Project N°: **FP7-231620**

Project Acronym: **HATS**

Project Title: **Highly Adaptable and Trustworthy Software using Formal Models**

Instrument: **Integrated Project**

Scheme: **Information & Communication Technologies**

Future and Emerging Technologies

Deliverable D2.4
Types for Variability

Due date of deliverable: (T36)

Actual submission date: 1st March 2012

Start date of the project: **1st March 2009** Duration: **48 months**

Organisation name of lead contractor for this deliverable: **BOL**

Final version

Integrated Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
Types for Variability

This document summarises deliverable D2.4 of project FP7-231620 (HATS), an Integrated Project supported by the 7th Framework Programme of the EC within the FET (Future and Emerging Technologies) scheme. Full information on this project, including the contents of this deliverable, is available online at `http://www.hats-project.eu`.

This deliverable reports on the work done within Task **T2.4**. The goal of this task is to study and design type systems for dealing with variability issues.

We report on type systems for Delta-oriented Programming, presenting two different complementary approaches which guarantee safety of the final product (the result of the application of the deltas) and conflict-freedom. Moreover, type systems for concurrency related errors are discussed, in particular a type system for Abstract Failure Models, designed to ensure that server side abstract failures are caught by the client and that client side failures are caught by the server.

List of Authors

Einar Broch Johnsen (UIO)
Dave Clarke (KUL)
Elena Giachino (BOL)
Ivan Lanese (BOL)
Michael Lienhardt (BOL)
Davide Sangiorgi (BOL)
Ina Schaefer (CTH)
Martin Steffen (UIO)
Gianluigi Zavattaro (BOL)

Contents

Chapter 1

Introduction

The task addresses the objective of achieving lightweight verification mechanisms by means of type systems. The challenge here is to take into account the novel features of the ABS language, notably features related to variability, dynamicity, and use of components.

The deliverable is structured as follows: This chapter provides an introduction to the HATS reasearch on types for variability. It gives an overview of the main concepts of Delta Oriented Programming (Section 1.1), the issues related to design type systems for it (Section 1.2), and Abstract Failure Models (Section 1.3). The remaining three chapters present the specific contributions of this task in more detail: Chapters 2 and 3 describe two different approaches for designing type systems for Delta Oriented Programming, Chapter 4 presents a type system for Abstract Failure Models.

1.1 Delta Oriented Programming

As pointed out in HATS deliverable **D2.2.b** [10] our approach to variability modeling is based on delta models, which integrates the advantages coming from feature modeling, namely abstraction and documentation, with model refinement, by bridging the gap between features and code.

A *software product line* (SPL) is a set of software systems with well-defined commonalities and variabilities [7]. *Delta-oriented programming* (DOP) [29, 31] is a flexible compositional approach for implementing software product lines. It relies on the notion of program deltas [32, 28], a term that was first used in [23] to describe the modifications of object-oriented programs. The implementation of a delta-oriented product line consists of a *code base* and a *product line declaration*. The code base comprises a set of delta modules describing modifications of object-oriented programs that are necessary to generate all possible product implementations. A delta module can add classes, remove classes or modify classes by changing the class structure. The product line declaration provides the connection between the delta modules and the product members of a product line that are usually determined in terms of product features [16]. In the product declaration, it is specified for which feature configurations a delta module has to be used and in which order the delta modules that are applicable for a particular feature configuration are applied to generate the corresponding product. Separating the application conditions and delta module ordering from the definition of the delta modules increases the reusability of the delta modules, making it possible to develop different product lines by sharing delta modules.

Delta-oriented programming is an extension of *feature-oriented programming* (FOP) [5, 26, 3], a prominent compositional approach for implementing software product lines, by explicit operations to remove classes, methods or fields from a program. In [31], we have shown that FOP can indeed be embedded straightforwardly into DOP. While in FOP a product line implementation always starts from base feature modules comprising common core functionalities, in DOP, any product can be chosen as starting point of product generation. Hence, DOP supports proactive product line development, where all products are planned in advance, as well as proactive and extractive product line development [20], where development starts from an initial product line or existing legacy products. Moreover, the application conditions associ-

ated to the delta modules allow handling combinations of features explicitly providing a way to counter the optional feature problem [17].

