Softw Syst Model (2017) 16:1223–1251
DOI 10.1007/s10270-016-0516-2

REGULAR PAPER

A novel model-based testing approach for software product lines

Ferruccio Damiani[1] · **David Faitelson**[2] · **Christoph Gladisch**[3] · **Shmuel Tyszberowicz**[4]

Received: 15 November 2013 / Revised: 22 December 2015 / Accepted: 9 January 2016 / Published online: 13 February 2016

Abstract Model-based testing relies on a model of the system under test. FineFit is a framework for model-based testing of Java programs. In the FineFit approach, the model is expressed by a set of tables based on Parnas tables. A software product line is a family of programs (the products) with well-defined commonalities and variabilities that are developed by (re)using common artifacts. In this paper, we address the issue of using the FineFit approach to support the development of correct software product lines. We specify a software product line as a specification product line where each product is a FineFit specification of the corresponding software product. The main challenge is to concisely specify the software product line while retaining the readability of the specification of a single system. To address this, we used delta-oriented programming, a recently proposed flexible approach for implementing software product lines, and developed: (1) delta tables as a means to apply the delta-oriented programming idea to the specification of software product lines; and (2) DeltaFineFit as a novel model-based testing approach for software product lines.

Keywords Java · Alloy · Software product line · Delta-oriented programming · Model-based testing · Refinement

Communicated by Profs. Einar Broch Johnsen and Luigia Petre.

The authors of this paper are listed in alphabetical order. This work has been partially supported by project HyVar (www.hyvar-project.eu), which has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 644298; by ICT COST Action IC1402 ARVI (www.cost-arvi.eu); by ICT COST Action IC1201 BETTY (www.behavioural-types.eu); by Italian MIUR PRIN 2010LHT4KM Project CINA (sysma.imtlucca.it/cina); by Ateneo/CSP D16D15000360005 project RunVar; and by GIF (Grant No. 1131-9.6/2011).

✉ Ferruccio Damiani
ferruccio.damiani@unito.it

David Faitelson
davidf@afeka.ac.il

Christoph Gladisch
gladisch@ira.uka.de

Shmuel Tyszberowicz
tyshbe@tau.ac.il

[1] Dipartimento di informatica, University of Torino, C.so Svizzera 185, 10149 Turin, Italy

[2] Afeka Tel Aviv Academic College of Engineering, Tel Aviv, Israel

[3] Karlsruhe Institute of Technology, Karlsruhe, Germany

[4] The Academic College of Tel Aviv Yaffo, Tel Aviv, Israel

1 Introduction

In this paper, we propose a novel model-based testing approach for *software product lines* (SPLs). An SPL is a set of programs (the products) that share significant common functionality and have well-understood and organized variabilities [1,2]. Our idea is to integrate data-refinement-based testing into SPL development.

FineFit [3] is an approach for model-based testing of Java programs which relies on the notion of *data refinement* [4] to compare the state of the model with the state of the *system under test* (SUT). Data refinement captures the relationship between an abstract model and its concrete representation. During the testing process, FineFit must both retrieve abstractions of the SUT current state and instruct the SUT to perform operations with specific input. These tasks are supported by two Java code fragments—the *retrieve function* and the *driver*, respectively. The retrieve function and the driver are written by FineFit's users. A prototypical implementation of the FineFit tool is available [5].

In the FineFit approach, the structure of the model and the specification of the system's operations are given by a set of tables, based on Parnas tables [6]. Parnas tables organize expressions, where rows and columns separate an expression into cases, and each table entry specifies either the result value for some case or a condition that partially identifies some case. The strength of the tabular notation is its clear readability which helps in reducing errors in specifications.

In this paper, we address the problem of using the FineFit approach to support the development of correct SPLs. Writing for each product to be tested, a FineFit specification, a retrieve function, and a driver would be error prone and not cost-effective. Our idea is to express the *specification of an SPL* as a product line, where each product is a FineFit model (i.e., a set of tables) for the corresponding product (i.e., a Java program) of the specified SPL.

The main challenge addressed in this paper is to devise a means to concisely specify the SPL being tested while retaining the readability of a FineFit specification of a single system. To this aim, we consider *delta-oriented programming* (DOP) [7,8] (see also [9], Section 6.6.1), a recently proposed flexible approach for implementing SPLs. DOP is an extension of *feature-oriented programming* (FOP) [10], a prominent approach for developing SPLs (cf. [9], Section 6.1).[1] DeltaJ [7,12,13] is the archetypal language for delta-oriented programming of SPLs of Java programs.

As pointed out in [9], "much of the tremendous power of features is yet to be unlocked by making features explicit throughout the entire systems and software lifecycle." In this paper, we present DeltaFineFit, a novel model-based testing approach for delta-oriented SPLs. DeltaFineFit integrates data-refinement-based testing into delta-oriented SPL development by ensuring that each product is generated together with its FineFit model and the suitable driver and retrieve functions.

DeltaFineFit enables the fully automated testing of all the products of an SPL. When the number of products is too large, testing all the products is unfeasible. This could be addressed by using, e.g., *sample-based SPL testing* techniques [14–17], where a subset of products—covering relevant combinations of features—is generated and tested by applying single-system testing techniques.

The main contribution of this paper is to introduce the *delta table* concept: an approach for deriving a new table from an existing one, based on the difference (the delta) between the tables. Delta tables are used to define a notion of "delta module for FineFit specifications" (i.e., a construct that describes how to modify the FineFit specification of a product to obtain the FineFit specification of another product), that we call *delta-table module*. Delta tables are illustrated and evaluated (in terms of their support to conciseness and readability of SPL specifications) by considering a small SPL as a case study that is used as a running example throughout the paper.

A prototypical implementation of DeltaTables, a tool that generates a FineFit specification by applying a delta-table module to a FineFit specification, is available at the DeltaFineFit home page [18]. All the tables presented in the paper that are the result of delta-table application have been automatically generated. The DeltaFineFit tool chain (which will provide fully automated support to the integrated use of DeltaTables, DeltaJ, and FineFit) is currently under development.

The remainder of the paper is organized as follows. Section 2 recalls the FineFit approach for specifying and testing single Java programs. Section 3 recalls DOP and illustrates, by using DeltaJ, the SPL example that is used through the paper. Section 4 introduces delta tables. Section 5 illustrates DeltaFineFit by means of the SPL introduced in Sect. 3. Section 6 evaluates the DeltaFineFit approach. Section 7 reviews some related work. We conclude in Sect. 8 that also describes some future directions. "Appendix 1" elaborates on the structure and the semantics of the FineFit tables. "Appendix 2" illustrates the algorithm that describes the semantics of delta tables.

A preliminary version of the material presented in this paper is briefly outlined in [19]. Here we present a new and a more flexible notion of delta tables, describe implementation details, and provide detailed explanations and examples.

2 A recollection of data-refinement testing of Java programs with FineFit

The data-refinement theory [4] captures the relationship between an abstract model and its concrete implementation. The state spaces of the two levels of abstraction are related by a *retrieve function* which maps the concrete representation into the abstract one. When we use data refinement for testing, the abstract model becomes a test oracle and a source of test cases for the concrete program.

The FineFit [3,5] model-based (or model-driven) testing framework uses data refinement to directly compare the state of the model (the specification) with the state of the SUT.

FineFit helps to understand the testing results and to trace the reason for any difference between the specification and the SUT (see [3]). To test a system, the user writes its specification as a collection of HTML[2] tables (where cells contain Alloy expressions [20]). The specification of the SUT con-

[1] A straightforward embedding of FOP into DOP is illustrated, e.g., in [11].

[2] In practice, the tables can be written using any tool that can export its output to HTML, for example MS Word. FineFit ignores anything that is not part of an HTML table.

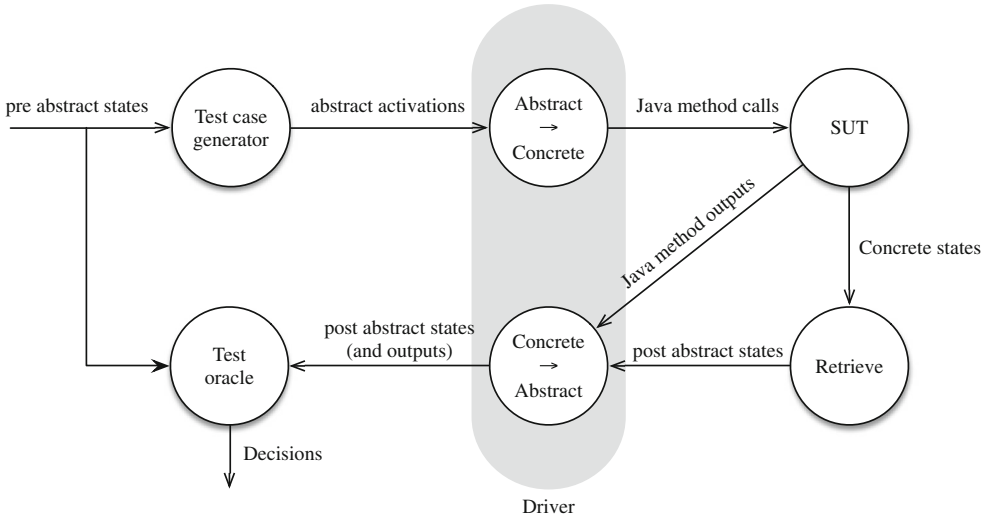


Fig. 1 FineFit testing procedure. Each abstract state is a Kodkod Instance object that maps the names of the state variables to relations; each relation is a set of tuples of atoms. Each abstract activation consists of an operation name and atoms for the input parameters of the operation. The driver is explained in Sect. 2.3

sists of tables for its basic types (called *atoms*), for the states of the SUT, for its invariants, and one table for each operation provided by the SUT. Each operation table defines an operation as a predicate on the system states immediately before and after the operation (similar to how operations are specified in languages like Z [21]). FineFit uses these predicates in two ways. First, it uses the predicates to test the behavior of the SUT's operations. For each operation X, it applies the corresponding predicate to the SUT's state, right after the SUT completes operation X. A false result indicates a discrepancy between the expected and the actual behaviors. Second, it uses the predicates to generate test cases by "solving" each predicate (a solution is an assignment of values that satisfy the predicate to the state variables and the inputs) and uses the solutions as test cases. These two capabilities are implemented with the help of Alloy's[3] Kodkod [22] relational model finder. We present an example in the rest of this section; for more details, see [3].

The SUT's retrieve function is a Java method that translates the concrete state (of the SUT) into an instance of the abstract state. The task of implementing the retrieve function is delegated to the developer—who must know how the data structures implement the system's specification. This makes the testing framework flexible and scalable, since the programmers can control the parts (i.e., select a subset) of the system to expose for the purpose of testing, even if the system is too large or complicated for automatic analysis.

[3] Alloy is a general purpose modeling language (in the style of Z) for reasoning about relational structures with first order logic. It has no direct concept of system states or operations, and it does not offer any tool for testing software.

Figure 1 illustrates the testing process of a SUT when FineFit is used. Before the actual testing process is executed, FineFit checks the consistency of the model. Then, FineFit begins the testing process by arbitrarily choosing an operation and input available in the current state, applying the corresponding concrete operation to the SUT, and checking whether the new state corresponds to the operation's specification. This process continues until either a discrepancy is found or until the user decides to stop the process. During testing, FineFit prints a trace that consists of the abstract snapshots (states) and the operation calls of the SUT. When a problem is detected, the user can review the entire history that led to it.

In the following subsections, we demonstrate data-refinement testing with FineFit by an example of a simple Java program implementing a photo album, that we call the Base Album. We begin by describing in Sect. 2.1 the program we would like to test. Then, we describe in Sect. 2.2 the FineFit model that is used to test the program. Finally, in Sect. 2.3, we present the interfaces between FineFit and the program. These interfaces are used to retrieve the current program state for analysis in FineFit and to allow FineFit to apply operations on the program.

2.1 The Java program to be tested

Listing 1 shows the Java code implementing the Base Album. Informally, the Base Album manages a sequence of photos and provides operations for adding and viewing photos. It consists of the class Photo, the interface PhotoAlbum (with the nested classes PhotoExists and AlbumIsFull), and the class ArrayPhotoAlbum which implements the interface,

```
1  public class Photo {
2    private String image;  // represents the bitmap image
3    public Photo(String theImage) {
4      if(theImage == null) throw new
5                          IllegalArgumentException("NullImage");
6      image = theImage;
7    }
8    public String getImage() { return image; }
9    public String toString() { return image; }
10 }
```

```
1  import java.util.Set;
2  public interface PhotoAlbum {
3    Photo addPhoto(String image);
4    Set<Photo> viewPhotos();
5    public class PhotoExists extends RuntimeException { }
6    public class AlbumIsFull extends RuntimeException { }
7  }
```

```
1  import java.util.Set;
2  import java.util.HashSet;
3  public class ArrayPhotoAlbum implements PhotoAlbum {
4    private int size = 0;
5    private Photo [] photoAt;
6    public ArrayPhotoAlbum(int maxSize) {
7      if(maxSize<1) throw new
8                          IllegalArgumentException("IllegalSize");
9      photoAt = new Photo[maxSize];
10   }
11   public boolean imageIsInAlbum(String image) {
12     for (int i= 0; i < size; i++) {
13       Photo p = photoAt[i];
14       if (p.getImage().equals(image)) return true;
15     }
16     return false;
17   }
18   public Photo addPhoto(String image) {
19     if(image == null) throw new
20                          IllegalArgumentException("NullImage");
21     if(size == photoAt.length) throw new AlbumIsFull();
22     if(imageIsInAlbum(image)) throw new PhotoExists();
23     Photo new_photo = new Photo(image);
24     photoAt[size] = new_photo;
25     size = size + 1;
26     return new_photo;
27   }
28   public Set<Photo> viewPhotos() {
29     Set<Photo> result = new HashSet<Photo>();
30     for (int i= 0; i < size; i++) { result.add(photoAt[i]); }
31     return result;
32   }
33 }
```

Listing 1: Java 1.5 code implementing the Base Album

providing functionality for adding a photo to the album and viewing its photos (the methods addPhoto and viewPhotos of PhotoAlbum, respectively).

2.2 The FineFit model

In order to test a system with FineFit, the developer has to write a model that specifies the desired behavior of the system. A FineFit model consists of relational variables and invariant predicates that represent the system states and of predicates that describe the effects of the operations on the system states. Both the structure (i.e., the state variables and the invariant predicates) of the model and its operations predicates are written in a tabular form. Formally, a FineFit *model* (or *specification*) consists of:

1. a *constants table* that defines the values of constants used in the model;
2. an *atoms table*, listing the name and the *scope* (i.e, the maximum number of instances) of each of the *atoms* (i.e., the basic types) used in the model;
3. a number of *enumeration tables* listing the values of each enumerated type, where each value has a unique symbolic name;
4. a *state table*, listing the name and the type of the variables representing the state of the model;
5. an *invariants table*, listing the invariants that the state of the model must satisfy when the model is created and that must be preserved by any operation that may be executed on the model;
6. a number of *operation tables* representing the operations of the model.

