application in an industrial modeling scenario.

1 Introduction

Systematic and successful code reuse in software construction remains a challenge and constitutes an important research problem in programming language design. The drive to digitalization, together with the fundamental changes of deployment platforms in recent years (cloud, multi-core), implies that modern software must be able to evolve and it must also support variability [33]. The standard reuse mechanism of mainstream object-oriented languages—class based inheritance—is insufficient to deal adequately with software evolution and reuse [14, 24] and provides no support for implementing software variability.

Traits are a mechanism for fine-grained reuse aimed at overcoming the limitations of class-based inheritance (see [9, 14, 25] for discussions and examples). Traits are sets of methods, defined independently of a class hierarchy, that can be composed in various ways to build other traits or classes. They were originally proposed and implemented in a SMALTALK-like, dynamically typed setting [14, 34]. Subsequently, various formulations of traits in a JAVA-like, statically typed setting were proposed [3, 6, 22, 26, 28, 29, 36].

* 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); project FormbaR (formbar.raillab.de), Innovationsallianz TU Darmstadt–Deutsche Bahn Netz AG.

Delta-oriented programming (DOP) [1, Sect. 6.6.1], [31] is a flexible and modular approach to implement Software Product Lines (SPL) [27]. Its core element is the *delta*, an explicit, structured construct for characterizing the difference between two program variants: DOP realizes SPL by associating deltas with product features (not necessarily one-to-one), which allows for a flexible and modular construction of program variants [32]. DOP is an extension of *Feature-Oriented Programming* (FOP) [1, Sect. 6.1], [2], a previously proposed approach to implement SPLs, where deltas are associated one-to-one with product features and have limited expressive power: like in DOP they can add and modify program elements (e.g., classes and attributes[3]), however, they cannot remove them. The explicit, flexible link between features and source code as realized in DOP is key to keep design-oriented and implementation-oriented views in sync. DOP was implemented on top of JAVA [21] and in the concurrent modeling language ABS [10], where it has been successfully used in industry [16, 18].

In this paper, we observe (and justify in Sect. 3.1) that, while deltas are ideal to realize *inter-product* reuse, they are unsuitable to achieve *intra-product* reuse. It is, therefore, natural to combine deltas and traits into a single language that ideally supports flexible inter- as well as flexible intra-product reuse. Moreover, we also observe (and justify in Sect. 3.2) that most delta and trait operations are nearly identical from an abstract point of view: they add attributes to an existing declaration, they remove attributes, and they modify existing attributes. It is, therefore, natural to unify the operations provided by deltas and traits. Such a unification simplifies the language and makes it easier to learn (it has a smaller number of concepts). Based on these observations, we design (in Sect. 3.3) a minimal language with a completely *uniform* integration of traits and deltas. Moreover, we implement our design in the concurrent modelling language ABS [10] and illustrate its application in an industrial modeling scenario. Indeed, ABS represents an ideal platform for our investigation as it supports DOP, but it so far lacks a construct for intra-product reuse: evaluations of ABS against industrial requirements repeatedly identified intra-product reuse mechanisms [15, 30] as an important factor to improve usability.

Paper organization. In Sect. 2 we introduce the FABS and FDABS languages which formalize a minimal fragment of core ABS [20] and its extension with deltas [10], respectively. In Sect. 3 we introduce the FTDABS language which formalizes our proposal by adding traits on top of FDABS. We explain and motivate our design decisions systematically with the help of an instructive example. In Sect. 4 we provide a formal semantics in the form of a rule system that eliminates traits and deltas by "flattening" [26]. In Sect. 5 we present the implementation of our ABS extension. In Sect. 6 we illustrate how it is applied in an industrial modeling scenario. In Sect. 7 we discuss related work and conclude.

[3] As usual in the OOP literature, with *attribute* we mean any declaration element, i.e., a field or a method, in contrast to the usage in the UML community, where attribute means only field.

P ::= $\overline{ID}\ \overline{CD}$
ID ::= `interface I [extends` $\overline{\texttt{I}}$`] {` $\overline{HD}$ `}`
CD ::= `class C [implements` $\overline{\texttt{I}}$`] {` $\overline{AD}$ `}`
AD ::= FD | MD

FD ::= `I f`
HD ::= `I m(`$\overline{\texttt{I x}}$`)`
MD ::= HD `{` $\overline{s}$ `return` e`; }`

Fig. 1. FABS syntax

```
interface IAccount {
  Int withdraw(Int amount);
}
class Account() implements IAccount {
  Int withdraw(Int amount) {
    if (balance-amount >= 0) { balance = balance-amount; }
    return balance;
  }
}
```

Fig. 2. Bank account example in FABS

2 FDABS: A Minimal Language for ABS with Deltas

We first present (in Sect. 2.1) FABS, a minimal fragment of core ABS [20], and then (in Sect. 2.2) recall—as previously shown for ABS in [10]—how deltas add the possibility to construct SPLs by factoring out the code that is common to different products in the SPL.

2.1 FABS: Featherweight ABS

The syntax of FABS is given by the grammar in Fig. 1. The non-terminal P represents programs, ID interface declarations, CD class declarations, AD attribute declarations, FD field declarations, HD header declarations, MD method declarations, e expressions, and s statements.[4] As usual, $\overline{X}$ denotes a finite sequence with zero or more occurrences of a syntax element of type X. Our development is independent of the exact expression and statement syntax, so we leave it unspecified. Our examples use standard operators and statements whose syntax and semantics is obvious.

The code snippet in Fig. 2 illustrates the FABS syntax. It is a fragment of an application that manages bank accounts. Withdrawals that would result in a negative balance are not carried out.