1.2 Type Systems for DOP

The flexibility of DOP to start product generation from any complete or partial product as well as the expressiveness to handle feature combinations explicitly makes it challenging to ensure that for every valid feature configuration a unique product can be generated and that the SPL is type safe. A SPL is called *type safe* if all possible products are well-typed programs according to the type system of the programming language used to implement the products. In principle, the type safety of a SPL could be checked by generating and type-checking all products separately. A major drawback of this approach is that it is hard for the programmer to understand which delta module causes an error, based on the results while type-checking a single product.

Therefore, a **first requirement** for a type system for delta-oriented product lines is that it allows ensuring the type safety of the SPL without generating and inspecting the code of all possible products. The separation of application conditions and delta module ordering from the definition of the delta modules allows reusing delta modules across different product lines. Hence, a **second requirement** for the design of a type system for DOP is that each delta module can be analyzed in isolation without relying on global information of the product line. This facilitates to reuse the analysis results also across different product lines. Furthermore, if one delta module in a product line implementation changes, only re-checking of the changed module is required.

We present in the following two approaches for designing type systems for DOP:

1. Constraint-based type systems, guaranteeing safety of the final product (the result of the application of the deltas). See Chapter 2.

2. A type system based on row-types, which is lighter but only expresses the effects of applications of deltas and conflict-freedom. See Chapter 3.

1.3 Deployment Variability and Abstract Failure Models

Another aspect of variability in ABS that has been studied and reported in the HATS deliverable **D2.1** [11] is *deployment variability*. Deployment variability is concerned with the lowest levels of a flexible architecture upon which to base ABS models. The mentioned deliverable reports on different aspects of deployment modeling. In particular, *Abstract Failure Models* are presented as a way to extend the use of futures in ABS to propagate failures. Futures are used in Core ABS to store reply values to asynchronous method calls. In the context of concurrent objects in ABS, a failure model inspired from web-services is introduced, where abstract failures may be inserted at both the client and server side of a method call. The future is perceived as a two-way communication channel related to the specific method call, used to communicate the occurrence of failures (in addition to regular replies to method calls). This approach can be used to define failures at the server side (e.g., memory exhaustion) but also to kill a request from the client side. The approach deals with abstract failures and their corresponding compensations.

In this deliverable we report about the type system designed to ensure that server side abstract failures are caught by the client and that client side failures are caught by the server. See Chapter 4.

1.4 List of Papers Comprising Deliverable D2.4

This section lists all the papers that comprise this deliverable, indicates where they were published, and explains how each paper is related to the main text of this deliverable. The papers are not directly attached to Deliverable **D2.4**, but are made available on the HATS web site at the following url: `http://www.hats-project.eu/sites/default/files/D2.4`. Direct links are also provided for each paper listed below.

Paper 1: Compositional Type-Checking for Delta-Oriented Programming

This paper [30] provide a foundation for compositional type checking of delta-oriented product lines by presenting a min- imal core calculus for delta-oriented programming. The calculus is equipped with a constraint-based type system that allows analyzing each delta module in isolation, such that also the results of the analysis can be reused. By combining the analysis results for the delta modules with the product line declaration it is possible to establish that all the products of the product line are well-typed according to the type system. The intuitions behind this work are explained in Chapter 2.

This paper was written by Ina Schaefer, Lorenzo Bettini, and Ferruccio Damiani, and it was published in the proceedings of AOSD 2011.

Download Paper 1.

Paper 2: Row Types for Delta-Oriented Programming

This paper [22] presents another approach towards the typing of delta-oriented programs based on row polymorphism, as presented in Chapter 3.