We name the first five kinds of tables *structural tables* (since they represent the structure of the model). The syntax of the tables of the model is illustrated in Fig. 2. The model of the Base Album is given in Fig. 3. The expressions used in the tables are written in Alloy [20], except that we use variables decorated with "?" and "!" to denote inputs and outputs, respectively. In most cases, the expressions are self-explanatory; when needed we provide a short explanation.

2.2.1 Structural tables

We now describe the structural tables of the Base Album example.

Constant table The constant table contains one constant, `MAX`, that specifies the maximal number of photographs in the album.

Atoms (basic types) table The atoms used in the model of Base Album are `Photo`, `Int`, and `Report`. `Report` is an enumeration type that we use in conjunction with an output variable to indicate the success or the failure of an operation (its values are specified by an enumeration table).

The atoms table associates a scope with each atom to determine its maximal number of instances used during testing. In the example, we test the system with at most five photos. Note that the scope is not part of the specification; it is only used to limit the amount of entities in the test cases. The scope of the `Report` type can be safely set to three, because `Report` has exactly three instances—one for each constant in the enumeration table.

Enumeration table The Base Album has an enumeration table for `Report`.

It defines `Report` to be a set of exactly three instances whose names are given in the table.

Fig. 2 Model table syntax

Constants table syntax

Constant name	Value
$Name_1$	$Value_1$
⋮	⋮
$Name_{n_C}$	$Value_{n_C}$

Atoms table syntax

Atom	Scope
$Atom_1$	$Number_1$
⋮	⋮
$Atom_{n_A}$	$Number_{n_A}$

Enumeration table syntax

Atom
$Value_1$
⋮
$Value_{n_E}$

State table syntax

State variable	Type
var_1	$Type_1$
⋮	⋮
var_{n_V}	$Type_{n_V}$

Invariants table syntax

Invariant name	Invariant
$Name_1$	$AlloyFormula_1$
⋮	⋮
$Name_{n_I}$	$AlloyFormula_{n_I}$

Operation table syntax

$\texttt{id}(\overline{\texttt{x?:Type}}, \overline{\texttt{y!:Type}})$	$C_1^{\{1..\}}$	…	$C_1^{\{..h..\}}$	…	$C_1^{\{..m\}}$
	…	…	…	…	…
	$C_p^{\{1\}}$	…	$C_p^{\{h\}}$	…	$C_p^{\{m\}}$
$varOrOut_1$	t_1^1	…	t_1^h	…	t_1^m
⋮	⋮	…	⋮	…	⋮
$varOrOut_n$	t_n^1	…	t_n^h	…	t_n^m

Fig. 3 Specification of the Base Album. Note that the expression photoAt.add[p?] is equivalent to the expression photoAt+(#photoAt→ p?); that is, it describes the union of two relations. The content of the cells in the tables is always free of side effects; there are no imperative statements in the FineFit language

Constants table

Constant name	Value
MAX	5

Atoms table

Atom	Scope
Photo	5
Int	5
Report	3

Enumeration table

Report
ALBUM_FULL
PHOTO_EXISTS
OK

State table

State variable	Type
photoAt	seq Photo

Invariants table

Invariant name	Invariant
Disjoint	#photoAt.elems = #photoAt
BoundedSize	#photoAt ≤ MAX

Operation tables

init()	true
photoAt	none -> none

addPhoto(p?:Photo, report!:Report)	#photoAt < MAX		#photoAt ≥ MAX
	p? !in ran[photoAt]	p? in ran[photoAt]	true
photoAt	photoAt.add[p?]	=	=
report!	OK	PHOTO_EXISTS	ALBUM_FULL

viewPhotos(result!: seq Photo)	true
result!	photoAt.elems

State table The state table describes the state of the Base Album by capturing, via the variable `photoAt`, the sequence of photos in the album.

Invariants table Two invariants are needed for the Base Album: (1) All the album photos are distinct (i.e., each photo has been added to the album only once), and (2) there are no more than `MAX` photos in the album. The first invariant is specified using the number of photos in the album (recall that the album is a sequence of photos). This number has to be equal to the size of the set of all photos in the album.[4]

2.2.2 Operation tables

A FineFit operation table is a predicate that specifies the behavior of an operation as a relation between the model's state variables before the operation starts (the pre-state) and after the operation completes (the post-state). It consists of two major areas: a *precondition tree* which consists of *condition cells* (first p rows in the operation table syntax at the bottom of Fig. 2) and an *expression table* which consists of variable and value cells (last n rows in the operation table syntax). The precondition tree consists of predicates that determine which columns to use in the definition of the post-state. The expression table is a set of columns, where each column defines the values of the state variables in the post-state (given their values in the pre-state) and the values of the output parameters. For example, the `addPhoto` operation of the Base Album (cf. Fig. 3) updates the state of the `photoAt` state variable and the `report!` output parameter (first column). This operation has three expression columns—columns two to four in the table. The first expression column (the second column in the table) is for the case where the input photo `p?` (of type `Photo`) does not appear in the album and the album is not full, then the photo is added to `photoAt`. The next expression column is for the case where the input photo already appears in the non-full album (`photoAt` does not change), and the last expression column is for the case where the album is full (`photoAt` does not change). The value of `report!` (of type `Report`) is OK only in the case that the `addPhoto` operation indeed added a new photo to the album. Note that predicates that appear on top of each other are conjoined, while predicates that appear side by side represent disjunction. If we consider the precondition part of an operation table (see the operation table syntax, Fig. 2) as a matrix of p rows and m columns, then

- each of the predicates $C_p^{\{1\}}..C_p^{\{m\}}$ in row p occupies exactly one cell, and
- each of the predicates $C_q^{\{1..\}}..C_q^{\{..h..\}}..C_q^{\{..m\}}$ in row q (with $1 \leq q \leq p$) spans over one or more consecutive cells (a predicate that spans over columns $j..k$ is superscripted by the set $\{j..k\}$ of the indexes of those columns).

We provide further explanations about the structure and semantics of operation tables in "Appendix 1."

In addition to `addPhoto`, the model of the Base Album provides the operations `viewPhotos` and `init`. The operation `viewPhotos` uses `elems` to return (using `result!`) the set of elements in a given sequence (in our case the `photoAt` sequence). Finally, the operation `init`—which corresponds a class constructor in Java—defines the initial state of the album, demanding that there are no photos in the album.

The meaning of "=" in a cell of the expression table part of an operation table (e.g., `addPhoto` in Fig. 3) is that the content of the state variable associated with the table row is left unchanged.

2.3 The retrieve function and the driver

During the testing process, the SUT must provide FineFit with abstractions of its current state, and FineFit must instruct the SUT to perform operations with specific inputs. The first task is supported by a *retrieve function* and the second by a *driver*.[5]

The retrieve function relates the concrete state of the SUT (Java code) to the abstract state of the specification. To provide a retrieve function, the SUT must implement a method that returns an abstract snapshot (a collection of named sets of tuples, one for each state variable) of its current state. For example, in our Base Album application, we have added the retrieve method illustrated in Listing 2 to the class ArrayPhotoAlbum (given in Listing 1). The abstract state is constructed using the State class that is part of the FineFit library. The class IdMap manages the mapping between the concrete Java objects and their corresponding abstract atoms.

A driver is needed to connect the SUT with FineFit. It is responsible for translating the abstract operation calls and data used by FineFit to the concrete representation of the SUT, and for translating concrete return values and exceptions back to their corresponding abstract atoms. To facilitate the implementation of the driver, we provide (as a part of the FineFit library) the base class FineFitDriver. It contains two maps, one (represented by the field ops) for translating abstract operation names to concrete operations and one (represented by the field exceptions) for translating concrete exceptions to abstract error codes. Any concrete driver class, like the PhotoAlbumDriver class for the Base Album

[4] Note that by definition a set does not contain duplicate elements.

[5] In the original paper about FineFit [3], we used the term *fixture*, which was borrowed from Fit [23], instead of *driver*. We think that driver is a more appropriate term.

```
import com.finefit.sut.IdMap;
import com.finefit.sut.State;
```

```
public State retrieve() {
    State state = new State();
    state.add_state("photoAt", 2);
    for (int i = 0; i < size; i++) {
        state.get_state("photoAt").add("" + i,
                                IdMap.instance().obj2atom(photoAt[i]));
    }
    return state;
}
```

Listing 2: Required import statements and method implementing the retrieve function for the Base Album specification (to be inserted in the class ArrayPhotoAlbum in Listing 1)

```
import java.util.Set;
import com.finefit.sut.*;
import com.finefit.sut.FineFitDriver;

public class PhotoAlbumDriver extends FineFitDriver {
    private ArrayPhotoAlbum sut;
    protected void setup_sut() { sut = new ArrayPhotoAlbum(5); }
    public void init_sut(com.finefit.model.State args) { }
    public State retrieve() { return sut.retrieve(); }
    protected void setup_operation_table() {
        ops.put("addPhoto", new Operation() {
            public void apply(com.finefit.model.State args,
                                        State outputs) throws Exception {
                String id = args.getArg("p");
                Photo p = sut.addPhoto(id);
                IdMap.instance().associate(p, id);
            } });
        ops.put("viewPhotos", new Operation() {
            public void apply(com.finefit.model.State args,
                                        State outputs) throws Exception {
                outputs.add_output("result!", 1);
                Set<Photo> photos = sut.viewPhotos();
                for(Photo p : photos) {
                    outputs.get_output("result!")
                                        .add(IdMap.instance().obj2atom(p));
                }
            } });
    }
    protected void setup_exception_table() {
        exceptions.put("PhotoAlbum$PhotoExists",
                                                "PHOTO_EXISTS");
        exceptions.put("PhotoAlbum$AlbumIsFull",
                                                "ALBUM_FULL");
    }
}
```

Listing 3: Java code implementing the driver for the Base Album

application in Listing 3, has to populate these two maps with the exceptions and the operations of the SUT. This is done by defining the methods set_up_operation_table and set_up_exception_table, respectively. The concrete driver class has also to define the methods set_up_sut (for creating the SUT instance), set_init_sut (for performing further initialization operations that might be needed), and retrieve (for calling the retrieve method on the SUT). The driver unpacks the operation names and arguments coming from FineFit, calls the operation, captures the exceptions thrown by the operation, converts them into error codes, and returns (by calling the SUT's retrieve method) to FineFit the state of the SUT after the operation has completed.

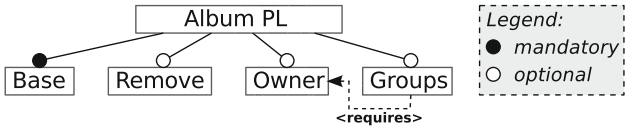


Fig. 4 Feature model of the Album PL

3 A recollection of delta-oriented programming of SPLs of Java programs with DeltaJ

DeltaJ is the archetypal language for delta-oriented programming of SPLs of Java programs [7]. A prototypical implementation of DeltaJ that supports Java 1.5, called DeltaJ 1.5, is available [12,13]. In this section, we briefly illustrate the use of DeltaJ 1.5 in the implementation of a simple product line—the Album PL. Each product in the Album PL is a Java 1.5 program implementing a photo album. This running example aims at presenting the proposed approach for integrating refinement-based testing into delta-oriented SPLs development rather than at providing a realistic case study.

In delta-oriented programming, each product is described by a set of features, where a feature is an abstract description of functionality [24]. Figure 4 depicts the feature model[6] of the Album PL as a feature diagram. The feature Base is mandatory, while all other features are optional. A product that has the feature Groups requires also the feature Owner. The Album PL has therefore six products. The product that has only the Base feature is the Base Album introduced in Sect. 2.

A DeltaJ product line consists of a *code base* and a *product-line declaration*. The code base (described in Sect. 3.1) consists of a set of delta modules describing modification to Java programs. The product-line declaration (described in Sect. 3.2) provides the connection of the delta modules with the product features; i.e., it specifies which delta modules must be used to generate each product.

3.1 DeltaJ delta modules

A DeltaJ delta module describes the changes in a given product (i.e., a Java program) that are needed to implement other products (i.e., other Java programs). The alterations inside a delta module act both at the class/interface level by

- adding or removing a Java *compilation unit*, that is, an interface or a class together with a package declaration and a list of import statements, or
- modifying the package declarations, or
- modifying the import statements;

[6] A *feature model* defines the valid feature configurations of an SPL, i.e., the feature configurations that describe the products (see, e.g., [24]).

```
1  SPL Album {
2    Features = {Base, Remove, Owner, Groups}
3    Deltas = {DBase, DRemove, DOwner, DRemoveAndOwner, DGroups}
4    Constraints { Base & (Groups => Owner); }
5    Partitions {
6      {DBase} when (Base);
7      {DRemove} when (Remove), {DOwner} when (Owner);
8      {DRemoveAndOwner} when (Remove & Owner);
9      {DGroups} when (Groups);
10   }
11   Products {
12     A_Base = {Base};
13     A_Remove = {Base, Remove};
14     A_Owner = {Base, Owner};
15     A_RemoveOwner = {Base, Remove, Owner};
16     A_OwnerGroups = {Base, Owner, Groups};
17     A_OwnerRemoveGroups = {Base, Remove, Owner, Groups};
18   }
19 }
```

Listing 4: DeltaJ 1.5 product-line declaration for the Album PL

and at the class/interface structure level by

- modifying existing interfaces (i.e., changing the super interfaces and adding, removing, or modifying method signatures or nested types), or
- modifying the internal structure of existing classes (i.e., changing the super class or the implemented interfaces and adding, removing, or modifying constructors, fields, methods, or nested types).

Modifying a method m of a class C means replacing the method body with a new one. The new body may contain the call **original**(···) that is replaced in the generated product by a call to a new method with a fresh name m', which has the same type and body as m before the modification. The new method m' is added to the class C when the product is generated.[7]

3.2 DeltaJ product-line declaration

Listing 4 illustrates the declaration for the Album PL. The product-line declaration:

- Lists the product features.
- Lists the delta modules.
- Describes the set of valid feature configurations by means of propositional constraints over the set of features (see e.g., [24]). For each feature φ in the set $\{\overline{\varphi}\}$ of the product-line features, we introduce a propositional variable with the same name. A propositional formula P characterizes a set of feature configurations $\Psi \subseteq \mathscr{P}(\{\overline{\varphi}\})$ if and only if, for every feature configuration $\{\overline{\psi}\} \in \Psi$, the formula P is true when the variables in $\{\overline{\psi}\}$ are true and the variables in $\{\overline{\varphi}\} \setminus \{\overline{\psi}\}$ are false. The propositional formula in Listing 4 (Base & (Groups ⇒ Owner)) represents the six valid feature configurations described by the feature diagram (Fig. 4).
- Describes the relation between delta modules and features by means of a totally ordered set of constraints called *partitions*. A partition consists of a set of *when-clauses*. When-clauses in the same partition are separated by a comma, and the end of each partition is indicated by a semicolon. Consider $p \geq 1$ partitions, such that the i-th partition contains $q_i \geq 1$ delta clauses $S_{i,j}$ when $\mathtt{P}_{i,j}$ ($1 \leq i \leq p$ and $1 \leq j \leq q_i$). Each when-clause consists of a set of delta module names $S_{i,j}$, followed by the **when** keyword and by a propositional formula over features $\mathtt{P}_{i,j}$. The sets of delta module names $S_{i,j}$ are pairwise disjoint, and their union consists of all the delta modules of the SPL (i.e., the set of sets of delta module names $\{S_{i,j} \mid 1 \leq i \leq p \text{ and } 1 \leq j \leq q_i\}$ represents a partition of the delta module set of the SPL).[8] The formula $\mathtt{P}_{i,j}$ (called the *application condition* of the delta modules in $S_{i,j}$) describes for which feature configurations the delta modules must be applied. Only valid feature configurations (according to the feature model) may be used for product generation—hence, the application conditions must be read by assuming that the formula describing the set of valid feature configurations holds. Delta modules in the same partition can be applied in any order, whereas the order between partitions must be respected. The ordering allows the designer to enforce semantic requires-relations that are necessary for the applicability of the delta modules.
- Declares some products that can be generated by giving a name to the associated feature configurations (this allows developers to disable the generation of some products without changing the set of valid feature configurations).