2.2 FDABS: Adding Deltas to FABS

Delta-Oriented Programming implements SPLs by adding three elements to the base language: a *feature model* which encodes the variability available in the SPL;

[4] ABS includes other features, like datatypes and concurrency, that we do not include in FABS as they are orthogonal to the delta and trait composition mechanisms.

$$
\begin{array}{lll}
L & ::= & \mathcal{M}\ \mathcal{K}\ \overline{\Delta}\ P \\
\Delta & ::= & \texttt{delta d}\ \{\overline{IO}\ \overline{CO}\ \} \\
IO & ::= & \texttt{adds}\ ID \mid \texttt{removes I} \mid \texttt{modifies I}\ [\texttt{extends}\ \overline{\texttt{I}}]\ \{\ \overline{HO}\ \} \\
HO & ::= & \texttt{adds}\ HD \mid \texttt{removes m} \\
CO & ::= & \texttt{adds}\ CD \mid \texttt{removes C} \mid \texttt{modifies C}\ [\texttt{implements}\ \overline{\texttt{I}}]\ \{\ \overline{AO}\ \} \\
AO & ::= & \texttt{adds}\ FD \mid \texttt{removes f} \mid \texttt{adds}\ MD \mid \texttt{removes m} \mid \texttt{modifies}\ MD
\end{array}
$$

Fig. 3. FDABS syntax: productions to be added to Fig. 1

a set of *deltas* that implement the variability expressed in the feature model; and *configuration knowledge* which links the feature model to the deltas.

The grammar resulting from the addition of DOP to FABS is shown in Fig. 3. An SPL L consists of a feature model $\mathcal{M}$, configuration knowledge $\mathcal{K}$, a (possibly empty) list of deltas $\overline{\Delta}$, and a (possibly empty or incomplete) base program P, defined as in Fig. 1. We leave the precise definition of the feature model and configuration knowledge unspecified, as it is not the focus of this work and invite the interested reader to look at [10] for a possible syntax for these elements. Deltas have a name `d` and a list 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 declaration of a method, with the `modifies` operation: in this operation, the new code can refer to a call of the original implementation of the method with the keyword `original`.

We illustrate this extension of FABS by declaring an SPL over the example in Fig. 2. We add two variants to the original code: one that enables interest payment, identified by the feature "Saving" and parameterized by the interest rate, and one that permits a negative balance, identified by the feature "Overdraft" and parameterized by the overdraft limit. A visual representation of the corresponding feature model is shown in Fig. 4.

We exemplify delta operations with delta `dOverdraft` shown in Fig. 5 which implements the feature "Overdraft". The parameter of "Overdraft" is encoded by the field `limit`.[5] Moreover, `dOverdraft` adds a setter method for that field, and it modifies `withdraw` to take the limit into account.

3 FTDABS: Adding Traits to FDABS

To motivate the design of the FTDABS language, we first demonstrate (in Sect. 3.1) that deltas cannot be used for intra-product code reuse. Then (in

[5] ABS uses *parameterized deltas* to manage feature parameters, which we do not include in our language to keep it as simple as possible.

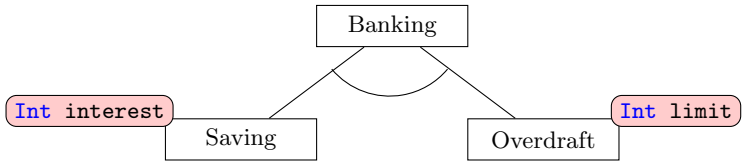


Fig. 4. Visual representation of the feature model of the bank account example

```
delta dOverdraft {
  modifies Account {
    adds Int limit;
    adds Unit setLimit(Int value) { limit = value; }
    modifies Int withdraw(Int amount) {
      if (balance-amount+limit >= 0) { balance = balance-amount; }
      return balance;
    }
  }
}
```

Fig. 5. The `dOverdraft` delta in FDABS

Sect. 3.2) we argue—as previously shown in [14]—that traits are a nice fit instead. Finally (in Sect. 3.3), we show how deltas and traits can be used in collaboration to smoothly integrate intra- as well as inter-product code reuse.

3.1 Motivating Traits

In the previous subsection we used deltas to construct three variants of the `Account` class: the base product, one with a feature that allows saving with interest, and one with a feature allowing overdraft. One can also imagine that a bank wants to have all these variants available at the same time to satisfy different client needs. The result would be three very similar classes `Account`, `AccountSaving` and `AccountOverdraft`: in this case, intra-product code reuse would be highly useful to avoid the duplication of the common parts of the three classes.

Deltas, however, cannot implement intra-product code reuse: by design, they associate each delta operation to one class, and it is thus impossible to use them to add the same code to different classes. Instead, traits, discussed in Sect. 3.2, are a well-known and very flexible approach for factoring out code shared by several classes. Moreover, in Sect. 3.3, we will illustrate our novel approach for combing deltas and traits, which exploits traits also for *inter-delta* code reuse (i.e, code reuse across different deltas and the base program).

3.2 FTABS: Adding Traits to FABS

Historically, traits and deltas were developed independently from each other in different communities (traits in the OOP community, deltas in the context of SPL). Accordingly, they are usually presented in quite different styles with

different notational conventions. Perhaps for these reasons, the surprisingly close analogies between traits and deltas have so far not been pointed out.

Traits are structurally simpler than deltas, because (i) they are declared independently of classes and interfaces and (ii) they satisfy the so called *flattening* principle [26], which states that each trait declaration just introduces a name for a set of methods and using a trait in a class declaration is the same as declaring the associated methods in the body of the class. Traits can be composed using operators[6] (where the first argument is always a trait) such as: (i) disjoint sum (taking as second argument a trait having method names disjoint with those of the first argument, resulting in a new trait that is their union); (ii) override (similar, but the methods in the second argument override those in the first); (iii) method exclusion (the second argument is a method name that is removed from the resulting trait); (iv) method alias (which duplicates a given method by supplying a new name).