This paper was written by Michael Lienhardt and Dave Clarke, and it will be published in the proceedings of VAMOS 2012.

Download Paper 2.

Paper 3: Safe Locking for Multi-Threaded Java with Exceptions

This paper [15] develops a static type and effect system to prevent the mentioned lock errors for a formal, object-oriented calculus which supports non-lexical lock handling and exceptions, as presented in Section 4.1. The role of this work within Task **T2.4** has been of a foundational background study to understand type systems for exception and lock handling. The result of this work represents a base for the following work on types for abstract failure models in ABS.

This paper was written by Einar Broch Johnsen, Thi Mai Thuong Tran, Olaf Owe, and Martin Steffen, and it has been accepted for publication in the Journal of Logic and Algebraic Programming.

Download Paper 3.

Paper 4: Fault in the Future

This paper [14] extends ABS with abstract failure models which allows the modeller to define failures. The paper takes an approach inspired from web services to define failure handling for concurrent objects, and defines a type system which ensures that all failures are handled. They also present the type system that is explained in Section 4.2.

This paper was written by Einar Broch Johnsen, Ivan Lanese, and Gianluigi Zavattaro, and was published in the proceedings of COORDINATION 2011.

Download Paper 4.

1.5 Other Type Systems

In the HATS Description of Work, where the intentions for designing type systems were described, we also mentioned behavioral types and types for components.

Behavioral contracts for deadlock analysis in ABS have been designed and they are part of the HATS Task **T4.3** *Correctness*. Therefore the result will appear in HATS Deliverable **D4.3**.

We have not started work on type systems for HATS components, which is announced in the HATS Description of Work, because the component model has undergone several changes recently. We plan to initiate this line in the next few months, as the component model is getting stable. The result will appear in HATS Deliverable **D3.3**.

Chapter 2

Compositional Type Checking of Delta-oriented Software Product Lines

This chapter summarizes the compositional constraint-based type system for type checking delta-oriented software product lines as published in [30]. An extended version of this paper containing all proofs is currently submitted for journal publication.

2.1 Related Work

The calculus Lightweight Feature Java (LFJ) [8], based on LJ (Lightweight Java) [33], provides a formalization of FOP [5, 26, 3] together with a constraint-based type system that satisfies the two design requirements described in Section 1.2. The calculus Featherweight Feature Java for Product Lines (FFJ_{PL}) [1], based on FJ [13], comprises an (independently developed) type system for FOP that does not meet the second requirement. These approaches for ensuring type safety of feature-oriented product lines are not straightforwardly adaptable to deal with the additional flexibility provided by DOP.

2.2 Results

We address the requirements described in Section 1.2 by developing Imperative Featherweight Delta Java (IFΔJ) a core calculus for DOP of product lines of Java programs, based on IFJ (an imperative variant of FJ [13]) that is used as underlying programming language for the products. An IFJ program consists of a *class table* `CT`, that is, a mapping from class names to class definitions. As first step towards a compositional type system for DOP, we define a constraint-based type system for IFJ that infers a set of *class constraints* $\mathcal{C}$ for an IFJ program `CT`. These constraints can be checked against the *class signature table* of `CT` (i.e., the program `CT` deprived from method bodies) in order to establish whether `CT` is a well-typed IFJ program. The pair $\langle \mathit{signature}(\texttt{CT}), \mathcal{C} \rangle$ represents an abstract representation of the program `CT` that can be used to establish whether `CT` is type safe.

In the second step, we define an abstract representation of the delta modules in IFΔJ, that can be inferred by analyzing each delta module in isolation and allows deriving abstract representations of the possible products directly without generating and inspecting their code (first requirement). The abstraction of a delta module δ consists of a pair $\langle \mathit{signature}(\delta), \mathcal{D}_\delta \rangle$, where $\mathit{signature}(\delta)$ is the *delta module signature* and $\mathcal{D}_\delta$ is a set of *delta clause-constraints*. The signature of a delta module δ is the analogue of a class signature for a delta module, i.e., a representation of the delta module without method bodies. It can be inferred by a straightforward inspection of the code of the delta module. The delta clause-constraints are inferred by a constraint-based type system for IFΔJ that analyzes each delta module δ in isolation.