According to the Album PL declaration in Listing 4: The features and the valid feature configurations are those described by the feature model in Fig. 4. The delta modules DBase, DRemove, DOwner, and DGroups are associated with the features Base, Remove, Owner, and Groups, respectively; moreover, when the two features Remove and Owner have to be realized, also the delta module DRemoveAndOwner must be applied. All the products corresponding to the six valid feature configurations can be generated.

The delta module DBase (Listing 5) introduces the class Photo, the interface PhotoAlbum, and the class ArrayPhotoAlbum representing the product which has only the Base

[7] This mechanism is similar to the **Super**(…) call of FOP [10] and to the *around advice* and *proceed* mechanisms of *aspect-oriented programming* (AOP)—see, e.g., [7,25] for a comparison between DOP and AOP.

[8] In DELTAJ 1.5, each constraint "$S_{i,1}$ when $\mathtt{P}_{i,1}$, …, S_{i,q_i} when $\mathtt{P}_{i,q_i}$;" is called "partition," since the set of sets of delta module names $\{S_{i,j} \mid 1 \leq j \leq q_i\}$ is a partition of $\cup_{1 \leq j \leq q_i} S_{i,j}$.

```
delta DBase {
   adds { package it.unito.Album;  /* Same as in Listing 1 (top)  */}
   adds { package it.unito.Album;  /* Same as in Listing 1 (middle)*/}
   adds { package it.unito.Album;  /* Same as in Listing 1 (bottom)*/}
}
```

Listing 5: Code base of the Album PL: delta module DBase

```
delta DRemove {

   modifies it.unito.Album.PhotoAlbum {
      adds public void removePhoto(int location);
   }

   modifies it.unito.Album.ArrayPhotoAlbum {
      adds public void removePhoto(int location) {
         if ((location < 0) || (size <= location))
            throw new IllegalArgumentException("IllegalLocation");
         photoAt[location] = photoAt[size−1];
         photoAt[size−1] = null;
         size = size −1;
      }
   }

}
```

Listing 6: Code base of the Album PL: delta module DRemove

```
delta DOwner {

   adds { package it.unito.Album;
      class User {
         ... /* Fields, constructor, methods */
      }
   }

   modifies it.unito.Album.PhotoAlbum {
      adds public void login(String name, String password);
      adds public void logout();
      adds nested { public class AlreadyLogged extends
                                                RuntimeException {} }
      adds nested { public class AuthFailed extends
                                                RuntimeException {} }
      adds nested { public class NotAuthorized extends
                                                RuntimeException {} }
      adds nested { public class OwnerNotLoggedIn extends
                                                RuntimeException {} }
   }

   modifies it.unito.Album.ArrayPhotoAlbum {

      ... /* Adds imports, adds fields, modifies constructor, adds methods */

      modifies addPhoto(String image) {
         if (!isOwnerLoggedIn()) throw new OwnerNotLoggedIn();
         return original(image);
      }
   }

}
```

Listing 7: Code base of the Album PL: delta module DOwner

feature (cf. Sect. 2.1). The delta module DRemove (Listing 6) introduces the Remove feature which enables to remove a photo from the album. The delta module DOwner (Listing 7) introduces the Owner feature which protects the photo album by enabling only the owner (who has to login) to modify the album. The delta module DRemoveAndOwner (Listing 8) introduces the code needed to handle the combi-

```
delta DRemoveAndOwner {

   modifies it.unito.Album.ArrayPhotoAlbum {
      modifies removePhoto(int location) {
         if (!isOwnerLoggedIn()) throw new OwnerNotLoggedIn();
         original(location);
      }
   }

}
```

Listing 8: Code base of the Album PL: delta module DRemoveAndOwner

```
delta DOwner {

   adds { package it.unito.Album;
         /* Imports */
      class Groups {
         ... /* Fields, constructor, methods */
      }
   }

   modifies it.unito.Album.Photo {
      ... /*Adds a field (group)  and two methods (getGroup
                                                and setGroup)  */
   }

   modifies it.unito.Album.PhotoAlbum {
      adds public User updateUser(String name, String password);
      adds public Group updateGroup(String name,
                                    Set<String> memberNames);
      adds public void removeUser(String name);
      adds public void removeGroup(String name);
      adds public void updatePhotoGroup(int location,
                                    String groupName);
      adds nested { public class MissingUser extends
                                    RuntimeException {} }
      adds nested { public class MissingUsers extends
                                    RuntimeException {} }
      adds nested { public class MissingGroup extends
                                    RuntimeException {} }
      adds nested { public class RemoveOwnerGroup extends
                                    RuntimeException {} }
      adds nested { public class RemoveOwner extends
                                    RuntimeException {} }
   }

   modifies it.unito.Album.ArrayPhotoAlbum {
         ... /* Adds fields, modifies constructor, modifies methods,
                                                adds methods */
   }

}
```

Listing 9: Code base of the Album PL: delta module DGroups

nation of the optional features DRemove and DOwner. The delta module DGroups (Listing 9) introduces a further level of protection by requiring the owner to create users and to associate with each photo the group of users that can view it. The complete code of the delta modules of the Album PL is available at the DeltaFineFit home page [18].

3.3 DeltaJ generation of the products

A product is *valid* if it corresponds to a valid feature configuration. The *product generation mapping* is the mapping that associates each valid feature configuration to the corresponding product (i.e., the Java program obtained by applying the

delta modules with a satisfied when-clause to the empty program). This mapping may be partial (since a non-applicable delta module may be encountered during product generation, resulting in an undefined product) and ambiguous (since, for a given feature consideration, two different orders of the delta modules that are compatible with the order of the partitions may generate two different products).

A delta module is applicable to a Java program if suitable syntactic conditions are satisfied. For example,

- each class or interface to be added does not exists;
- each class or interface to be removed or modified exists;
- for every interface to be modified, each method to be added does not exist and each method to be removed exists; and
- for every class to be modified, each method or field to be added does not exist, each method or field to be removed exists, and each method to be modified exists and has the same signature as in the method-modify operation.

A suitable type system could guarantee that if a DeltaJ product line is well typed, then its product generation mapping is total and unambiguous and all its products are well-typed Java programs. Such a type system has been formalized for the minimal core calculus IMPERATIVE FEATHERWEIGHT DELTA JAVA (IFΔJ) [7]. However, it is not yet implemented in DeltaJ 1.5 [12,13], where the products are generated by considering delta modules in the order in which they occur in the product-line declaration,[9] and most type errors in the products are detected only when the product is generated.

3.4 Delta-oriented SPL development

Delta-oriented programming is a transformational approach for the development of SPLs [26]. For instance, it supports developing an SPL by starting from at least one complete product, called the *core product*, and writing program transformations (the delta modules) that specify changes to be applied to the core product in order to implement other products. In the following, we use the phrase *core delta module* to refer to a delta module that when applied to the empty product generates the code of a complete product.

The main advantage of transformational approaches over compositional ones (such as, e.g., FOP [10,27–29]) is that when developing an SPL starting from a set of core products, the latter approaches require to begin from simple products (implementing minimal sets of features) to be extended in order to implement the other products. Instead, DOP supports also developing an SPL starting from complex products (implementing arbitrary large sets of features) and transforming them into simpler products by removing features (see [8,11]). More general, DOP is well suited to support the following SPL development approaches: *proactive* (i.e., all reusable artifacts are planned and developed in advance), *reactive* (i.e., only a basic set of products is planned and developed—when new customer requirements arise, the existing SPL is evolved), and *extractive* (i.e., turning a set of existing applications into an SPL); whereas FOP supports only the proactive approach. Note that the simple-core approach roughly corresponds to the proactive approach, while the complex-core approach may be seen as a particular case of the extractive approach. Proactive SPL development is the most appealing choice from the quality point of view. However, it requires a high upfront investment. Krueger [30] proposed therefore the reactive and extractive approaches to reduce the adoption barrier for SPL engineering.

```
1  SPL Album {
2    Features = ... /* Same as in Listing 4 */
3    Deltas = ... /* Same as in Listing 4 */
4    Constraints {... /* Same as in Listing 4 */}
5    Partitions {
6      {Dall} when (Base);
7      {DnoGroups} when (!Groups);
8      {DnoOwner} when (!Owner);
9      {DnoRemove} when (!Remove);
10   }
11   Products { ... /* Same as in Listing 4 */}
12 }
```

Listing 10: DeltaJ 1.5 product-line declaration for the complex-core implementation of the Album PL

The implementation of the Album PL illustrated above follows the simple-core approach. Listing 10 illustrates the SPL declaration of an implementation of the Album PL that follows the complex-core approach. The delta modules (their code is available in the DeltaFineFit home page [18]) are such that Dall introduces the code of the product with all the features, while DnoGroups, DnoOwner, and DnoRemove remove the code that implements the features Groups, Owner, and Remove, respectively.

4 Delta-oriented specification of SPLs

In this section, we describe *delta tables*, a concept that we have developed and used to derive tabular specifications of products by specifying the difference between the product specifications.

4.1 Delta tables and delta-table modules

In Sect. 2.2, we have described the tables used by the FineFit model. We refer to them hereafter as *ordinary tables*. Figure 2 presents the structure of each ordinary table. The idea of delta tables is to derive a new table from an existing one, based on

[9] Thus, ruling out ambiguity.

the difference (the delta) between the tables. This difference is provided in an easily readable way.

A delta table is recognized by its name (the upper-left cell), which is preceded with the Δ symbol. Delta tables have the same structure as the ordinary tables, yet their cells may contain *delta operators* (namely, match "∗," remove "−," insert "►," and replace "∗►"). Let us denote the set of ordinary tables by T_O and the set of delta tables by T_Δ. A delta table t_Δ can be *applied* to an ordinary table t_o, written as $apply(t_o, t_\Delta)$, resulting in a new ordinary table. That is, *apply* is a function $apply : T_O \times T_\Delta \rightarrow T_O \cup \varepsilon$, where ϵ denotes an empty (non-existent) table in the case that the table is removed by the application. Following is an example of a delta-table application which yields a resulting table (to be explained later). We use the infix notation: T_O *apply* T_Δ.

f(x)	$x < 0$	$x \geq 0$
y	x	x
z	1	0

apply

Δf(x)	$x < 0$	$x \geq 0$
y	∗►-x	∗
–	–	–

yields

f(x)	$x < 0$	$x \geq 0$
y	-x	x

A *delta-table module* (DTM) is a set of delta and non-delta tables. Delta-table modules specify the changes to the FineFit specification of a product that are needed to obtain the specification of another product (i.e., another set of tables). For this we introduce the function *Apply*: $\mathscr{P}(T_O) \times \mathscr{P}((T_O \cup T_\Delta)) \rightarrow \mathscr{P}(T_O)$, where $\mathscr{P}(T_O)$ and $\mathscr{P}(T_\Delta)$ denote sets of tables.

To define the *Apply* function, we must distinguish between a table and its name, since when overriding a table a with a table b the tables may be different, i.e., $a \neq b$, yet they must share the same name, e.g., specify the same operation. Let $\bar{t}$ denote the name of table t and $\bar{T}$ denote the set of all table names of the set of tables T. A delta-table module $B \in \mathscr{P}((T_O \cup T_\Delta))$ is applied to a FineFit specification $A \in \mathscr{P}(T_O)$ as follows:

$$Apply(A, B) = \bigcup_{a \in A, b \in B} \begin{cases} \{a\} & \text{if } \bar{a} \notin \bar{B} \\ \{b\} & \text{if } b \in T_O \\ \{apply(a, b)\} & \text{if } b \in T_\Delta, \Delta\bar{a} = \bar{b} \text{ and } apply(a, b) \neq \epsilon \end{cases}$$

The three different cases are: (case 1) tables in original table set that are not present in the DTM are copied to the resulting set of tables; (case 2)[10] ordinary tables of the DTM are copied to the resulting table (and may overwrite tables from the original set A); and (case 3) if a table is marked with the Δ symbol, then it is a delta table and it is applied to the corresponding table in the original set. When $apply(a, b)$ returns ϵ, a is effectively removed. Note that multiple conditions may be satisfied simultaneously; in this case, all tables whose conditions are satisfied are included.

The first version of delta tables, as proposed in [19], is capable of describing in a concise and an intuitive way the modification of value and variable cells and the refinement of condition cells.

The latter means that subconditions can be introduced below existing condition cells of a table. We realized, however, that this condition refinement is not sufficient for handling all modifications that are necessary in practice, as it is often required to change the condition hierarchy in a more complex way. We have investigated several syntactic notations, for example dividing the condition cells into sections with different semantics of applications. Those notations, however, either have been complex or their application process was hard to understand.

The challenge of defining transformations on Parnas tables results from the fact that Parnas tables have both a tabular and a hierarchical structure. When defining operations, we have to decide whether the tables should be treated as matrices or as trees. We found that embedding a matrix in a tree is easier than the other way around, and it allows us to define basic operations that are applicable to all kinds of tables and to all kinds of cells. In particular, these operations allow a concise treatment of hierarchical conditional cells which is the most difficult feature to design. These operations are defined by the *apply* function that is introduced in Sect. 4.2. To handle also the matrix structure of the tables in a concise way and to provide also abbreviations and convenience rules for the user, we define another function–"prepare," which is described in Sect. 4.3. A strength of the approach is that it has one core algorithm that is relatively small and is recursively and uniformly applied on all kinds of cells of the table. Furthermore, the same algorithm is applied to all kinds of tables in our framework.