The crucial observation is that these composition operators, except method aliasing, are present in the class-modify operation of deltas as well (where they have as implicit first argument the set of methods in the modified class): disjoint sum (with a singleton set of methods as second argument) corresponds to `adds`, overriding to `modifies` (without `original`),[7] method exclusion to `removes`.

We show in Fig. 6 our extension of FABS with traits. A program with traits *PT* is a finite number of trait declarations *TD* with an FABS program *P* using these traits. A trait is declared with the keyword `trait`, given a name `t`, and defined by a trait expression *TE*. A trait expression defines a set of methods by either declaring the methods directly, referencing other traits `t`, or applying a trait operation *TO* to a trait expression. The trait operations are the same as those of deltas, with the exception that `adds` and `modifies` manipulate a set of methods (described by a trait expression *TE*) instead of a single method. Moreover, our `modifies` trait operation is actually an extension of the trait override operation: each overriding method may contain occurrences of the keyword `original` to refer to the implementation of the overridden methods (in the same way as in deltas). In previous proposals of traits in a JAVA-like setting (see [3] for a brief overview) the attributes found in a trait (i.e., fields and methods accessed with `this` in method bodies) are listed in a separate declaration as requirements to classes that will use the trait. Here we adopt the convention from deltas to let requirements implicitly contain all undefined attributes invoked on `this`.

The last production in Fig. 6 overrides the production for attribute declarations in Fig. 1 by extending it with the possibility to import a trait into a class and thus make use of it. This latter extension is the *only* change that is necessary in the syntax of FABS classes for them to use traits.

In Fig. 7 we illustrate traits in FTABS with a new implementation of the `Account` class that uses a trait `tUpdate` that can be shared by classes `AccountSaving` and `AccountOverdraft`. The trait defines an `update` method which performs an

[6] We mention those proposed in the original formulation of traits [14].

[7] To the best of our knowledge, the `original` concept is not present in any formulation of traits in the literature. It can be encoded in traits with aliasing.

$$
\begin{array}{l}
PT ::= \overline{TD}\ P \\
TD ::= \texttt{trait t} = TE \\
TE ::= \{\ \overline{MD}\ \} \mid \texttt{t} \mid TE\ TO \\
TO ::= \texttt{adds}\ TE \mid \texttt{removes m} \mid \texttt{modifies}\ TE \\
AD ::= FD \mid MD \mid \texttt{uses}\ TE
\end{array}
$$

Fig. 6. FTABS syntax: productions to be added to Fig. 1—the last production overrides the production in the last line of Fig. 1 (the differences are highlighted in gray)

```
trait tUpdate = {
  Int update(Int amount) {
    // this assignment is in reality a complex database transaction:
    balance = balance+amount;
  }
}
class Account() implements IAccount {
  uses tUpdate
  Int withdraw(Int amount) {
    if (balance-amount >= 0) update(-amount);
    return balance;
  }
}
```

Fig. 7. The `tUpdate` trait and the refactored `Account` class in FTABS

unconditional update of the account's balance. The trait `tUpdate` is then re-used by the three different classes to define their `withdraw` method.

3.3 FTDABS: Combining Traits and Deltas

Our chosen style of declaration for traits makes it extremely simple to combine traits and deltas without the need to introduce further keywords and with merely one change in one syntax rule. The key observation is that the production rule for trait operations TO in Fig. 6 and the one for delta operations on methods in Fig. 3 (final three slots in rule for AO) are identical, with the small exception that the `adds` and `modifies` trait operations work on a set of methods (described by a trait expression TE) instead of a single method. Hence, we can unify trait and delta operations by simply replacing delta operations on methods by trait operations. We present the full grammar of the resulting language in Fig. 8.

The desired effect of extending attribute operations to include trait operations is that we can now use traits and trait operations for the declaration of *deltas*, thereby supporting also intra- and inter-*delta* code reuse. It is worth to observe that trait declarations are not part of the base program, i.e., traits are not provided by the base language (the language in which each variant is written)—therefore, deltas are not able to modify trait declarations and `uses` clauses in classes. Our design decision is to provide traits as a construct to enabling code

$$
\begin{array}{lll}
L & ::= \mathcal{M}\ \mathcal{K}\ \overline{TD}\ \overline{\Delta}\ P & AD ::= FD \mid MD \mid \texttt{uses}\ TE \\
P & ::= \overline{ID}\ \overline{CD} & FD ::= \texttt{I f} \\
ID & ::= \texttt{interface I [extends } \overline{\texttt{I}}\texttt{]} \{\ \overline{HD}\ \} & HD ::= \texttt{I m(}\overline{\texttt{I x}}\texttt{)} \\
CD & ::= \texttt{class C [implements } \overline{\texttt{I}}\texttt{]} \{\ \overline{AD}\ \} & MD ::= HD \{\ \overline{s}\ \texttt{return}\ e;\ \}
\end{array}
$$

$$
\begin{array}{lll}
TD & ::= \texttt{trait t} = TE \\
TE & ::= \{\ \overline{MD}\ \} \mid \texttt{t} \mid TE\ TO \\
TO & ::= \texttt{adds}\ TE \mid \texttt{removes m} \mid \texttt{modifies}\ TE
\end{array}
$$

$$
\begin{array}{lll}
\Delta & ::= \texttt{delta d} \{\overline{IO}\ \overline{CO}\ \} \\
IO & ::= \texttt{adds}\ ID \mid \texttt{removes I} \mid \texttt{modifies I [extends } \overline{\texttt{I}}\texttt{]} \{\ \overline{HO}\ \} \\
HO & ::= \texttt{adds}\ HD \mid \texttt{removes m} \\
CO & ::= \texttt{adds}\ CD \mid \texttt{removes C} \mid \texttt{modifies C [implements } \overline{\texttt{I}}\texttt{]} \{\ \overline{AO}\ \} \\
AO & ::= \texttt{adds}\ FD \mid \texttt{removes f} \mid TO
\end{array}
$$