A difficulty to ensure type safety by considering only the code of the delta modules is that delta modules are incomplete and only define differences to existing products. Thus, a type system for delta-oriented product lines that only requires to inspect the code of each delta module in isolation (second requirement)

has to explicitly capture the expectations a delta module has on the context in which it is applied. The delta clause-constraints express exactly these expectations of a delta module during delta module application and can be reused across different product lines, as they only depend on the delta module itself. The inferred sets of delta clause-constraints are organized such that the set of class constraints required to establish type safety of a product can be derived directly without generating and inspecting the code of the product. Therefore, type safety of a product line can be established only from the delta module signatures, the delta clause-constraints and the product line declaration.

Checking type safety of a DOP product line is linear in the number of its products (which may be exponential in the number of features). We expect that generating the constraints and performing the associated checks will be more efficient than generating all products and checking them by a JAVA compiler. Moreover, in the latter case it would be hard for the programmer to determine which delta module causes an error from the result of the JAVA compiler. Instead, the constraints collected by the IFΔJ type system can be exploited to show which part of a delta module generated the error. The idea (not formalized in current presentation of the type system) is to keep track of the location of the code in a delta module for each generated constraint.

Typing of the ABS language. The type system is designed for Java, but it could be easily adapted to the sequential part of the ABS. This sequential subset is pretty similar to Java and even simpler since there is no inheritance. For the concurrent part, the constraint-based type system needs to be changed to deal with futures, asynchronous calls, etc. But the extension for deltas should then follow the same principles as presented in this chapter and in [30]. We plan to examine this when considering the integration of this type techniques with the type techniques in Chapter 3. The work will be done as part of Task **T1.5** *Integrated Tool Platform.*

2.2.1 Example

In order to illustrate the main concepts, we use a variant of the *expression product line* (EPL) as described in [23]. We consider the following grammar:

```
Exp  ::=  Lit | Add | Neg
Lit  ::=  <non-negative integers>
Add  ::=  Exp "+" Exp
Neg  ::=  "-" Exp
```

Two different operations can be performed on the expressions described by this grammar: printing, which returns the expression as a string, and evaluating, which returns the value of the expression. The products in the EPL can be described by two feature sets, the ones concerned with data Lit, Add, Neg and the ones concerned with operations Eval and Print. Lit and Print are mandatory features. The features Add, Neg and Eval are optional.

Figure 2.1 contains a delta module for introducing an existing legacy product, realizing the features Lit, Add and Print. Figure 2.2 contains the delta modules for adding the evaluation functionality to the classes `Lit` and `Add`. Figure 2.3 contains the delta modules for incorporating the Neg feature by adding and modifying the class `Neg` and for adding glue code required by the two optional features Add and Neg to cooperate properly. Figure 2.4 contains the delta module for removing the Add feature from the legacy product. Figure 2.5 shows a product line declaration for the EPL. The order of delta module application is defined by an ordered list of the delta module sets which are enclosed by `[ .. ]`.

In order to obtain a product for a particular feature configuration, the modifications specified in the delta modules with valid application conditions are applied incrementally to the previously generated product. The first delta module is applied to the empty product. The modifications of a delta model are applicable to a (possibly empty) product if each class to be removed or modified exists and, for every modified class, if each method or field to be removed exists, if each method to be modified exists and has the same header as the modified method, and if each class, method or field to be added does not exist. During the generation

```
delta DLitAddPrint{
   adds class Exp extends Object {  // only used as a type
       String toString() { return ""; }
   }
   adds class Lit extends Exp {
      int value;
      Lit setLit(int n) { value = n; return this; }
      String toString() { return value + ""; }
   }
   adds class Add extends Exp {
      Exp expr1;
      Exp expr2;
      Add setAdd(Exp a, Exp b) { expr1 = a; expr2 = b; return this; }
      String toString() { return  expr1.toString() + "␣+␣" + expr2.toString(); }
   }
}
```