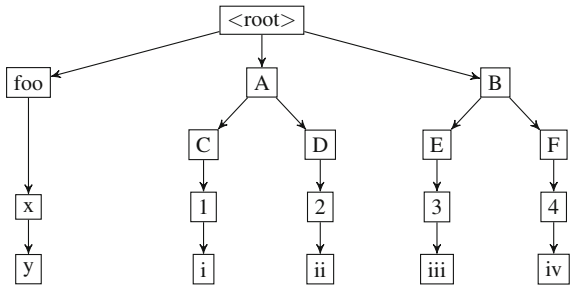
4.2 Hierarchical representation of tables and operations of delta tables

Delta tables have the same syntax as the tables shown in Fig. 2. However, their cells may contain special symbols representing operations that are executed by the *apply* function. The operations and their corresponding symbols are: match "∗," remove "−," insert "►," and replace "∗►." In order to deal with the condition hierarchy of operation tables, the *apply* function treats tables as ordered trees rather than as matrices and traverses them recursively from top to bottom. Consider, for example, the table foo and its encoding as a tree (Fig. 5). The columns of the table are the branches of the tree. A table cell c_1 which is located directly below cell c_0 is a child of c_0 in the tree representation. If c_0 spans several cells, e.g., $c_1, \ldots, c_k$, those cells are children of c_0. The

[10] Note that tables satisfying this case do not satisfy case 1.

Fig. 5 Table foo and its tree structure representation (used for explanations)

foo	A		B	
	C	D	E	F
x	1	2	3	4
y	i	ii	iii	iv



order of siblings must be the same as in the table. Implicitly, there also is present a $< root >$ cell, whose children are the top-level cells of the table. Note that each cell represents a tree, namely the subtree which has the cell as its root. Hence, the entire tree is represented by the $< root >$ cell.

In the following, we define the *apply* function and respectively the semantics of the delta-table operations which are interpreted by this function. We use the infix notation of *apply* and write $\boxed{\text{A}}$ to represent a cell with content A. The notation

$$c_0\text{:}\ \begin{array}{|c|}\hline C \\ \hline c_1 \mid \ldots \mid c_k \\ \hline \end{array}$$

represents a node c_0 with value C and with subnodes $c_1, \ldots, c_k$.[11] The values of c_i are denoted by $op(c_i)$. We omit writing "c_0 :" when possible. Given a table t_o and a delta table t_Δ, t_o *apply* t_Δ yields a resulting table t_r by simultaneously traversing the structures of t_o and t_Δ.

In "Appendix 2," we provide the algorithm of the *apply* function and additional details. Here we present a more intuitive definition, using graphical notation and examples.

4.2.1 Definition of *apply* : $T_O \times T_\Delta \rightarrow T_O$

For each table $t_o \in T_O$ with content A and subnodes $o_1, \ldots, o_m$, and for each $t_\Delta \in T_\Delta$ with subnodes $d_1, \ldots, d_n$ and operation $\delta = op(t_\Delta)$ where $\delta \in \{*, -, \blacktriangleright, * \blacktriangleright\}$, t_o *apply* t_Δ is defined inductively over the tree structure representation of the tables.

– Case "$*$": The match operator acts as a placeholder that matches any cell and copies it to the resulting table.

$$o_0 : \begin{array}{|c|}\hline A \\ \hline o_1 \mid \ldots \mid o_m \\ \hline \end{array} \quad \textit{apply} \quad \begin{array}{|c|}\hline * \\ \hline d_1 \mid \ldots \mid d_n \\ \hline \end{array} \quad \text{yields} \quad \begin{array}{|c|}\hline A \\ \hline r_1 \mid \ldots \mid r_n \\ \hline \end{array}$$

The branches $r_1, \ldots, r_n$ of the resulting table are defined as:

$$r_i := \text{if } op(d_i) \neq \text{" } \blacktriangleright \text{ " then } apply(o_j, d_i) \text{ else } apply(o_0, d_i), \tag{1}$$

where $\blacktriangleright$ is the insert operator, $j = min(i, m)$, and $op(d_i)$ denotes the delta-table operator of node d_i.

Remark 1 The same definition of r_i applies to all the other cases below. Formally, the definition of r_i is the inductive step of the definition of *apply*, but it is presented here in the interest of readability.

Remark 2 When the leaf of the condition structure in the delta table is a match operator and the original table's hierarchy, however, has further subconditions at this position, then the match operator is applied also to the subconditions of the original table. For example, if the delta table has only one condition cell and it is marked with "$*$," then the entire condition hierarchy of the original table is copied to the resulting table.

The placeholder can also be used as a subexpression of a bigger expression, but then it must be embedded in braces "$(*)$," as in the following example:

$$\boxed{x} \quad \textit{apply} \quad \boxed{(*) + y} \quad \text{yields} \quad \boxed{x + y}$$

– Case "$-$": The remove operator matches any cell content A and removes the cell from the resulting table.

$$\begin{array}{|c|}\hline A \\ \hline o_1 \mid \ldots \mid o_m \\ \hline \end{array} \quad \textit{apply} \quad \begin{array}{|c|}\hline - \\ \hline d_1 \mid \ldots \mid d_n \\ \hline \end{array} \quad \text{yields} \quad \begin{array}{|c|}\hline r_1 \mid \ldots \mid r_n \\ \hline \end{array}$$

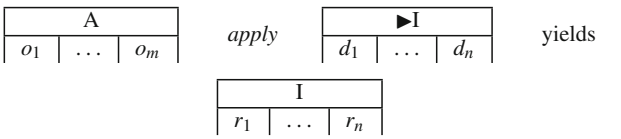
The following example combines operations. The first operation removes a cell (i.e., it is not copied to the resulting table), whereas the second copies the cell into the resulting table.

$$\begin{array}{|c|}\hline A \\ \hline D \\ \hline \end{array} \quad \textit{apply} \quad \begin{array}{|c|}\hline - \\ \hline * \\ \hline \end{array} \quad \text{yields} \quad \boxed{D}$$

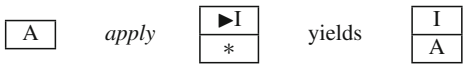
– Case "$\blacktriangleright$": The insert operator inserts a cell c_{ins} at the position of the current cell c_i of the original table. The *apply* function proceeds on subsequent cells as if c_i and all its siblings, $c_1, \ldots, c_i, \ldots, c_n$, are children of c_{ins}.

[11] The bottom line of the cells is removed to indicate that c_i represents a subtree rather than the content of a cell.

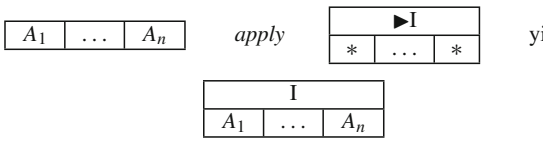
The insert operation is defined partially by the following schema and partially by the recursive step described above (Eq. 1).



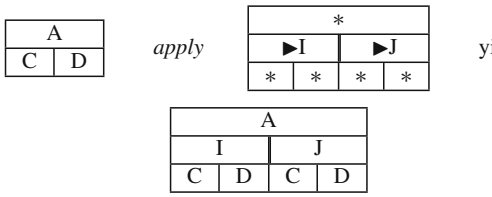
The following example presents the case where c_{ins} has a subsequent cell:



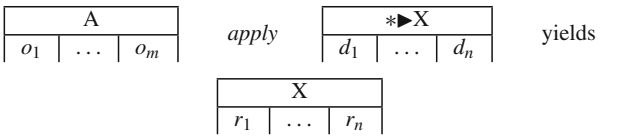
We now demonstrate the case where c_{ins} has several subsequent cells:



Multiple insert operations can be applied to the current cell of the original table. This results in several subbranches such that the children $c_1, \ldots, c_n$ of c_0 occur on each subbranch. This is the desired behavior for refining conditions. For example,

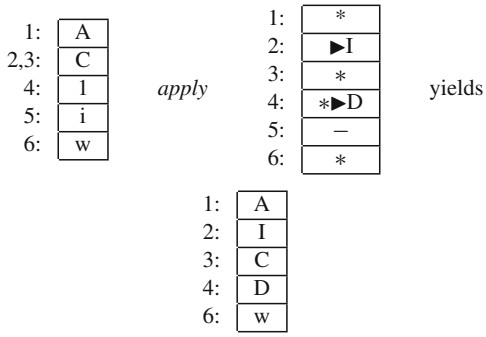


– Case "∗▶": The replace operation replaces the current cell of the original table with the cell of the delta table. This operation can be simulated by a combination of remove and insert operations; introducing it, however, greatly improves readability. For example,

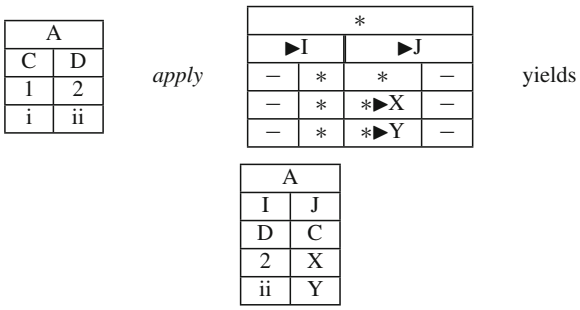


If the replace operation cannot be applied because the original table does not contain a cell at the current position (e.g., it is at the end of the branch), the operation is changed to an insert operation.

Let us demonstrate the application of a delta table that consists of one column and contains all the described operators. The *apply* function simultaneously traverses both the original and the delta tables from top to bottom and applies the delta-table operators. For convenience, we write the recursion step numbers of the *apply* function left to the cells.



Here we show the application of a hierarchical delta table with several columns (branches):



Note that the *apply* function treats columns independently and does not prevent vertical misalignment. Delta tables have to be written in such way that the desired outcome is generated, e.g., Parnas tables. The next section describes techniques that help writing delta tables.

4.3 Preprocessing rules

The *apply* function defined in the previous section takes care of the hierarchical nature of Parnas tables and provides a simple-core transformation that is uniformly applied to all cells. Here we introduce the *prepare* function, which takes care of the *tabular*[12] nature of Parnas tables and defines a set of rules that simplify writing and reading of delta tables. The *prepare* function acts as a transformation layer on top of the *apply* function; i.e., it compiles a delta table with abbreviations and higher-level constructs into a delta table that uses only the basic operations.

The preprocessing rules are defined by the function $prepare : T_O \times T_\Delta \rightarrow T_\Delta$ which takes as parameters an ordinary table and a delta table and returns a resulting delta table. The resulting delta table is used as input to the *apply* function, as has been defined in the previous section. The function $apply' : T_O \times T_\Delta \rightarrow T_O \cup \epsilon$ is the composition of the $apply : T_O \times T_\Delta \rightarrow T_O \cup \epsilon$ and the *prepare* functions and is defined as:

[12] Unlike the *apply* function, the *prepare* function is aware of the row-alignment of cells in a table and provides special treatment for different kinds of cells.

$$apply'(t_o, t_\Delta) = apply(t_o, prepare(t_o, t_\Delta)) \quad (2)$$

for all $t_o \in T_O$ and $t_\Delta \in T_\Delta$. The function $Apply' : \mathscr{P}(T_O) \times \mathscr{P}((T_O \cup T_\Delta)) \rightarrow \mathscr{P}(T_O)$ which applies a delta-table module on a set of ordinary tables is similar to the function *Apply* that is described in Sect. 4.1, but uses the function *apply′* rather than *apply*.

In the following, we informally describe the *prepare* function with a set of rules that must be applied in the given order. Some of the rules do not depend on a particular ordinary table, and in this case we omit the ordinary table (i.e., we omit writing the first argument).

Rule 1 When the delta table contains a cell with no associated operation, $*\blacktriangleright$ is the default operation. Thus, in principle the user does not have to use the $*\blacktriangleright$ symbol. However, the user should write $*\blacktriangleright$ explicitly when a cell's content is changed to indicate that it is not just overwritten with the same content.

Recall that $*\blacktriangleright$ can behave like an insert operation (from the definition of *apply* in Sect. 4.2.1). Hence, depending on the position of $*\blacktriangleright$ and its content it can replace a cell, keep a cell (by replacing it with the same content), or insert a cell.

The leading Δ symbol is removed from the name of the delta table to match the name with the original table.

Rule 2 If *all* cells below a condition cell or a name cell c contain the remove operation, c is removed as well. For example:[13]

ΔT	A			
	B		C	
	D	–	F	G
x	–	2	–	–
y	–	ii	–	–

prepare yields

*▶T	*▶A			
	*▶B		–	
	–	–	–	–
*▶x	–	*▶2	–	–
*▶y	–	*▶ii	–	–

The first rule is applied here and is responsible for adding the $*\blacktriangleright$ operations. The cells A and B are not removed because cells exist below them which are not removed, namely 2 and ii.

This rule enables the user to remove a column of value cells and the corresponding conditions, without having to write "–" in the condition cells. This makes it easier to see which condition is removed. Furthermore, this rule enables to remove an entire table T from the DTM by writing:

ΔT
–

In order to remove all cells of the table except for the name cell, the user may write | ΔT | .

Rule 3 A variable cell may contain a set of variables. These are expanded by the preprocessing step into a set of rows, one for each variable. For instance:

Given

ΔT	A	B
{x, y}	1	3

, *prepare* yields

*▶ T	*▶ A	*▶ B
*▶ x	*▶ 1	*▶ 3
*▶ y	*▶ 1	*▶ 3

Rule 4 The user may write $*$ in a variable cell in order to refer to all variables of the original table that are not explicitly mentioned in the delta table. A row that contains the $*$ operator in its variable cell is called a *default row*. Only one default row may be specified. A default row is added for each variable not mentioned in the variable cells of the delta table.[14] For instance, given

T	A	B
x	1	3
y	2	4

and

ΔT	A	B
*	–	*
z	–	6

prepare yields

*▶T	–	*▶B
*	–	*
*	–	*
*▶z	–	*▶6

The actual delta table that is applied in the following example is, hence, the one above, as created by the preprocessing. Recall that according to the definition of *apply*, the replace operations in the last row of this example behave like insert operations.

T	A	B
x	1	3
y	2	4

apply′

ΔT	A	B
*	–	*
z	–	6

yields

T	B
x	3
y	4
z	6

Rule 5 When no default row is specified in the delta table, rows of the original tables with variable names that do not occur in the delta table are copied to the resulting table. This is achieved by filling the cells of the row with the

[13] Recall that the row span of the name cell defines that the first three rows of the example are condition rows and the other rows contain variable and value cells.

[14] This includes variables checked by the conditional operator.

match (*) operation, except for those cells corresponding to columns that are removed by the delta table. A column of a delta table is removed if all its value or variable cells contain the remove operation. For instance, given

T	A	B
x	1	3
y	2	4

and

ΔT	A	B
z	–	6

, *prepare* yields

*▶T	–	*▶B
*	–	*
*	–	*
*▶z	–	*▶6

Hence:

T	A	B
x	1	3
y	2	4

apply′

ΔT	A	B
z	–	6

yields

T	B
x	3
y	4
z	6

Rule 6 Here we introduce the conditional remove operator X**:**– which may be used only in the left most column of a table. The operator applies a remove operation to the first occurrence of a cell with content X. If such a cell does not exist in the original table, the entire row of the delta table with the X**:**– operator is ignored. For example:

Given

A
B
C
D
E

and

C :–
*

, *prepare* yields

*
*
–
*
*

The ∗ operator below C**:**– works as a default row (see Rule 4) and is needed to keep all the other cells. Without the ∗ operator, rule 2 treats C**:**– simply as a remove operation and the entire column would be removed in this example.
Hence:

A
B
C
D
E

apply′

C :–
*

yields

A
B
D
E

The purpose of the conditional remove operator is to provide a convenient notation to remove cells from an enumeration table.