Fig. 8. FTDABS syntax (differences to FDABS syntax in Fig. 3 are highlighted)

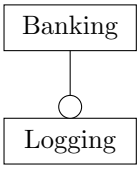
```
delta dOverdraft {
  modifies Account {
    adds Int limit;
    adds Unit setLimit(Int value) { limit = value; }
    modifies Int withdraw(Int amount) {
      if (balance-amount+limit >= 0) update(-amount);
      return balance;
    }
  }
}
```

Fig. 9. Refactored `dOverdraft` delta

reuse in FDABS in the base program *as well as* in deltas. The alternative design choice of adding deltas to a base language that provides traits [13] is briefly discussed below in Sect. 7.

We illustrate the capabilities of the FTDABS language with the refactored `dOverdraft` delta in Fig. 9. Observe that `dOverdraft` does not have to add the trait `tUpdate`, because it was already used in the base product as shown in Fig 7. We achieved the maximal possible degree of reuse, because the method header of `withdraw` and the changed guard in its body must be repeated in any case.

The capability to use traits inside deltas is a powerful tool to describe cross-cutting feature implementations in a succinct manner. Assume we want to add a logging feature as illustrated in the feature diagram on the right (ABS permits multi-feature diagrams, i.e., orthogonal feature hierarchies). To implement logging we create a delta that adds a suitable method call to `update`. Since the latter is defined as a trait, we can use trait composition. First we declare a trait `tUpdateLog` that uses trait `tUpdate`, adds a logger and suitably modifies the original `update` method, see Fig. 10. Please



```
trait tUpdateLog = tUpdate
  adds { Unit log(Int value) { ... } } // logging facility
  modifies { Int update(Int amount) {
                original(amount);
                log(amount);
              }
           }
delta dLogging {
  modifies Account {
    removes update
    adds tUpdateLog
  }
}
```

Fig. 10. Using traits inside deltas

observe that the `original` keyword (which, to the best of our knowledge, is not present in other formulation of traits) can be used in the same manner within traits as within deltas to refer to the most recent implementation. The delta that realizes logging now simply removes the old version of the obsolete `update` method and adds the new trait. This has to be done for each class, where the new trait is to be used, but that is intentional: for example, logging might not be desired to take place in each call of `update` throughout the whole product. This is in line with the general design philosophy of ABS-based languages that code changes should be specified extensionally (in contrast to aspect-oriented programming, for example) to facilitate code comprehension and analysis.

4 Semantics

We present the formal semantics of the FTDABS language. The *artifact base* AB of an SPL consists of its traits, deltas and base program. Given a specific product to generate, the semantics eliminates from the artifact base all traits and deltas to produce an FABS program corresponding to the specified product (in particular, first eliminating all traits produces an FDABS program). For simplicity, we suppose in our presentation that all the deltas that do not take part in the generation of the chosen product have already been removed from the artifact base and that all the remaining deltas have been sorted following the partial order in the configuration knowledge $\mathcal{K}$. This initial step is standard in DOP [4] and allows us to focus on the semantics of traits and delta operations.

4.1 Semantics of Traits

We structure the semantics of traits, shown in Fig. 11, into two rule sets. The first set formalizes the intuitive semantics of trait operations. This semantics uses the *name* function which retrieves the name of a method. We extend that notation and, given a sequence of field and method declarations $\overline{AD}$, also use

$$\text{T:Adds}\quad \frac{name(\overline{MD}) \cap name(\overline{MD}') = \emptyset}{\{\ \overline{MD}\ \}\ \texttt{adds}\ \{\ \overline{MD}'\ \} \triangleright \{\ \overline{MD}\ \overline{MD}'\ \}} \qquad \text{T:Rems}\quad \frac{name(MD) = \texttt{m}}{\{\ MD\ \overline{MD}\ \}\ \texttt{removes}\ \texttt{m} \triangleright \{\ \overline{MD}\ \}}$$

$$\text{T:Mods}\quad \frac{\forall 1 \leq i \leq n,\ (MD_i = \texttt{I}\ \texttt{m}_i(\overline{\texttt{I}\ \texttt{x}})\ \{\ \texttt{return}\ e_i;\ \}) \wedge (name(MD'_i) = \texttt{m}_i)}{\begin{array}{c}\{\ MD_1\ \ldots\ MD_n\ \overline{MD}\ \}\ \texttt{modifies}\ \{\ MD'_1\ \ldots\ MD'_n\ \} \\ \triangleright \{\ MD'_1[^{e_1}/_{\texttt{original}(\overline{\texttt{x}})}]\ \ldots\ MD'_n[^{e_n}/_{\texttt{original}(\overline{\texttt{x}})}]\ \overline{MD}\ \}\end{array}}$$

$$\text{T:Trait}\quad (\texttt{trait}\ \texttt{t} = TE\ AB) \triangleright AB[^{TE}/_{\texttt{t}}] \qquad \text{T:Class}\quad \frac{name(\overline{AD}) \cap name(\overline{MD}) = \emptyset}{\begin{array}{l}\texttt{class}\ \texttt{C}\ \texttt{implements}\ \overline{\texttt{I}}\ \{\ \overline{AD}\ \ \texttt{uses}\ \{\overline{MD}\}\ \} \\ \quad \triangleright\ \texttt{class}\ \texttt{C}\ \texttt{implements}\ \overline{\texttt{I}}\ \{\ \overline{AD}\ \overline{MD}\ \}\end{array}}$$

Fig. 11. Semantics of traits

$name(\overline{AD})$ to obtain the names of the fields and methods declared in $\overline{AD}$. Rule (T:Adds) states that the `adds` operation combines two sets of methods that have no name in common. Rule (T:Rems) removes a method only if it exists in the given set of methods. Finally, rule (T:Mods) implements the modification of a set of methods. It replaces existing methods MD_i with new implementations MD'_i. The latter, however, may refer to the most recent implementation with references to `original` which our semantics inlines.