Figure 2.1: Delta module introducing a legacy product

```
delta DLitEval {
   modifies Exp {
      adds int eval() { return 0; }
   }
   modifies Lit {
      adds int eval() { return value; }
   }
}

delta DAddEval {
   modifies Add {
      adds int eval() { return expr1.eval() + expr2.eval(); }
   }
}
```

Figure 2.2: Delta modules for the Eval feature

of a product, every delta module must be applicable. Otherwise, the generation of the product fails. In particular, the first delta module that is applied can only contain additions.

Product Generation The generation of a product for a given feature configuration consists of two steps, performed automatically: (*i*) Find all delta modules with a valid application condition; and (*ii*) Apply the selected delta modules to the empty product in any linear ordering that respects the total order on the partition of the delta modules. If two delta modules add, remove or modify the same class, the ordering in which the delta modules are applied may influence the resulting product. However, for a product line implementation, it is essential to guarantee that for every valid feature configuration exactly one product is generated. This property is called *unambiguity* of the product line. A sufficient condition for unambiguity is that each part in the partition of the set of delta modules is *consistent*, that is, if one delta module in a part adds or removes a class, no other delta module in the same part may add, remove or modify the same class, and the modifications of the same class in different delta modules in the same part have to be disjoint. Defining the order of delta module application by a total ordering on a delta module partition provides an efficient way to ensure unambiguity, since only the consistency within each part has to be checked. Figure 2.6 depicts the product generated when the Lit, Neg, Print and Eval features are selected.

```
delta DNeg  {
   adds class Neg extends Exp {
      Exp expr;
      Neg setNeg(Exp a) { expr = a; return this; }
   }
}

delta DNegPrint {
   modifies Neg {
      adds String toString() { return "-" +  expr.toString(); }
   }
}

delta DNegEval{
   modifies Neg {
      adds int eval() { return (-1) * expr.eval(); }
   }
}

delta DAddNegPrint {
   modifies Add {
      modifies toString { return "(" +  original + ")"; }
   }
}
```

Figure 2.3: Delta modules for Neg, Print and Eval features

```
delta DremAdd {
   removes Add
}
```

Figure 2.4: Delta module removing the Add feature

2.2.2 Constraint-based Type System for IFJ

In this section, we present a constraint based type system for IFJ that infers a set of class constraints $\mathcal{F}$ for a given program CT. These constraints can then be checked against the class signature table *signature*(CT) in order to establish whether CT is a well-typed IFJ program.

Flat constraints, illustrated below, involve the type $\bot$, class names and type variables. Type variables, ranged over by α, β and γ, will be instantiated to class names when checking the constraints. The metavariable η denotes either a class name or a type variable, while the metavariable τ denotes either the type $\bot$ (the type of `null`), or a class name, or a type variable.

class(C)	class C must be defined
subtype(τ, η)	τ must be a subtype of η
cast$(\mathtt{C}, \tau)$	type τ must be castable to C
field$(\eta, \mathtt{f}, \alpha)$	class η must define or inherit field f of type α
meth$(\eta, \mathtt{m}, \overline{\alpha} \to \beta)$	class η must define or inherit method m of type $\overline{\alpha} \to \beta$

A *class signature table* CST is a mapping from class names to class signatures. The checking judgment for flat constraints is $\mathtt{CST} \models \mathcal{F}$, to be read "the constraints in the set of flat constraints $\mathcal{F}$ are satisfied with respect to the class signature table CST".