Rule 7 Rows of the delta table with variable and value cells are sorted such that the variables names match those of the original table.[15] Thus, the user does not have to pay attention to the exact order of variables but only to their names. Rows of the delta table with additional variables (that do not exist in the original table) are appended at the end. For example:

Given

T	A
x	1
y	2

and

ΔT	A
y	3
z	4
x	5

, *prepare* yields

*▶T	*▶A
*▶x	*▶5
*▶y	*▶3
*▶z	*▶4

Note that the ∗▶ operation is used as the default one due to rule 1. This has the effect that the value cells of corresponding rows are overwritten with values from the delta table, and rows defining new variables are added to the resulting table. For example:

T	A	B
x	1	3
y	2	4

apply′

ΔT	A	B
y	5	*
z	6	7
x	*	8

yields

T	A	B
x	1	8
y	5	4
z	6	7

Rule 8 When all value cells in a row are removed, also the variable cell of that row is removed. This provides an intuitive way to remove variables. For instance:

Given

ΔT	A	B
x	–	–
y	5	6

, *prepare* yields

*▶T	*▶A	*▶B
–	–	–
*▶y	*▶5	*▶6

Hence:

T	A	B
x	1	3
y	2	4

apply′

ΔT	A	B
x	–	–
y	5	6

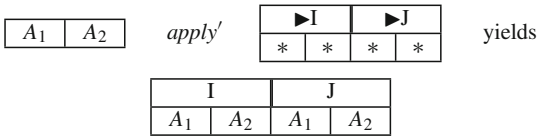
yields

T	A	B
y	5	6

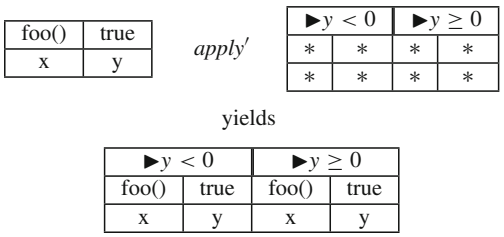
Rule 9 The insert operation has been defined in a way that enables refinement of conditions (see Sect. 4.2). Recall that if a cell c_{ins} is inserted at the subtree where the

[15] Rows with conditional remove operators are also moved to the right place by this rule, i.e., where the content of a cell matches the condition.

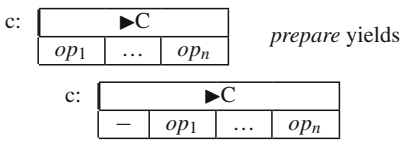
corresponding original subtree has cells $c_1, \ldots, c_n$, then $c_1, \ldots, c_n$ are treated as children of c_{ins} for subsequent operations.
For example:



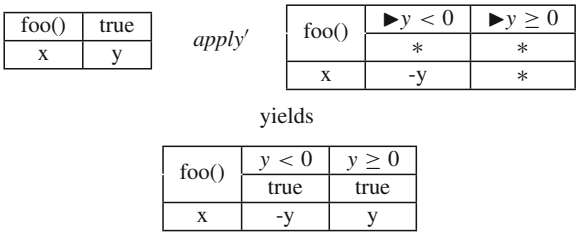
The above example is problematic if A_1 is a name cell; i.e., it contains the table's name, and I and J are condition cells. In this case, the name cell A_1 becomes a child of the condition cells I and J which clearly is wrong.



Hence, in order to obtain a well-formed operation table (Sect. 2.2.2), the first column below the inserted condition cell must be removed when refining the top-level conditions of a table. To release the user from this task and to improve readability of the tables, the following preprocessing rule is applied. If c is a top-level table cell, i.e., a child of the root node, containing an insert operation (say ▶C), then given:



where $op_1 \ldots op_n$ are delta operations. Once this rule is used, the delta application in the following example is correct:



The "true" has not been removed for demonstration purpose, though it easily can be removed using the remove operation above or below the insert operation. Another possibility to remove the cells with the "true" condition is to use the remove operation instead of the insert operation. This works despite the fact that the second remove operation does not have a corresponding cell in the original table.

5 Specifying and testing the Album PL with DeltaFineFit

In this section, we outline how to specify and test a delta-oriented SPL using DeltaFineFit. First (in Sect. 5.1), we illustrate how to write the DeltaFineFit specification. Then (in Sect. 5.2), we show how to extend a DeltaJ SPL in order to generate for each product a version of the product equipped with the code for using FineFit. Finally (in Sect. 5.3), we outline how DeltaFineFit can be used effectively to test an SPL.

5.1 DeltaFineFit specification

A DeltaFineFit specification of an SPL is a product line where each product is a FineFit specification of a Java program (cf. Sect. 2.2). It consists of a product-line declaration and a set of delta-table modules. Delta-table modules are sets of delta and non-delta tables (cf. Sect. 4.1), and a DeltaFineFit product-line declaration is a DeltaJ product-line declaration where delta-table modules are used instead of delta modules. Notably, a DeltaJ product-line declaration can be understood as a DeltaFineFit product-line declaration by providing a one-to-one mapping between delta-table modules and delta modules.

The DeltaFineFit generation procedure is analogous to the DeltaJ 1.5 generation procedure of a Java program (outlined in Sect. 3.3): given a valid feature configuration the DTMs with a satisfied when-clause are applied to the empty FineFit specification in the order in which they appear in the DeltaFineFit product-line declaration. This procedure describes a mapping that associates each valid feature configuration to the corresponding FineFit specification, and this mapping may be partial.

The tables describing the model of the Base Album (see Fig. 3) can be seen as the DTMBase delta-table module that when applied to the empty abstract model produces the model of the Base Album. The other DTMs of the Album PL specification are listed in Figs. 6, 7, 8, and 9. The delta-table module DTMRemove (Fig. 6) specifies the Remove feature which enables to remove a photo from the album. The delta-table module DTMOwner (Fig. 7) specifies the Owner feature which protects the photo album by enabling only the owner (who has to login) to modify the album. The DTMRemoveAndOwner delta-table module (Fig. 8) introduces the needed modifications to specify the combination of the two optional features Remove and Owner. The delta-

removePhoto(l?:Int, report!:Report)	l? in dom[photoAt]	l? !in dom[photoAt]
photoAt	photoAt.delete[l?]	=
report!	OK	NO_PHOTO

Δ Report
NO_PHOTO

Fig. 6 Delta-table module DTMRemove: It enables to remove a photo from the album

ΔAtom	**Scope**
Name	4
Password	4

Δ Report
AUTH_FAILED
ALREADY_IN

ΔState variable	**Type**
ownerName	Name
ownerPassword	Password
loggedIn	set Name

ΔInvariant name	**Invariant**
AtMostOneLoggedIn	lone loggedIn

Δ init(n?:Name, p?:Password)	true
ownerName	n?
ownerPassword	p?
loggedIn	none

login(n?:Name, p?:Password, report!:Report)	no loggedIn		some loggedIn
	n? = ownerName and p? = ownerPassword	n? != ownerName or p? != ownerPassword	true
loggedIn	loggedIn + n?	=	=
report!	OK	AUTH_FAILED	ALREADY_IN

logout()	true
loggedIn	none

Δ addPhoto(p?:Photo, report!:Report)	▶ ownerName in loggedIn			▶ ownerName !in loggedIn		
	*		*	-		true
	*	*	*	-	-	true
report!	*	*	*	-	-	AUTH_FAILED

Fig. 7 Delta-table module DTMOwner (it contains a delta-atoms table, a delta-elements table, a delta-state table, two operation tables, and one delta-operation table): Only the owner can add a photo

Δ removePhoto(l?:Int, report!:Report)	▶ ownerName in loggedIn		▶ ownerName !in loggedIn	
	*	*	-	true
report!	*	*	-	AUTH_FAILED

Fig. 8 Delta-table module DTMRemoveAndOwner: A photo can be removed only by its owner

table module DTMGroups (Fig. 9) specifies a further level of protection by requiring the owner to create users and to associate each photo with the group of users that can view it.

The simple-core implementation of the Album PL (Listing 4) can therefore be understood as the DeltaFineFit product-line declaration for the specification of the Album PL, modulo the mapping that associates the delta-table modules DTMBase, DTMRemove, DTMOwner, DTMRemoveAndOwner, and DTMOwner to the delta modules DBase, DRemove, DOwner, DRemoveAndOwner, and DOwner, respectively.

The specification of the Album PL (Figs. 6, 7, 8 and 9) uses the simple-core approach (cf. Sect. 3.4). The declaration of the complex-core DeltaJ 1.5 implementation of the Album PL given in Listing 10 can be used as a declaration of a complex-core DeltaFineFit specification of the Album PL. This is achieved by associating to the delta modules Dall, DnoGroups, DnoOwner, and DnoRemove the appropriate delta-table modules DTMall, DTMnoGroups, DTMnoOwner, and DnoRemove, respectively. The tables of the complex-core specification are available at the DeltaFineFit home page [18]. Note that the tables of the core DTM for the product with all features, DTMall, are the tables in the specification of the product with all features.

In the following examples, we take a close look at some of the specification tables from the figures.

Example 1 This example shows the application of the *Δ***addPhoto** delta operation table of the delta-table module DTMOwner to the operation table **addPhoto** of the base product. The original operation table (from Fig. 3) is defined as:

Fig. 9 Delta-table module DTMGroups: Only groups of users authorized by the owner may view photos in the album

Δ Atom	Scope
User	4
Group	4

ΔState variable	Type
groups	Name → Group
users	Name → User
members	Group → User
passwords	User → Password
groupPhotos	Photo → Group
ownerPassword	Name

ΔReport
WRONG_PASSWORD
ALREADY_IN
MISSING_USERS
MISSING_USER
REM_OWNER
REM_OWNER_GROUP
NO_GROUP
NO_PHOTO

ΔInvariant name	Invariant
validMembers	members in ran[groups] → ran[users]
validPasswords	passwords in ran[users] → one Password
validGroups	groupPhotos in Photo → ran[groups]

Δ init(o?:User, p?:Password, n?:Name, ogn?:Name, og?:Group)	true
ownerPassword	–
passwords	o? → p?
users	n? → o?
groups	ogn? → og?
members	og? → o?
ownerGroupName	ogn?

Δ addPhoto(p?:Photo, report!:Report)	*			
	*		*	*
	*	*		
groupPhotos	groupPhotos + p? → OWNER_GROUP	=	=	=

Δ viewPhotos(result!: seq Photo, report!:Report)	-	
	one loggedIn	no loggedIn
result!	photoAt.elems	none
report!	OK	AUTH_FAILED

Δ login(n?:Name, p?:Password, report!:Report)	*		*
	∗▶n? in dom[users] and passwords[users[n?]] = p?	∗▶n? !in dom[users] or passwords[users[n?]] != p?	*
	▶ p? = ownerPassword	▶ true	▶ true
loggedIn	loggedIn + users[n?]	=	=

addPhoto(p?:Photo, report!:Report)	#photoAt < MAX		#photoAt ≥ MAX
	p? !in ran[photoAt]	p? in ran[photoAt]	true
photoAt	photoAt.add[p?]	=	=
report!	OK	PHOTO_EXISTS	ALBUM_FULL

The delta-operation table (at the bottom of Fig. 7) is defined as:

Δ addPhoto(p?:Photo, report!:Report)	▶ ownerName in loggedIn			▶ ownerName !in loggedIn		
	*		*	-		true
	*	*	*	-	-	true
report!	*	*	*	-	-	AUTH_FAILED

The application results in the following operation table:

addPhoto(p?:Photo, report!:Report)	ownerName in loggedIn			ownerName !in loggedIn
	#photoAt < MAX		#photoAt ≥ MAX	true
	p? !in ran[photoAt]	p? in ran[photoAt]	true	true
photoAt	photoAt.add[p?]	=	=	=
report!	OK	PHOTO_EXISTS	ALBUM_FULL	AUTH_FAILED

Then, applying to the above table the following delta-operation table **ΔaddPhoto** of the delta-table module DTMGroups (from Fig. 9)

Δ addPhoto(p?:Photo, report!:Report)	*			
	*		*	*
	*	*		
groupPhotos	groupPhotos + p? → OWNER_GROUP	=	=	=

yields an operation table with an additional row (which belongs to the specification of the product with features Base, Owner and Groups). Here we use a new syntax, where condition cells with a match operation span multiple cells of a column. These condition cells in the corresponding column of the original table are copied to the resulting table.

Note that using the following table to add the additional row is not correct, because multiple variable cells occur below a condition cell:

Δ addPhoto(p?:Photo, report!:Report)	*			
groupPhotos	groupPhotos + p? → OWNER_GROUP	=	=	=

This is not a valid operation table, and delta tables must satisfy the syntax of ordinary tables as defined in Fig. 2.

Example 2 Next, we consider applying the delta-operation table **Δlogin** of DTMGroups to the operation table **login** of DTMOwner. The original operation table **login** (last but third table in Fig. 7) is:

login(n?:Name, p?:Password, report!:Report)	no loggedIn		some loggedIn
	n? = ownerName and p? = ownerPassword	n? != ownerName or p? != ownerPassword	true
loggedIn	loggedIn + n?	=	=
report!	OK	AUTH_FAILED	ALREADY_IN

The delta-operation table **Δlogin** (third table in Fig. 9) is defined as:

Δ login(n?:Name, p?:Password, report!:Report)	*		*
	*▶n? in dom[users] and passwords[users[n?]] = p?	*▶n? !in dom[users] or passwords[users[n?]] != p?	*
	▶ p? = ownerPassword	▶ true	▶ true
loggedIn	loggedIn + users[n?]	=	=

As a result, the following operation table is generated:

login(n?:Name, p?:Password, report!:Report)	no loggedIn		some loggedIn
	n? in dom[users] and passwords[users[n?]] = p?	n? !in dom[users] or passwords[users[n?]] != p?	true
	p? = ownerPassword	true	true
loggedIn	loggedIn + users[n?]	=	=
report!	OK	AUTH_FAILED	ALREADY_IN

5.2 Delta-oriented programming of the retrieve function and of the driver

The retrieve function and the driver can be conveniently programmed in a delta-oriented way by extending the SPL with a new optional feature, FineFit, that when selected generates a product that is equipped with the retrieve method and the class extending the FineFitDriver base class (cf. Sect. 2.3).

To this aim, the declaration of the Album PL is modified by adding the feature FineFit and appending (after the last line of the partitions block) the partitions with the delta modules for adding the retrieve function and the driver. The resulting declaration is given in Listing 11. There is no need for a FineFit delta module corresponding to RemoveAndOwner since it only changes the implementation of remove and therefore does not affect the interface (cf. Listing 8).