The second set of rules enforces the flattening principle (cf. Sect. 3.2). Rule (T:Trait) eliminates a trait declaration from a program by replacing occurrences of its name by its definition. Finally, rule (T:Class) is applicable after the traits operations inside a class have been eliminated and puts the resulting set of method declarations inside the body of the class, provided that there is no name clash.

4.2 Semantics of Deltas

Due to the large number of operations a delta may contain, we split the set or reduction rules in three parts. The first part, in Fig. 12, presents the simplest elements of the semantics of deltas, which applies in sequence all the operations contained in a delta. Rule (D:Empty) is applicable when a delta does not contain any operation to execute: the delta is simply deleted. Rules (D:Inter)/(D:Class) extract the first interface/class operation from the delta and apply it to the full artifact base (denoted $AB \bullet IO$ / $AB \bullet CO$).

In case when the interface operation is the addition of an interface (rule (D:AddsI)), the specified interface is added to the artifact base AB, provided that it was not already declared. In case the interface operation is the removal of an interface `I`, the interface with that name is extracted from the artifact base and deleted (rule (D:RemsI)). The addition and removal of classes is similar.

The rules for modifying interfaces and classes are shown in Figs. 13 & 14. The structure of these rules is similar to the ones for deltas, in the sense that they apply in order all the operations contained in the modification. Rule (D:I:Empty)

D:Empty
$$\texttt{delta d \{ \} } AB \triangleright AB$$

D:Inter
$$\frac{AB = (\texttt{delta d \{ } IO\ \overline{IO}\ \overline{CO}\ \texttt{\} } AB')}{AB \triangleright (\texttt{delta d \{ } \overline{IO}\ \overline{CO}\ \texttt{\} } AB') \bullet IO}$$

D:Class
$$\frac{AB = (\texttt{delta d \{ } CO\ \overline{CO}\ \texttt{\} } AB')}{AB \triangleright (\texttt{delta d \{ } \overline{CO}\ \texttt{\} } AB') \bullet CO}$$

D:AddsI
$$\frac{name(ID) \notin name(AB)}{AB \bullet (\texttt{adds } ID) \triangleright ID\ AB}$$

D:RemsI
$$\frac{name(ID) = \texttt{I}}{(ID\ AB) \bullet (\texttt{removes I}) \triangleright AB}$$

D:AddsC
$$\frac{name(CD) \notin name(AB)}{AB \bullet (\texttt{adds } CD) \triangleright CD\ AB}$$

D:RemsC
$$\frac{name(CD) = \texttt{C}}{(CD\ AB) \bullet (\texttt{removes C}) \triangleright AB}$$

Fig. 12. Semantics of deltas: top-level

D:I:Empty
$$(\texttt{interface I extends } \overline{\texttt{I}}\ \texttt{\{ } \overline{HD}\ \texttt{\} } AB) \bullet (\texttt{modifies I \{ \}}) \triangleright \texttt{interface I extends } \overline{\texttt{I}}\ \texttt{\{ } \overline{HD}\ \texttt{\} } AB$$

D:I:Adds
$$\frac{name(HD) \notin name(\overline{HD})}{(\texttt{interface I extends } \overline{\texttt{I}}\ \texttt{\{ } \overline{HD}\ \texttt{\} } AB) \bullet (\texttt{modifies I \{ (adds } HD) \overline{HO}\ \texttt{\}}) \triangleright (\texttt{interface I extends } \overline{\texttt{I}}\ \texttt{\{ } HD\ \overline{HD}\ \texttt{\} } AB) \bullet (\texttt{modifies I \{ } \overline{HO}\ \texttt{\}})}$$

D:I:Rems
$$\frac{name(HD) = \texttt{m}}{(\texttt{interface I extends } \overline{\texttt{I}}\ \texttt{\{ } HD\ \overline{HD}\ \texttt{\} } AB) \bullet (\texttt{modifies I \{ (removes m) } \overline{HO}\ \texttt{\}}) \triangleright (\texttt{interface I extends } \overline{\texttt{I}}\ \texttt{\{ } \overline{HD}\ \texttt{\} } AB) \bullet (\texttt{modifies I \{ } \overline{HO}\ \texttt{\}})}$$

D:I:Extends
$$(\texttt{interface I extends } \overline{\texttt{I}}\ \texttt{\{ } \overline{HD}\ \texttt{\} } AB) \bullet (\texttt{modifies I extends } \overline{\texttt{I}'}\ \texttt{\{ } \overline{HO}\ \texttt{\}}) \triangleright (\texttt{interface I extends } \overline{\texttt{I}'}\ \texttt{\{ } \overline{HD}\ \texttt{\} } AB) \bullet (\texttt{modifies I \{ } \overline{HO}\ \texttt{\}})$$

Fig. 13. Semantics of deltas: interface modification

is applicable when no further modification is requested on the given interface, so that the result is the interface itself. Rule (D:I:Adds) adds the specified method header to the interface (provided that no header with this name is already present in the interface). Rule (D:I:Rems) removes an existing method header from the interface. Finally, rule (D:I:Extends) is applicable when a modification of the `extends` clause is requested, in which case that clause is entirely replaced with the set specified in the modification.