The constraint-based typing rules for IFJ organize the inferred constraints in a two-level hierarchy, corresponding to the structure of the class table of the IFJ program. Namely: (i) the typing rules infer a

```
features Lit, Add, Neg, Print, Eval
configurations Lit & Print
deltas
   [ DLitAddPrint,
     DNeg when Neg ]

   [ DLitEval when Eval,
     DNegPrint when Neg,
     DNegEval when (Neg & Eval),
     DremAdd when !Add ]

   [ DAddEval when (Add & Eval),
     DAddNegPrint when (Add & Neg) ]
```

Figure 2.5: Specification of the EPL

```
class Exp extends Object {
   String toString() { return ""; }
   int eval() { return 0; }
}
class Lit extends Exp {
   int value;
   Lit setLit(int n) { value = n; return this; }
   String toString() { return value + ""; }
   int eval() { return value; }
}
class Neg extends Exp {
   Exp expr;
   Net setNeg(Exp a) { expr = a; return this; }
   String toString() { return "-" +  expr.toString(); }
   int eval() { return (-1) * expr.eval(); }
}
```

Figure 2.6: Generated code for Lit, Neg, Print and Eval features

set of *class constraints* (one for each class definition in the program); (*ii*) each class constraint consists of the name of the respective class `C` and of a set of *method constraints* inferred for the methods defined in the class; and (*iii*) each method constraint consists of the name of the respective method and of the set of flat constraints inferred for the body of the method. Thus, a set of class constraints $\mathcal{C}$ can be understood as a mapping from class names to class constraints, and a class constraint can be understood as a mapping from method names to method constraints. The syntax of class constraints and method constraints is as follows:

Method constraints:

`m` **with** $\mathcal{F}$ method `m` has the set of flat constraints $\mathcal{F}$

Class constraints:

`C` **with** $\mathcal{K}$ class `C` has the set of method constraints $\mathcal{K}$

The constraint-based typing judgment for programs is $\vdash$ `CT` : $\mathcal{C}$, to be read "program `CT` has the class constraints $\mathcal{C}$".

The following example illustrates the constraint-based type system by considering an encoding in IFJ of methods from the EPL example introduced in Section 2.2.1.

Example 1 The sequential composition of expressions, "$\mathtt{e}_1;\mathtt{e}_2;\mathtt{e}_3$", which is not part of the IFJ syntax, can be encoded as "`new Encode().sc3C(`$\mathtt{e}_1,\mathtt{e}_2,\mathtt{e}_3$`)`" where `C` is the type of $\mathtt{e}_3$ and the class `Encode` defines the method

```
C sc3C(Object x1, Object x2, C x3) { return x3; }
```

Therefore, the method `setAdd` of class `Add` introduced in Figure 2.1 can be encoded in IFJ as follows:

```
Add setAdd(Exp a, Exp b) {
 return new Encode().sc3Add(this.expr1=a, this.expr2=b, this); }
```

The inferred method constraint is

$$\texttt{setAdd}\ \textbf{with}\ \left\{\begin{array}{l}\textbf{field}(\texttt{Add},\texttt{expr1},\alpha'),\ \textbf{subtype}(\texttt{Exp},\alpha'),\\ \textbf{field}(\texttt{Add},\texttt{expr2},\alpha''),\ \textbf{subtype}(\texttt{Exp},\alpha''),\\ \textbf{subtype}(\texttt{Exp},\texttt{Object}),\ \textbf{subtype}(\texttt{Add},\texttt{Add}),\ \textbf{class}(\texttt{Encode}),\\ \textbf{meth}(\texttt{Encode},\texttt{sc3C},(\texttt{Object},\texttt{Object},\texttt{Add}\to\texttt{Add}))\end{array}\right\}$$

Some optimizations (not considered in this paper) are possible. Constraints like **subtype**(`Exp`, `Object`) and **subtype**(`Add`, `Add`) can be dropped. Information about standard library classes, like `Encode`, that cannot be modified by the delta modules can be exploited to infer simpler constraints. For example, the following simpler constraints could be inferred:

$$\texttt{setAdd}\ \textbf{with}\ \left\{\begin{array}{l}\textbf{field}(\texttt{Add},\texttt{expr1},\alpha'),\ \textbf{subtype}(\texttt{Exp},\alpha'),\\ \textbf{field}(\texttt{Add},\texttt{expr2},\alpha''),\ \textbf{subtype}(\texttt{Exp},\alpha'')\end{array}\right\}$$