The code of the delta module DBaseFineFit is given in Listings 12. The complete code of all the delta modules is available at [18].

```
1  SPL Album {
2    Features = {Base, Remove, Owner, Groups, FineFit}
3    Deltas = {DBase, DRemove, DOwner, DRemoveAndOwner,
4              DGroups, DBaseFineFit, DRemoveFineFit,
5              DOwnerFineFit, DGroupsFineFit}
6    Constraints { ... /* Same as in Listing 4 */ }
7    Partitions {
8          ... /* Same as in Listing 4 */
9      {DBaseFineFit} when (Base & FineFit);
10     {DRemoveFineFit} when (Remove & FineFit),
11     {DOwnerFineFit} when (Owner & FineFit);
12     {DGroupsFineFit} when (Groups & FineFit);
13   }
14   Products {
15         ... /* Same as in Listing 4 */
16     A_BaseFineFit = {Base, FineFit};
17     A_RemoveFineFit = {Base, Remove, FineFit};
18     A_OwnerFineFit = {Base, Owner, FineFit};
19     A_RemoveOwnerFineFit = {Base, Remove, Owner,FineFit};
20     A_OwnerGroupsFineFit = {Base, Owner, Groups, FineFit};
21     A_OwnerRemoveGroupsFineFit = {Base, Remove, Owner,
22                                   Groups, FineFit};
23   }
24 }
```

Listing 11: Declaration of the Album PL extended with the products equipped with code for FineFit testing

```
1  delta DBaseFineFit {
2
3    modifies it.unito.Album.ArrayPhotoAlbum {
4      import com.finefit.sut.IdMap;
5      import com.finefit.sut.State;
6      adds  ... /* Same as in Listing 2 (bottom) */
7      }
8
9    adds { package it.unito.Album;
10     ... /* Same as in Listing 3 */
11   }
12
13 }
```

Listing 12: Delta module DBaseFineFit for adding the retrieve method and the FineFitDriver class to the Base product of the Album PL

5.3 Executing the tests

The product for a given feature configuration can be tested by: (1) generating its FineFit specification from the DeltaFineFit specification of the SPL, (2) generating a version of the product equipped with the support code for using FineFit, and (3) running FineFit. Note that the entire testing procedure for any product is fully automatic once the delta modules for implementing the feature FineFit have been written and the SPL declaration has been extended (cf. Sect. 5.2).

The number of products of an SPL can be exponential in the number of features. Testing all the products may be therefore unfeasible, even when the entire testing procedure is fully automatic (as in the case of DeltaFineFit). This issue can be handled by using *sample-based SPL testing* techniques [14–17], which address the problem of scalability by identifying a subset of products that is supposed to cover relevant combinations of features.

Thus, only the products in the identified subset have to be generated and tested by applying single-system testing techniques.

6 Evaluation of the approach

In this section, we briefly evaluate the benefits of integrating data-refinement-based testing into delta-oriented SPL development from both a qualitative and a quantitative perspective.

6.1 Qualitative evaluation

The DeltaFineFit product-line declaration and the DeltaJ product-line declaration may be different in principle. For instance (as an extreme case), the complex-core DeltaFineFit specification of the Album PL can be used as a specification of the simple-core DeltaJ implementation of the Album PL (cf. Sects. 3.4 and 5.1). However, it is most effective to share the same product-line declaration for both the specification and the implementation, as illustrated in Sect. 5.1. Therefore, the delta-table construct must be expressive enough to allow delta-table modules to mimic delta modules, i.e., to allow developers to establish a one-to-one correspondence between delta-table modules and delta modules. Based on our experiments, we believe that the delta-table construct presented in this paper satisfies this requirement.

Overall, the proposed integration of data-refinement-based testing with delta-oriented SPL development supports:

1. automatic generation of product specifications from delta-table modules that correspond to the delta modules by sharing the product-line declaration with the implementation;
2. automatic generation and inclusion of auxiliary code for FineFit's refinement-based testing into the products (cf. Sect. 5.2); and
3. a clear layout of the commonalities and differences between product configurations in both the product specifications and the auxiliary code needed for using FineFit.

6.2 Quantitative evaluation

In the following, we present a quantitative evaluation of how concise is the specification of an SPL using the delta-oriented approach compared with the specification that does not use this approach. As case studies we considered the simple-core-based development and the complex-core-based development of the Album PL. Table 1 illustrates the size of the specification of each of the six products of the Album PL, where each product is specified separately (without using the delta-oriented approach). In this counting scheme, we counted all rows of the tables excluding the title row for columns in constant, atom, state, and invariant tables.[16] Using the information in Table 1, it is straightforward to compute for each kind of table the total number of tables and rows for all products (these numbers are presented in Table 2 in the columns of the naive approach).

Table 2 illustrates a quantitative comparison of three different specification approaches for the Album PL study:

- naive specification (i.e., each product is specified separately),
- simple-core delta-oriented specification, and
- complex-core delta-oriented specification.

The naive approach requires 69 tables with a total of 301 rows to specify the whole product line. Less than half the number of tables (32) and rows (114) are required when using DTMs in the simple-core delta-oriented specification approach. The number of tables and rows in the complex-core delta-oriented specification approach is bigger than in the simple-core approach. Note that:

- DTMs in simple-core specifications define how to add and expand tables,
- DTMs in complex-core specifications define how to remove and trim tables, and
- the set of delta tables for expanding tables in the simple-core specification has roughly the size (in the number of both tables and rows) of the set of delta tables for removing and trimming tables in the complex-core specification.

Therefore, the difference in the number of tables and rows is mainly due to the fact that the DTM used to build the core for complex-core specifications is lager than the one used for simple-core specifications, in the number of both tables (16 vs 8) and rows (93 vs 18)—c.f. Table 1. Conversely, the complex-core specification has less DTMs.

It is worth to observe that the ability to support complex-core specifications and, more general, the ability to build specifications where the DTMs mimic the delta modules in the implementation (cf. Sect. 6.1) makes it possible to use DeltaFineFit in reactive and extractive SPL development (cf. Sect. 3.4).

The case study confirms our expectations that the delta-oriented specification approach reduces the size of the specification of the SPL (both with the simple- and complex-core approaches). The Album PL has only four features and six products. We expect that these benefits will be even more significant in more complex software product lines.

[16] The first row of enumeration tables, which consists of an atom name, was counted.

Table 1 Size of the specifications of the six products of the Album PL without using DTMs

Features	Base		Base remove		Base owner		Base remove owner		Base owner groups		Base remove owner groups	
Tables	*T*	*L*	*T*	*L*	*T*	*L*	*T*	*L*	*T*	*L*	*T*	*L*
Constants	1	1	1	1	1	1	1	1	1	1	1	1
Atoms	1	3	1	3	1	5	1	5	1	7	1	7
Enumeration	1	3	1	4	1	6	1	7	1	14	1	14
State	1	1	1	1	1	4	1	4	1	8	1	8
Invariants	1	2	1	2	1	3	1	3	1	6	1	6
Operations	3	8	4	11	5	18	6	2	10	53	11	57
Total	8	18	9	22	10	37	11	42	15	89	16	93

T is the number of tables, and *L* is the total number of lines in these tables. The specification of the Base Album is illustrated in Fig. 3, and the specifications of the other five products are available at the DeltaFineFit home page [18]

Table 2 Quantitative comparison of specification approaches

Approach	Naive		Simple-core delta oriented			Complex-core delta oriented		
Number of DTMs	6		5			4		
Tables	*T*	*L*	*T*	*L*	in Figures	*T*	*L*	The tables are available at the DeltaFineFit home page [18]
(Δ)Constants	6	6	1	1	3	1	1	
(Δ)Atoms	6	30	3	7	3, 7, 9	3	13	
(Δ)Enumeration	6	48	3	15	3, 7, 9	4	28	
(Δ)State	6	26	3	10	3, 7, 9	3	18	
(Δ)Invariants	6	22	3	6	3, 7, 9	3	12	
(Δ)Operation	39	169	11	75	3, 6, 7, 8, 9	26	103	
Total	69	301	32	114	–	40	175	–

T is the number of tables, and *L* is the total number of lines in these tables. For the naive specification approach, the specification of each of the six products of the Album PL can be considered as a core DTM (remember that a constants/atoms/enumeration/state/invariants table can be considered as a delta table)

7 Related work

The use of a tabular notation to specify operations was inspired by Parnas [6]. Tabular expressions can be represented and interpreted in many ways. Parnas proposes precise definitions for ten varieties of tabular expressions [6]. We use the simple and intuitive Vector Function Tables to describe functions.

Various surveys deal with testing in SPLs; e.g., [31–34]. Delta tables leverage the idea of tabular notation to the specification of product lines. To the best of our knowledge, this idea has not been proposed yet. The only papers introducing similar operations on tables, also named delta tables, describe updates or views on database, e.g., [35]. However, it seems that this idea has not been proposed before to specify programs.

The approach proposed by Uzuncaova et al. [36] seems the most related to our testing approach. The paper describes a scope-bounded testing technique for programs in an SPL developed by using the feature-oriented programming (FOP) approach (cf. Sects. 1 and 3.4). In the FOP approach, a program is a composition of feature modules, where each module implements an increment in the functionality of the program corresponding to a feature. This testing approach differs from ours as their test suite enables only to add features, whereas we allow also to remove features. Hence, the test suite in [36] is monotonically extended with each additional feature. Using the Alloy Analyzer, tests are generated and the correctness of the program is checked. Each feature F is specified as an Alloy formula, and the Alloy specification of a product is the conjunction of the specifications for its features. To generate larger inputs, tests are generated incrementally, as the analyzer is used each time on partial specifications rather than on the complete specification of the program. First, the analyzer solves the specification for the Base feature (which corresponds to the base product), and the created relations are used when running the analyzer on the specification of a feature F that is composed with Base. In our approach, we generate both the program and its specification using the DOP approach, where a product is

generated from another one by applying a delta operation that may add or remove functionalities. The tests are then generated by considering the complete specification of a product. However, when a program is a composition of features, i.e., when the delta modules do not use the remove operation, it should be possible to apply our approach by using the incremental approach as in [36].

More recently, Lochau et al. [37,38] proposed a model-based SPL testing framework that is based on a delta-oriented SPL test model and on regression-based test artifact generation. The framework comprises state machines as test models extended with delta-oriented modeling concepts to express variability. The framework is aimed to capture reuse potential, reusing both (1) test cases for different product variants and (2) test results across different products. In our framework, no reuse of test cases or test results is currently supported. We believe that the technique proposed in [37] could be applicable also to our approach, although we are using a different specification language (Parnas tables).

Proof systems for the verification of delta-oriented SPLs have been recently proposed [39,40]. These approaches are related to ours since they use DOP to generate a product together with its specification; however, they aim at a formal verification. Hähnle and Schaefer [40] presented a verification approach for DOP that relies on the Liskov principle for DOP; i.e, the specification of a method modified by a delta module must entail the specification of the previous versions of the method. This ensures that each delta module can be verified by approximating called methods that are defined in other delta modules by the specification of their first introduction. In order to relax the restriction imposed by the Liskov principle for DOP [40], Damiani et-al. [39] suggest to use symbolic assumptions on called methods in order to separate the specifications of method implementations from the requirements to method calls in a way which is similar to lazy behavioral subtyping [41]. The idea is to verify first each delta module in isolation, based on symbolic assumption for methods calls that may be defined in other delta modules, and then to verify each product based on the already established specification for the used delta modules. The combination of software verification and testing techniques is encouraged due to their complementary strengths. We believe that some ideas for combining verification and testing that already had been implemented for a single system (see [42]) could be applied to SPLs.

8 Conclusion and future work

We illustrated how the FineFit approach can be integrated into DOP to support the development of correct software product lines. Refinement-based testing is completely integrated into the delta-oriented SPL development process. Delta-oriented programming is used not only to generate the system under test, but also to generate the FineFit specification, the retrieve function, and the driver for each product.

We would like to complete the development of the DeltaFineFit tool chain. In particular, we would like to improve the tool support for the delta-oriented programming of the retrieve function and the driver by developing a tabular notation that will be automatically translated into the corresponding delta modules.

A type system for DOP, formalized for the minimal core calculus IMPERATIVE FEATHERWEIGHT DELTA JAVA (IFΔJ) [7], guarantees that if an IFΔJ product line is well typed then its product generation mapping is total and all its products are well-typed Java programs (cf. Sect. 3.3). In future work, we plan to formalize a suitable notion of well-formed FineFit specification and to develop static analysis techniques for checking whether the product generation mapping of a DeltaFineFit specification is total and all its products are well-formed FineFit specifications. Moreover, we would like to extend the DeltaFineFit approach to deal with dynamic DOP [43,44].

We also plan to develop case studies to assess the effectiveness of using DeltaFineFit with sample-based SPL testing techniques (see Sect. 5.3) and to enhance DeltaFineFit by exploring both the use of techniques for reusing test cases or test results similar to [37] (cf. Sect. 7) and the use of test prioritization techniques similar to [45]. Another enhancement of DeltaFineFit that we would like to explore is the integration of verification techniques (see the discussion at the end of Sect. 7).

Acknowledgements We thank the anonymous referees of PPPJ'13 for valuable comments on a preliminary version of this paper and the anonymous SoSyM referees for many insightful comments and suggestions for improving the paper.

Appendix 1: The structure and semantics of operation tables

A FineFit operation table is a predicate that specifies the behavior of an operation as a relation between the model's state variables before the operation starts (the pre-state) and after the operation completes (the post-state). It consists of two major areas: an expression table and a precondition tree. The expression table is a set of columns, where each column defines the values of the state variables in the post-state, given their values in the pre-state. The predcondition tree consists of predicates that determine which columns to use in the definition of the post-state. For example, consider the following operation table:

R(c?:Int)	$c? \leq 0$		$c? > 0$
	$x < 0$	$x \geq 0$	true
x	=	0	$x + c?$
y	=	0	$x + y$

The operation R has a single input parameter $c?$ and determines the value of its two state variables x and y as follows: If $c?$ is positive, then the operation must set x to $x + c?$ and y to $x + y$ (the right most column), otherwise, if $c?$ is not positive, the effect of the operation depends on the current value of x. When x is not negative x and y must be set to zero and when x is negative, then x and y do not change (their value must be the same as it was when the operation started). Note that the predicate for determining whether columns 1 and 2[17] are relevant for the specification is the conjunction of the child predicates $x < 0$ and $x \geq 0$ with their parent predicate $c? \leq 0$.

We now define the conditions necessary for the precondition part of the table to form a tree. Consider the precondition part of the table as a matrix of cells. Then, each predicate spans one or more consecutive cells (in the same row) and the space that each predicate occupies must be included below the space that its parent (the predicate above) occupies. For example, the predicate $x \geq 0$ occupies cell $(2, 2)$ and its parent, the predicate $c? \leq 0$ occupies cells $(1, 1)$, $(1, 2)$.

Formally, consider a precondition part that is arranged in a matrix of n rows by m columns of cells. The i-th row contains a sequence of predicates, each occupying a span of one or more cells. Let k_i indicate the sequence of spans that each predicate occupies (and therefore k_{ij} is the j-th span in the i-th row). For example, in the operation above, $k_1 = 2, 1$ and $k_2 = 1, 1, 1$. In order for the spans to describe a valid precondition tree, three conditions must be met:

1. The sum of spans in each row must be equal to the number of columns in the matrix:

$$\sum_{j=1}^{|k_i|} k_{ij} = m$$

where $|k_i|$ is the number of elements in the ith sequence of spans.