The rules for class modification in Fig. 14 are very similar to the ones for interfaces, with two exceptions: first, manipulation (`adds` and `removes`) of method headers is replaced by manipulation of fields (rules (D:C:AddsF) and (D:C:RemsF)); second, class operations also include trait operations to modify their method set. Rule (D:C:Trait) applies a trait operation contained in a delta to the given class simply by applying it to its set of methods.

D:C:Empty
$$(\texttt{class C implements } \overline{\texttt{I}} \ \{ \ \overline{AD} \ \} \ AB) \bullet (\texttt{modifies C } \{ \ \}) \triangleright \texttt{class C implements } \overline{\texttt{I}} \ \{ \ \overline{AD} \ \} \ AB$$

D:C:AddsF
$$\frac{name(FD) \notin name(\overline{AD})}{(\texttt{class C implements } \overline{\texttt{I}} \ \{ \ \overline{AD} \ \} \ AB) \bullet (\texttt{modifies C } \{ \ (\texttt{adds } FD) \ \overline{AO} \ \}) \triangleright (\texttt{class C implements } \overline{\texttt{I}} \ \{ \ FD \ \overline{AD} \ \} \ AB) \bullet (\texttt{modifies C } \{ \ \overline{AO} \ \})}$$

D:C:RemsF
$$\frac{name(FD) = \texttt{f}}{(\texttt{class C implements } \overline{\texttt{I}} \ \{ \ FD \ \overline{AD} \ \} \ AB) \bullet (\texttt{modifies C } \{ \ (\texttt{removes f}) \ \overline{AO} \ \}) \triangleright (\texttt{class C implements } \overline{\texttt{I}} \ \{ \ \overline{AD} \ \} \ AB) \bullet (\texttt{modifies C } \{ \ \overline{AO} \ \})}$$

D:C:Trait
$$(\texttt{class C implements } \overline{\texttt{I}} \ \{ \ \overline{FD} \ \overline{MD} \ \} \ AB) \bullet (\texttt{modifies C } \{ \ TO \ \overline{AO} \ \}) \triangleright (\texttt{class C implements } \overline{\texttt{I}} \ \{ \ \overline{FD} \ (\texttt{adds } \{ \ \overline{MD} \ \} \ TO) \ \} \ AB) \bullet (\texttt{modifies C } \{ \ \overline{AO} \ \})$$

D:C:Extends
$$(\texttt{class C implements } \overline{\texttt{I}} \ \{ \ \overline{AD} \ \} \ AB) \bullet (\texttt{modifies C implements } \overline{\texttt{I}'} \ \{ \ \overline{AO} \ \}) \triangleright (\texttt{class C implements } \overline{\texttt{I}'} \ \{ \ \overline{AD} \ \} \ AB) \bullet (\texttt{modifies C } \{ \ \overline{AO} \ \})$$

Fig. 14. Semantics of deltas: class modification

5 Integration into the ABS Tool Chain

We implemented our approach as a part of the ABS compiler tool chain, as illustrated in Fig. 15. The tool chain is structured as a pipeline of three components. The *parser* takes as its input an ABS program composed of a set of ABS files, and produces an extended Abstract Syntax Tree (AST) corresponding to that program. The *rewriter* is the component responsible for generating the variant corresponding to the selected features. This is done by applying in order the various deltas required by the selected features: the result is a *core* AST which does not contain any deltas. The core AST can then be analyzed by different tools developed for the ABS language [8]. It can also be executed using one of the ABS code generation backends [7, 19, 35].

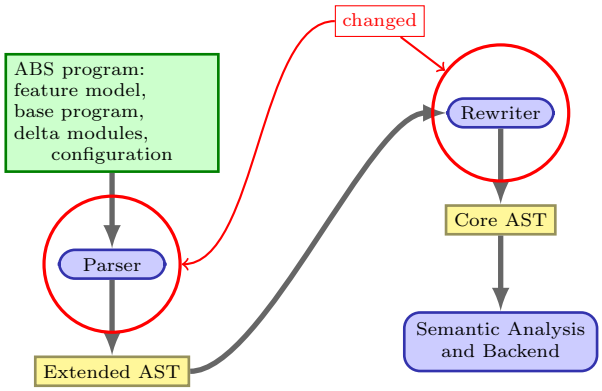


Fig. 15. Structure of the ABS compiler tool chain

The integration of traits and deltas, motivated and discussed in the previous sections, was implemented in the ABS compiler tool chain by modifying the parser and the rewriter components. We stress that, unlike in the previous

sections, the implementation is based not merely on FABS, but on the complete core ABS language [20]. The parser was extended with the new syntax for traits, and with the new elements in the delta and class syntax (deltas may use trait operations, classes can use traits). The changes in the rewriter mostly concern the semantics of deltas that now include trait operations. Moreover, the extended rewriter eliminates traits in classes and deltas (as in Sect. 4.1): the rewriter now first eliminates all the traits declared in a program, then it applies the activated deltas to generate a core AST as before.

The trait extension of ABS is designed to be syntactically and semantically conservative, i.e., it is backward compatible: legacy ABS code produces the same result as before with the extended parser and rewriter. Moreover, as the rewriter still generates a core AST, none of the existing analysis tools and code generation backends for ABS need to be updated. They can correctly analyse and execute any ABS program with traits.

Our implementation performs some checks on the input program to make sure that traits and deltas are well-formed. First, it ensures that any call to `original` is performed inside a `modifies` trait operation. Second, it controls the validity of the `removes` operation, i.e., an error is raised if "`removes m`" is performed on a set of methods or a class that does not contain `m`. Third, it controls that traits do not contain circular definitions. The remaining well-formedness checks for traits and deltas are delegated to the ABS type system. For instance, if a method is added twice, or if a method is called that is not part of the class after all deltas and traits have been applied, then standard ABS type checking will detect that during semantic analysis of the generated core AST.

6 Using Deltas and Traits in an Industrial Case Study

We adapted the FormbaR [16] case study modelling railway operations[8] to use traits. Among other aspects, FormbaR models components and rules of operation of a railway infrastructure in a precise and comprehensive manner. It is to date the largest ABS model with currently ca. 2600 LoC. Deltas contribute 830 LoC and are used to model different scenarios for simulation.

Due to the large number of different track elements and model ranges for these infrastructure elements, deltas are used for *variability management* [17]: Deltas are able to describe different types of track components which then can be added to a scenario. Traits, on the other hand, are used to encapsulate aspects or behavior of track elements shared by the core railway operations model. The following trait, for example, encapsulates that a track element transmits no information to the rear end of a train:

```
trait NoRear =
 { Info triggerRear(TrainId train, Edge e){ return NoInfo; } }
```

We use traits in two situations: as ABS does not have class-based inheritance, we declare at least one trait for each interface that is implemented multiple times

[8] The model is available under `formbar.raillab.de`

```
trait Sig = {
  [Atomic] Unit setSignal(Signal sig){ this.s = sig; }
  SignalState getState() { return this.state; }
  Unit setState (SignalState nState, Time t){ this.state = nState;}
}
interface TrackElement { ... }
interface MainSignal extends TrackElement { ... }
class MainSignalImpl implements MainSignal {
  uses Sig;
  ...
}
```

Fig. 16. Usage of traits in the railway case study

```
delta RandomDefect;
  modifies class TrackElements.MainSignalImpl {
    modifies Unit setState(SignalState nState, Time t) {
      if (random(100) > 95) this.s!defect(t);
      original(nState, t);
  }
}
```

Fig. 17. Usage of a delta with implicit trait in the railway case study

and use it in the implementing classes. The trait in Fig. 16 is used in three different classes (only one of which is shown) that implement different components of a signal: the methods declared in the trait encapsulate the inner signal state. The `NoRear` trait is used in a different scenario: It does not accompany an interface and, hence, is not used in all classes implementing an interface, but only in a subset that shares behavior, but is not distinguished by type.

In one variant of the railway SPL it is modeled that a signal shows a defect with a certain probaility after the main signal is set. The delta in Fig. 17 modifies only one of the classes where trait `Sig` is used and merely describes the additional behavior, while calling the original method afterwards.

The case study is an SPL with 7 features, 7 deltas and 9 products. Table 1 gives statistics on the current usage of traits in the railway study. The ABS model uses five traits with one to three methods each (only one of the traits is currently an extension of another trait and requires trait operations). In the module with 12 classes describing track elements, where 4 of the traits are used,

Trait	# methods in trait	# classes where used
Sig	3	3
NoSig	1	5
Block	2	2
NoRear	1	10
NoFront	2	3

Table 1. Usage statistics of traits in the FormbaR model

the number of LoC shrinks from 262 to 203 (-22%). Using traits appears to be natural and straightforward for the modeler. It makes the railway model considerably easier to read and to maintain.

7 Related Work and Conclusions

We proposed a combination of traits and deltas in a uniform language framework that stresses the similarities between the composition operators. We have formalized it by means of the minimal language FTDABS, implemented it in the full language as part of the ABS tool chain, and illustrated its applicabilty in an industrial modeling scenario. The resulting language is a *conservative extension* (syntactically and semantically) of the ABS language.

The commonality between delta and trait operations had not been formally worked out and put to use in the literature so far. Relevant papers on deltas and traits have already been discussed above. In addition we mention some work on using traits to implement software variability and on adding deltas on top of trait-based languages.

Bettini et al. [5] propose to use traits to implement software variability. It is noted in [13] that this way to model variability is less structured that traits, i.e., traits are less suitable than deltas for the purpose. Lopez-Herrejon et al. [23] evaluate five technologies for implementing software variability, including the "trait" construct of Scala, which is in fact a mixin (see [14] for a detailed discussion about the differences between trait and mixins).

Damiani et al. [13] address the problem of defining deltas on top of *pure trait-based languages* (languages, where class inheritance is *replaced* by trait composition). The proposal does not permit using traits for inter-delta code reuse. Moreover, it does not exploit the commonalities between deltas and traits. Therefore, it results in a quite complex language, containing a disjoint union of (the operations provided by) deltas and traits. No formal description of the semantics and no implementation are provided.

The flattening semantics given in Sect. 4 allows to smoothly integrate traits into the ABS tool chain (cf. Sect. 5). We plan to improve the integration further by a type checking phase that identifies type errors before flattening traits (building on existing type checking approaches for deltas [4, 12, 11] and traits [5]).

In our proposal traits are not assumed to be part of the base language: they extend DOP by enabling code reuse in the base program as well as in deltas (see Sect. 3.3). Our proposal to extend DOP with traits can be straightforwardly added on top of languages that, like Java, support class-based inheritance: the flattening principle [26] provides a straightforward semantics for using traits in combination with class-based inheritance. In future work we would like to extend DOP for Java along these lines and evaluate, by means of case studies, its benefits with respect to the current implementation of DOP for Java [21, 37].

Acknowledgments. We thank the anonymous reviewers for comments and suggestions for improving the presentation.