2. The cells that each predicate spans must be included below the span of cells of its parent:

$$S_{ij} \subseteq S_{(i-1)l} \quad \text{for some } 1 \leq l \leq m$$

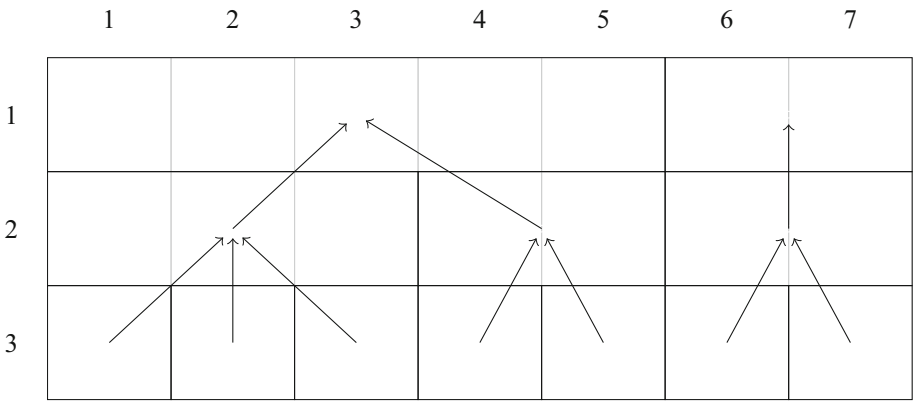
where the span of cells of the j-th predicate in the i-th row is

$$S_{ij} = \sum_{l=1}^{j-1} k_{il} + 1, \ldots, \sum_{l=1}^{j} k_{il}$$

3. The last row (n) must consist of individual cells, that is

$$k_{nj} = 1 \quad \text{for all } 1 \leq j \leq m.$$

This arrangement ensures that the predicates form a tree structure[18] with m leaves, all occupying the last row. The following matrix illustrates this structure:



There are three span sequences, one for each row:

$$k_3 = 1, 1, 1, 1, 1, 1, 1 \quad (3)$$
$$k_2 = 3, 2, 2 \quad (4)$$
$$k_1 = 5, 2 \quad (5)$$

[17] We do not count the left most column containing the state variables.

[18] In fact, this is a forest. However, it can be transformed into a tree by adding an imaginary *true* as the parent of the predicates at the first row.

We can see that each span is included below the span of its parent. For example,

$$S_{2,2} = \{4, 5\} \subseteq S_{1,1} = \{1, 2, 3, 4, 5\}$$

We say that the precondition tree is well formed when it satisfies the three conditions we have defined above. In the rest of the discussion, we always assume that we are working with well-formed precondition trees. Given a column i, we define the guard for this column as the conjunction of the leaf that occupies the i-th column and all its ancestors. Let $\bar{v}' = v'_1, \ldots, v'_l$ be the state space vector. Let $\bar{c}_i$ be the i-th column of the expression table. Then, the meaning of the operation specification is:

$$\bar{v}' \in \{\bar{c}_i : 1 \leq i \leq m \wedge guard(i)\}$$

That is, the value of the state variables vector in the post-state can be equal to the value of any of the expression table columns whose guard was true in the pre-state.
The semantics of individual predicates and expressions is explained in [3].

Appendix 2: The algorithm that computes the *apply* function

We provide (in Sect. "Executable specification in Haskell") a Haskell executable specification of the *apply* function (cf. Sect. 4.2.1) and present (in Sect. "The Java implementation") its Java implementation. Note that in this paper all the tables resulting from delta-table application have been generated using this implementation.

Executable specification in Haskell

The executable specification of the algorithm that computes the *apply* function is provided as a program written in Haskell (Listing 13).

Haskell is a functional programming language with a syntax that strongly resembles the usual mathematical notation for defining function by cases, via pattern matching. The Haskell code and the comments in Listing 13 should be almost self-explanatory. In the following, we shortly explain some technicalities. The expression contained in an ordinary table cell and in a delta-table cell containing the insert or replace operators is represented using the standard library type **String** (Line 3). Both ordinary tables and delta tables are represented as trees, where each node corresponds to a cell. A tree that represents a table (ordinary or delta) is expressed as a value of the recursive data type Table (Line 10)—a value of type Table describes the root cell and the list of its immediate subtrees. The data type Table has five data constructors (each describes a different kind of node): Basic, Insert, and Replace (all have arity two), Match and Remove (arity one).[19] Ordinary tables are represented by using only the data constructor Basic, while delta tables are represented by using only the other data constructors, which correspond to the delta-table operators.

Example 3 (Representation of ordinary tables and delta tables in Haskell) Since ordinary tables may have more than one cell in their first row, they are always encoded by adding a top node containing the string "ROOT". Delta tables are therefore encoded by adding a top node containing the match operator (matching the top node in the ordinary tables). The following ordinary and delta tables

A	
C	D
1	2
i	ii

*			
▶I		▶J	
–	*	*	–
–	*	*▶X	–
–	*	*▶Y	–

are respectively represented by the following values of type Table:

```
Basic "ROOT" [Basic "A" [Basic "C" [Basic "1" [Basic "i" []]],
                         Basic "D"[Basic "2" [Basic "ii" []]]]]

Match [Match [Insert "I" [Remove [Remove     [Remove []]],
                          Match  [Match      [Match []]]],
              Insert "J" [Match  [Replace "X" [Replace "Y" []]],
                          Remove [Remove     [Remove []]]]]]
```

The Haskell function apply (Lines 15–64) corresponds to the *apply* function (introduced in Sect. 4.2.1) that executes the delta operations of the delta table. To improve readability, we have not modeled the behavior described in Remark 2 of Sect. 4.2.1.

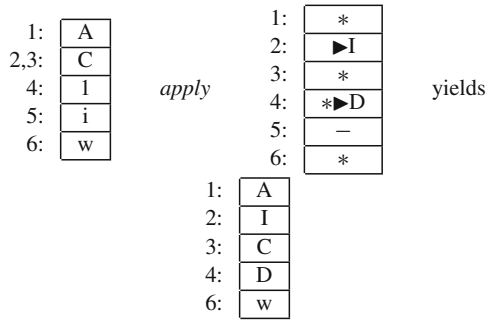
Line 30 declares the type of the apply function. The first argument of the function apply has type **Maybe** Table. The standard library **Maybe** data type has two constructors: the unary data constructor **Just** and the constant **Nothing**. It is used to specify optional values: A value of type **Maybe** Table either contains a table t (represented as **Just** t), or it is empty (represented as **Nothing**).

The standard library function **concat** takes two lists and returns their concatenation. The standard library function **zipWith** calls a given function pairwise on each member of both lists, returning a list. For the convenience of readers, **zipWith**'s code is as follows:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = (f x y) : zipWith f xs ys
```

Example 4 (An execution of the Haskell function apply) Consider the following delta-table application, which already has been presented at the end of Sect. 4.2.1 (recall that the numbers to the left of the cells denote the recursion step):

[19] Adding **deriving(Show)** at the end of a data declaration enables printing instances on standard output.



The above delta-table application can be defined by appending to the code in Listing 13 the following three lines:[20]

```
t = Basic "ROOT" [Basic "A" [Basic "C" [Basic "l" [Basic "i"
                                                   [Basic "w" []]]]]]
dt = Match [Match [Insert "I" [Match [Remove [Insert "D"
                                                  [Remove [Match []]]]]]]]
main = print (apply Nothing t dt)
```

Executing the main program yields

```
Basic "ROOT" [Basic "A" [Basic "I" [Basic "C" [Basic "D" [Basic "w" []]]]]]
```

The Java implementation

The Java implementation of the algorithm that computes the *apply* function is given in Listings 14 and 15. A cell of a table is represented by the Node class which extends the Vector class. Tables are represented as trees, and an object of class Node represents a cell, which is the root of a subtree, and contains (in the Vector's elements) the reference to the roots of the immediate subtrees.

The applyPrime method (Lines 16–23 in Listing 14) corresponds to the *apply′* function (introduced in Sect. 4.3). It first calls (Line 19) the prepare method (which applies the rules described in Sect. 4.3) and then invokes (Line 21) the apply method (Lines 30–64 in Listing 14). The apply method corresponds to the *apply* function (introduced in Sect. 4.2.1 and specified in Sect. 1). When calling the apply method, the this object is the current node of the delta table, and orig and res are the current nodes of the original table and of the resulting table, respectively.

The body of the apply method implements the recursive walking of the parse tree, which is controlled by the operations in the cells of the delta table and, therefore, describes how each operation works. The first part of the method (Lines 36–50) reads the operation of the current delta node and updates the content of the resulting node; Lines 40–46 implement the behavior described in Remark 2 of Sect. 4.2.1. The second part of the method (Lines 52–62) is responsible for the recursive call.[21] At each recursive call (Lines 43, 56, and 60), the method getOrCreateChild(i), which creates the

[20] Adding a definition for the name main makes Listing 13 a complete program.

[21] Line 37 is a special case, where ∗ matches multiple condition cells.

```
1  module Main where
2
3  type Expr = String
4
5  -- |A table is a tree that consists of the following nodes:
6  -- 1. Basic nodes that represent the expressions in Finefit tables.
7  -- 2. Match, Remove, Insert, and Replace nodes, that represent
8  --    the operations of the delta tables.
9
10 data Table = Basic Expr [Table]
11            | Match [Table] | Remove [Table] |
12              Insert Expr [Table] | Replace Expr [Table]
13           deriving(Show)
14
15 -- |We assume that the tables given to the apply function,
16 -- have a correct structure. For example, that each
17 -- table is a tree rooted at a dummy node, the original table has
18 -- only basic expression nodes, and so on.
19 -- The complete system checks that the tables have the correct
20 -- structure, adding these checks to the specification would
21 -- hinder understanding the essential ideas.
22 -- The apply function takes the following arguments:
23 -- 1. the parent of the node in the original table (or the constant
24 --    Nothing, if the node is the root),
25 -- 2. the node itself,
26 -- 3. the corresponding node in the delta table.
27 -- The function returns the sequence of nodes that corresponds
28 -- to the new table.
29
30 apply :: Maybe Table -> Table -> Table -> [Table]
31
32 -- |Some notes on the recursive specification of the function:
33 -- 1. We divide the specification into several cases by pattern
34 --    matching on the kinds of delta table nodes
35 --    (each kind of delta table operator node has its own case).
36 -- 2. In all cases, we take care to pass the correct parent for
37 --    the nodes in the recursive call.
38 -- Applying a delta table match operator to a node of the original
39 -- table, keeps the content of the original table, and
40 -- recursively applies the children of the match node,
41 -- to those of the original node.
42
43 apply _ node (Match deltas) =
44    [Basic (content node) (apply_to_children node deltas)]
45
46 -- |Applying a delta table remove to a node of the original table,
47 -- replaces the node by its children
48 -- (after they are transformed by the children of the delta table node).
49 -- This case is the reason why apply returns a list of tables.
50
51 apply _ node (Remove deltas) = apply_to_children node deltas
52
53 -- |Applying a delta table insert to a node of the original table,
54 -- inserts a new node as the child of the
55 -- original node's parent, and all of the parent's children
56 -- (including the original node) as the children of
57 -- the newly inserted node.
58
59 apply (Just parent) _ (Insert e' deltas) =
60   [Basic e' (apply_to_   children parent deltas)]
61
62 -- Replace is implemented in terms of remove followed by insert
63
64 apply p x (Replace e cs) = apply p x (Insert e [Remove cs])
65
66 -- |To process the children of a table node against a sequence
67 -- of delta table operators, we traverse both lists,
68 -- calling apply on each pair of corresponding nodes.
69 -- This generates a list of lists, that we then
70 -- concatenate to form the result.
71
72 apply_to_children parent deltas = concat (zipWith
73     (apply (Just parent)) (children parent) deltas)
74
75 children :: Table -> [Table]
76 children (Basic _ xs) = xs
77
78 content :: Table -> Expr
79 content (Basic c _) = c
```

Listing 13: Executable specification of the algorithm that computes the *apply* function introduced in Sect. 4.2.1; the behavior described in Remark 2 is not modeled

```
1  import java.util.Vector;
2
3  /* The Node class represents a cell of a table. Subnodes
4     represent cells below this cell. */
5
6  public class Node extends Vector<Node>{
7
8    public enum Op {nop,match,remove,insert,replace};
9    public Op op = Op.nop; //Default is no operation
10   public String val = " "; //Value or content of the cell
11   public boolean cond = false; //true if this node is a
12      condition node and not a value node
13
14   public Node(String val){this.val=val;}
15
16   /* "this" represents the root of the delta table. "orig" is
17      the root of the original table. */
18   public Node applyPrime(Node orig){  //corresponds to the function
     apply' of Section 4.3
19     prepare(orig, this);
20     Node result = new Node("<root>");
21     apply(orig, result);
22     return result;
23   }
24
25
26   public static void prepare(Node orig, Node delta){...}
27                                        //See Section 4.3
28
29
30   /* The parameters "this", "orig", and "res" represent the
31    current node (or cell) of the delta table, original table, and
32    resulting table, respectively. */
33   public void apply(Node orig, Node res){   /*corresponds to the
34     function apply of Section 4.2.1*/
35
36     switch(op){
37       case nop: break;
38       case remove:  res.op = Op.remove; //no break here
39       case match: res.val = unifyValue(orig.val);
40             if(isLastConditionNode() && !orig.isLastConditionNode())
41                { //see Remark 2 in Section 4.2.1
42               for(int i=0; i<orig.size(); i++){
43                 apply(orig.get(i), res.getOrCreateChild(i, cond));
44               }
45               return;
46             }
47             break;
48       case insert:  res.val = val; break;
49       case replace: res.val = val; break;
50     }
51
52     for(int i=0; i<size(); i++){
53       if(orig.size()==0 && get(i).op==Op.replace){  get(i).op=Op.insert;  }
54
55       if(get(i).op==Op.insert){
56          get(i).apply(orig, res.getOrCreateChild(i, cond));
57       }else{
58          if(orig.size()==0) break;
59          Node newOrig = (i<orig.size()? orig.get(i) : orig.lastElement() );
60          get(i).apply(newOrig, res.getOrCreateChild(i, cond));
61       }
62     }
63     res.deleteTemporaryRemoveNodes();
64   }  // end of method apply
65
66
67   ... // See Listing 15
68
69 } // end of class Node
```

Listing 14: Java implementation of the algorithms that compute the *apply'* and *apply* functions