References

1. S. Apel, D. S. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.
2. D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, 30(6):355–371, 2004.
3. L. Bettini and F. Damiani. Generic traits for the Java platform. In *PPPJ*, pages 5–16. ACM, 2014.
4. L. Bettini, F. Damiani, and I. Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Informatica*, 50(2):77–122, 2013.
5. L. Bettini, F. Damiani, and I. Schaefer. Implementing type-safe software product lines using parametric traits. *Sci. of Comp. Progr.*, 97, part 3:282–308, 2015.
6. L. Bettini, F. Damiani, I. Schaefer, and F. Strocco. TraitRecordJ: A programming language with traits and records. *Science of Computer Programming*, 78(5):521 – 541, 2013.
7. N. Bezirgiannis and F. de Boer. ABS: a high-level modeling language for cloud-aware programming. In *SOFSEM*, pages 433–444. Springer, 2016.
8. R. Bubel, A. Flores Montoya, and R. Hähnle. Analysis of executable software models. In *Executable Software Models: 14th Intl. School on Formal Methods for the Design of Computer, Communication, and Software Systems*, volume 8483 of *LNCS*, pages 1–27. Springer, June 2014.
9. D. Cassou, S. Ducasse, and R. Wuyts. Traits at work: The design of a new trait-based stream library. *Comput. Lang. Syst. Struct.*, 35(1):2–20, 2009.
10. 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 *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *LNCS*, pages 417–457. Springer, 2011.
11. F. Damiani and M. Lienhardt. On type checking delta-oriented product lines. In *iFM*, volume 9681 of *LNCS*, pages 47–62. Springer, 2016.
12. F. Damiani and I. Schaefer. Family-based analysis of type safety for delta-oriented software product lines. In *ISOLA*, volume 7609 of *LNCS*, pages 193–207. Springer, 2012.
13. F. Damiani, I. Schaefer, S. Schuster, and T. Winkelmann. Delta-trait programming of software product lines. In *ISOLA*, volume 8802 of *LNCS*, pages 289–303. Springer, 2014.
14. S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006.
15. Evaluation of Core Framework, Aug. 2010. Deliverable 5.2 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.
16. R. Hähnle and E. Kamburjan. Uniform modeling of railway operations. In *Proc. Fifth Intl. Workshop on Formal Techniques for Safety-Critical Systems (FTSCS)*, CCIS. Springer, 2016.
17. R. Hähnle and R. Muschevici. Towards incremental validation of railway systems. In *ISOLA*, volume 9953 of *LNCS*, pages 433–446. Springer, Oct. 2016.
18. M. Helvensteijn, R. Muschevici, and P. Wong. Delta Modeling in Practice, a Fredhopper Case Study. In *6th Intl. Workshop on Variability Modelling of Software-intensive Systems, Leipzig, Germany*. ACM, 2012.
19. L. Henrio and J. Rochas. From modelling to systematic deployment of distributed active objects. In *Coordination Models and Languages, 18th IFIP WG 6.1 Intl. Conf., COORDINATION*, volume 9686 of *LNCS*, pages 208–226. Springer, 2016.

20. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: a core language for abstract behavioral specification. In *Formal Methods for Components and Objects: 9th Intl. Symp., FMCO. Revised Papers*, pages 142–164. Springer, 2012.
21. J. Koscielny, S. Holthusen, I. Schaefer, S. Schulze, L. Bettini, and F. Damiani. DeltaJ 1.5: Delta-oriented programming for Java 1.5. In *PPPJ*, pages 63–74. ACM, 2014.
22. L. Liquori and A. Spiwack. Feathertrait: A modest extension of featherweight java. *ACM Trans. Program. Lang. Syst.*, 30(2):11:1–11:32, 2008.
23. R. E. Lopez-Herrejon, D. S. Batory, and W. R. Cook. Evaluating support for features in advanced modularization technologies. In *ECOOP*, volume 3586 of *LNCS*, pages 169–194. Springer, 2005.
24. L. Mikhajlov and E. Sekerinski. A Study of the Fragile Base Class Problem. In *ECOOP*, volume 1445 of *LNCS*, pages 355–383. Springer, 1998.
25. E. R. Murphy-Hill, P. J. Quitslund, and A. P. Black. Removing duplication from java.io: a case study using traits. In *Companion 20th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 282–291. ACM, 2005.
26. O. Nierstrasz, S. Ducasse, and N. Schärli. Flattening traits. *J. of Object Technology*, 5(4):129–148, 2006.
27. K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.
28. P. J. Quitslund, R. Murphy-Hill, and A. P. Black. Supporting Java traits in Eclipse. In *OOPSLA workshop on Eclipse Technology eXchange, ETX*, pages 37–41. ACM, 2004.
29. J. Reppy and A. Turon. Metaprogramming with traits. In *ECOOP*, volume 4609 of *LNCS*, pages 373–398. Springer, 2007.
30. Resource-aware Modeling of the ENG Case Study, July 2015. Deliverable 4.4.2 of project FP7-610582 (Envisage), available at `http://www.envisage-project.eu`.
31. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond, SPLC*, volume 6287 of *LNCS*, pages 77–91, 2010.
32. I. Schaefer and F. Damiani. Pure Delta-oriented Programming. In *Proc. 2nd Intl. Workshop Feature-Oriented Software Development, FOSD*, pages 49–56. ACM, 2010.
33. 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. *J. on Software Tools for Technology Transfer*, 14(5):477–495, 2012.
34. N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *ECOOP*, pages 248–274. Springer, 2003.
35. V. Serbanescu, K. Azadbakht, F. S. de Boer, C. Nagarajagowda, and B. Nobakht. A design pattern for optimizations in data intensive applications using ABS and Java 8. *Concurrency & Computation: Practice & Experience*, 28(2):374–385, 2016.
36. C. Smith and S. Drossopoulou. *Chai*: Traits for Java-like languages. In *ECOOP*, volume 3586 of *LNCS*, pages 453–478. Springer, 2005.
37. T. Winkelmann, J. Koscielny, C. Seidl, S. Schuster, F. Damiani, and I. Schaefer. Parametric DeltaJ 1.5: propagating feature attributes into implementation artifacts. In *Workshops Software Engineering*, volume 1559 of *CEUR Workshop Proc.*, pages 40–54. CEUR-WS.org, 2016.