```
1  /* Returns true if this node is a condition node and its child
2     nodes are non-condition (value) nodes.*/
3  public boolean isLastConditionNode(){
4    if(!cond) return false;
5    if(size()==0) return true;
6    if(get(0).cond) return false;  //For simplicity only one
7                                   //child is checked.
8    return true;
9  }
10
11 /* Returns the i'th subnode. If the subnode does not exist,
12                                   then it is create and returned. */
13 public Node getOrCreateChild(int i, boolean cond){
14   if(i<size()){
15     return get(i);
16   }else{
17     Node tmp = new Node(" ");
18     tmp.cond = cond;
19     tmp.op = Op.nop;
20     add(tmp);
21     return getOrCreateChild(i, cond);
22   }
23 }
24
25 /* Replaces each child of this node that is marked as
26    remove "-" by its grand children.
27    The algorithm operates recursively until no child is
28    marked with remove "-".*/
29 protected void deleteTemporaryRemoveNodes(){
30   boolean childRemoved=false;
31   do{
32     childRemoved=false;
33     for(int i=0; i<size();i++){
34       Node child = get(i);
35       if(child.op==Node.Op.remove){ //Replace the removed
36             child by its children
37         remove(i);
38         for(int j=i,k=0; k<child.size();k++,j++){
39           insertElementAt(child.get(k), j);
40         }
41         childRemoved = true;
42         break;
43       }
44     }
45   }while(childRemoved);
46 }
47
48 /* If the delta value is "*", then origVal is returned.
49    Otherwise all occurrences
50    of "*" in the delta value are replace with origVal.
51                                   E.g., "x" apply "(*)+y" yields "x+y".*/
52 public String unifyValue(String origVal){
53   if(val.equals("**"))
54     return origVal;
55   else
56     return val.replaceAll("(*)",origVal);
57 }
```

Listing 15: Helper methods of the apply method

nodes of the resulting table, is called; it either returns the i-th subnode if it already exists, or it creates the i-th subnode and returns it. We now continue illustrating the apply method using small examples. (An example of an execution trace of the method apply is provided in Example 5.)

The loop that begins in Line 52 iterates over the siblings of the current delta-table node. This means that if at the current level the original table has more cells than the delta table, then the additional cells are ignored. For example:

| A | B | C |

apply

| * | * |

yields

| A | B |

If the delta table has more cells than the original table at the current level, the last cell of the original table is repeatedly used when processing the additional operations of the delta table (Line 53). For example:

A	B

apply

∗	∗	∗

yields

A	B	B

If the original table contains no cells at the current level and the current operation of the delta table is ∗ or −, the remaining operations of the delta table on the current branch are ignored (Line 52).

A

apply

∗
∗

yields

A

If the original table contains no cells at the current level and the current operation of the delta table is ▶ or ∗▶, a cell with the value of the insert or replace operations is inserted (Line 50) in the resulting table. Line 47 maps in this case the replace operation to an insert operation. For example:

A

apply

∗
▶B
∗▶C

yields

A
B
C

It is therefore possible to extend a table with new variable and value rows by using either the insert or replace operation.

The algorithm traverses the original table and the delta table in parallel. The recursive call in Line 54 is responsible for the operations match, remove, and replace.[22] The recursive call in Line 50 handles the *insert* operation, where the current node of the original table is passed as the first argument rather than the current child of the original table. Delaying the recursive step on the original table results in the described semantics of the insert operation.

The recursive call in Line 37 is used for convenience. If either the remove or the match operation occurs as the last condition in a condition hierarchy, but the condition cell c of the original table has additional subconditions (this is checked by isLastConditionNode()), the remove or the match operation is applied to all subcondition of c. For example, let A, B, and C be condition cells, then

A	
B	C

apply

∗

yields

A	
B	C

The reader may have noticed that unlike the other operations, the remove operation is not handled in the loop in Lines 52–62. When a remove operation occurs, a temporary cell is created in the resulting table that is marked to be removed (Line 38). The marked cells are removed in Line 57. Removing nodes at the end of the recursion simplifies the implementation, because removing a node c pulls up its children $c_1, \ldots, c_n$ to the current level. Hence, by removing a node the number of siblings at the current level may decrease, stay constant, or increase. In the following example, we assume that A is not a condition cell.

A	B	
C	D	E

apply

∗	−

yields

A	D	E

Example 5 (An execution trace of the method apply) For a complete example consider the following delta-table application, which already has been presented at the end of Sect. 4.2.1 and in Example 4 of "Executable specification in Haskell" of appendix:

1:	A
2,3:	C
4:	1
5:	i
6:	w

apply

1:	∗
2:	▶I
3:	∗
4:	∗▶D
5:	−
6:	∗

yields

1:	A
2:	I
3:	C
4:	D
6:	w

Executing the method apply in (Lines 34–64) will result in the following recursive invocations of the method:

Recursion step	this	orig	res
0	< root >	< root >	< root >
1	∗	A	A
2	▶I	C	I
3	∗	C	C
4	∗▶D	1	D
5	−	i	−
6	∗	w	w

At the end of the last recursion step of the apply algorithm (Line 57 in Listing 14), the method deleteTemporaryRemoveNodes is called which removes the temporary "−" node from the resulting tree res that is introduced at recursion step 5.

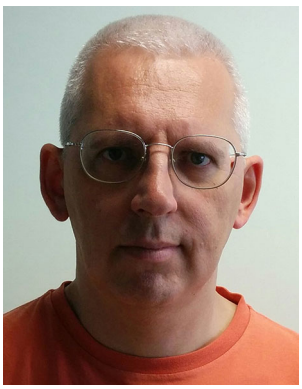
References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison Wesley Longman, Boston (2001)
2. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering-Foundations, Principles, and Techniques. Springer, Berlin (2005)
3. Faitelson, D., Tyszberowicz, S.S.: Data refinement based testing. Int. J. Syst. Assur. Eng. Manag. **2**(2), 144–154 (2011)
4. de Roever, W.P., Engelhardt, K.: Data Refinement: Model-Oriented Proof Theories and Their Comparison. Cambridge Tracts in Theoretical Computer Science, vol. 46. Cambridge University Press, Cambridge (1998)
5. The FineFit home page. https://github.com/coderocket/finefit

[22] If *replace* is not substituted by *insert* in Line 47.

6. Parnas, D.L.: Tabular representation of relations. Tech. rep. 260, Research Institute of Ontario, McMaster University (1992)
7. Bettini, L., Damiani, F., Schaefer, I.: Compositional type checking of delta-oriented software product lines. Acta Inf. **50**, 77–122 (2013)
8. Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Software Product Line Conference (SPLC), LNCS, vol. 6287, pp. 77–91. Springer (2010)
9. Apel, S., Batory, D.S., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines: Concepts and Implementation. Springer, Berlin (2013)
10. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling step-wise refinement. IEEE TSE **30**(6), 355–371 (2004)
11. Schaefer, I., Damiani, F.: Pure delta-oriented programming. In: Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development, FOSD'10, pp. 49–56. ACM, New York, NY, USA (2010). doi:10.1145/1868688.1868696
12. Koscielny, J., Holthusen, S., Schaefer, I., Schulze, S., Bettini, L., Damiani, F.: Deltaj 1.5: delta-oriented programming for java 1.5. In: Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ'14, pp. 63–74. ACM, New York, NY, USA (2014). doi:10.1145/2647508.2647512
13. The DeltaJ home page. https://www.tu-braunschweig.de/isf/research/deltas
14. Johansen, M.F., Haugen, O., Fleurey, F.: Properties of realistic feature models make combinatorial testing of product lines feasible. In: Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 638–652. Springer, Berlin (2011)
15. Johansen, M.F., Haugen, O., Fleurey, F.: An algorithm for generating t-wise covering arrays from large feature models. In: Proceedings of the 16th International Software Product Line Conference, vol. 1, SPLC'12, pp. 46–55. ACM, New York, NY, USA (2012). doi:10.1145/2362536.2362547
16. Kowal, M., Schulze, S., Schaefer, I.: Towards efficient spl testing by variant reduction. In: Proceedings of the 4th International Workshop on Variability & Composition, VariComp'13, pp. 1–6. ACM, New York, NY, USA (2013). doi:10.1145/2451617.2451619
17. Lochau, M., Goltz, U.: Feature interaction aware test case generation for embedded control systems. Electron. Notes Theor. Comput. Sci. **264**(3), 37–52 (2010)
18. The DeltaFineFit home page. http://di.unito.it/deltafinefit
19. Damiani, F., Gladisch, C., Tyszberowicz, S.: Refinement-based testing of delta-oriented product lines. In: Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ'13, pp. 135–140. ACM, New York, NY, USA (2013). doi:10.1145/2500828.2500841
20. Jackson, D.: Software Abstractions-Logic, Language, and Analysis. MIT Press, Cambridge (2012)
21. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice Hall International, Upper Saddle River (2001)
22. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 4424, pp. 632–647. Springer (2007)
23. Mugridge, R., Cunningham, W.: Fit for Developing Software: Framework for Integrated Tests. Prentice Education Inc., New Jersy (2005)
24. Batory, D.: Feature models, grammars, and propositional formulas. In: Proceedings of the International Conference on Software Product Lines (SPLC), LNCS, vol. 3714, pp. 7–20. Springer (2005)
25. Schaefer, I., Bettini, L., Damiani, F.: Compositional type-checking for delta-oriented programming. In: Proceedings of the Tenth International Conference on Aspect-oriented Software Development, AOSD'11, pp. 43–56. ACM, New York, NY, USA (2011). doi:10.1145/1960275.1960283
26. Schaefer, I., Rabiser, R., Clarke, D., Bettini, L., Benavides, D., Botterweck, G., Pathak, A., Trujillo, S., Villela, K.: Software diversity: state of the art and perspectives. Int. J. Softw. Tools Technol. Transf. **14**(5), 477–495 (2012)
27. Apel, S., Kästner, C., Grösslinger, A., Lengauer, C.: Type safety for feature-oriented product lines. Autom. Softw. Eng. **17**(3), 251–300 (2010)
28. Apel, S., Kästner, C., Lengauer, C.: Feature featherweight java: a calculus for feature-oriented programming and stepwise refinement. In: Proceedings of the 7th International Conference on Generative Programming and Component Engineering, GPCE'08, pp. 101–112. ACM, New York, NY, USA (2008). doi:10.1145/1449913.1449931
29. Delaware, B., Cook, W.R., Batory, D.: Fitting the pieces together: A machine-checked model of safe composition. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE'09, pp. 243–252. ACM, New York, NY, USA (2009). doi:10.1145/1595696.1595733
30. Krueger, C.: Eliminating the adoption barrier. IEEE Softw. **19**(4), 29–31 (2002)
31. do Carmo Machado, I., McGregor, J.D., Cavalcanti, Y.C., de Almeida, E.S.: On strategies for testing software product lines: a systematic literature review. Inf. Softw. Technol. **56**(10), 1183–1199 (2014)
32. Engström, E., Runeson, P.: Software product line testing–a systematic mapping study. Inf. Softw. Technol. **53**(1), 2–13 (2011)
33. Lee, J., Kang, S., Lee, D.: A survey on software product line testing. In: Proceedings of the 16th International Software Product Line Conference, vol. 1, SPLC'12, pp. 31–40. ACM, New York, NY, USA (2012). doi:10.1145/2362536.2362545
34. da Mota Silveira Neto, P.A., do Carmo Machado, I., McGregor, J.D., de Almeida, E.S., de Lemos Meira, S.R.: A systematic mapping study of software product lines testing. Inf. Softw. Technol. **53**(5), 407–423 (2011). Special Section on Best Papers from XP2010
35. Salem, K., Beyer, K., Lindsay, B., Cochrane, R.: How to roll a join: asynchronous incremental view maintenance. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD'00, pp. 129–140. ACM, New York, NY, USA (2000). doi:10.1145/342009.335393
36. Uzuncaova, E., Khurshid, S., Batory, D.S.: Incremental test generation for software product lines. IEEE TSE **36**(3), 309–322 (2010)
37. Lochau, M., Lity, S., Lachmann, R., Schaefer, I., Goltz, U.: Delta-oriented model-based integration testing of large-scale systems. J. Syst. Softw. **91**, 63–84 (2014)
38. Lochau, M., Schaefer, I., Kamischke, J., Lity, S.: Incremental model-based testing of delta-oriented software product lines. In: TAP, LNCS, vol. 7305, pp. 67–82. Springer (2012)
39. Damiani, F., Owe, O., Dovland, J., Schaefer, I., Johnsen, E.B., Yu, I.C.: A transformational proof system for delta-oriented programming. In: Proceedings of the 16th International Software Product Line Conference, vol. 2, SPLC'12, pp. 53–60. ACM, New York, NY, USA (2012). doi:10.1145/2364412.2364422
40. Hähnle, R., Schaefer, I.: A Liskov principle for delta-oriented programming. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change International Symposium (ISoLA), Part I, LNCS, vol. 7609, pp. 32–46. Springer (2012)
41. Dovland, J., Johnsen, E.B., Owe, O., Steffen, M.: Lazy behavioral subtyping. J. Log. Algebr. Program. **79**(7), 578–607 (2010)
42. Beckert, B., Gladisch, C., Tyszberowicz, S., Yehudai, A.: KeYGenU: combining verification-based and capture and replay tech-

niques for regression unit testing. Int. J. Syst. Assur. Eng. Manag. **2**(2), 97–113 (2011)
43. Damiani, F., Padovani, L., Schaefer, I.: A formal foundation for dynamic delta-oriented software product lines. In: Proceedings of the 11th International Conference on Generative Programming and Component Engineering, GPCE'12, pp. 1–10. ACM, New York, NY, USA (2012). doi:10.1145/2371401.2371403
44. Damiani, F., Schaefer, I.: Dynamic delta-oriented programming. In: Proceedings of the 15th International Software Product Line Conference, vol. 2, SPLC'11, pp. 34:1–34:8. ACM, New York, NY, USA (2011). doi:10.1145/2019136.2019175
45. Li, Z., Harman, M., Hierons, R.M.: Search algorithms for regression test case prioritization. IEEE TSE **33**(4), 225–237 (2007)



Ferruccio Damiani is associate professor at the Computer Science Department of the University of Torino. There he founded and coordinates the MoVeRe (System Modeling, Verification, and Reuse) research group. He received his Ph.D. in 1998 from the same university. His research interests include: computational models and languages; concurrent, distributed, and mobile systems; domain-specific languages; static and dynamic analysis techniques; system evolution and dynamic software updates; variability modeling and software product lines. He is author/coauthor of more than 70 papers on these topics.

David Faitelson received his B.Sc. (1992–1995) in Computer Science & Mathematics from Tel Aviv University. He then pursued a career as a software engineer and entrepreneur for 15 years. He has received his M.Sc. (2001–2004) and Ph.D. (2005–2008) in Computer Science from the University of Oxford. After a postdoc position at the Technion (2010), he moved to the Tel Aviv Academic College of Engineering, where he is a senior lecturer at the Software Engineering Department. The focus of his research is the application of formal methods to software engineering.

Christoph Gladisch is a research engineer at the Center for Research and Advance Engineering of Robert Bosch GmbH. He studied Computer Visualistics at University of Koblenz-Landau, received a PhD at Karlsruhe Institue of Technology (KIT), and was the head of a Young Investigator Group at KIT. His research topics include logics, formal methods, model-based testing of embedded systems, software variability modeling, and software evolution techniques.

Shmuel Tyszberowicz studied mathematics and computer sciences. He is an associate professor at the school of computer science at the Academic College of Tel Aviv Yaffo, Israel. His research interests include formal methods for software engineering, reliable software evolution, systematic design modularity, static and dynamic analysis techniques, integrated approach to software product-line development, creating tools and techniques for high-quality software development, formal methods for developing reactive systems.