The method `eval` introduced in class `Add` by the delta module `DAddEval` in Figure 2.2 can be encoded in IFJ as follows:

```
Int eval() { return this.expr1.eval().sum(this.expr2.eval()); }
```

where `Int` is the class for integers. The inferred class constraint is

$$\texttt{eval}\ \textbf{with}\ \left\{\begin{array}{l}\textbf{field}(\texttt{Add},\texttt{expr1},\beta'),\ \textbf{meth}(\beta',\texttt{eval},(\bullet\to\gamma'')),\\ \textbf{field}(\texttt{Add},\texttt{expr2},\beta''),\ \textbf{meth}(\beta'',\texttt{eval},(\bullet\to\gamma')),\\ \textbf{meth}(\gamma',\texttt{sum},(\gamma''\to\gamma''')),\ \textbf{subtype}(\gamma''',\texttt{Int})\end{array}\right\}$$

Assuming that `Int` is a standard library final class, it would be possible to infer a simpler constraint (namely, replace γ''' by `Int` and drop **subtype**(γ''', `Int`)). Further optimizations are possible in presence of primitive types (not formalized in IFJ). For instance, given the original version of method `eval` (that uses the primitive type `int`)

```
int eval() { return this.expr1.eval() + this.expr2.eval(); }
```

it would be possible to infer the simpler method constraint

$$\texttt{eval}\ \textbf{with}\ \left\{\begin{array}{l}\textbf{field}(\texttt{Add},\texttt{expr1},\beta'),\ \textbf{meth}(\beta',\texttt{eval},(\bullet\to\texttt{int})),\\ \textbf{field}(\texttt{Add},\texttt{expr2},\beta''),\ \textbf{meth}(\beta'',\texttt{eval},(\bullet\to\texttt{int}))\end{array}\right\}$$

2.2.3 Constraint-based Type System for IFΔJ

Unambiguous and Type-Safe IFΔJ Product Lines

In order to find a criterion for unambiguity, we define the notion of consistency of a set of delta modules. A set of delta modules is *consistent* if no class added or removed in one delta module is added, removed or modified in another delta module contained in the same set, and for every class modified in more than one delta module, its direct superclass is changed at most by one delta clause and the fields and methods added, modified or removed are distinct. For a consistent set of delta modules, any order of delta module application yields the same class table since the alterations do not interfere with each other. Consistency of a set of delta modules can be inferred by only considering delta module signatures that can be obtained by a straightforward inspection of each delta module in isolation. A *delta module signature* `DMS` is the analogue of a class signature for a delta module.

A SPL is *locally unambiguous* if every set S of delta modules in the partition of $dom(\texttt{DMT})$ provided by the application partial order $\prec$ is consistent. If the SPL `L` is locally unambiguous, then it is also unambiguous. For local ambiguity, two delta modules that modify the same method cannot be placed in the same partition even if they are never applied together. However, local unambiguity can be checked by only relying on delta module signatures and the partition of $dom(\texttt{DMT})$ provided by the application partial order. This makes local ambiguity robust to change since it is preserved by changes of delta modules without changing their signatures, by refinement of the application partial order, by changes to the application conditions and by changes in the set of valid feature configurations.

A IFΔJ SPL is *type-safe* if the following conditions hold: (i) its product generation mapping is total, (ii) it is locally unambiguous, and (iii) all generated products are well-typed IFJ programs.

The constraint-based type system for IFΔJ analyzes each delta module in isolation. The results of the analysis can be combined with the product line declaration in order to check whether all the products that can be generated are well-typed.

Moreover, the type safety of a product line can be established by relying only on the the delta module signatures, the delta clause-constraints and the product line declaration without re-inspecting the delta modules and without generating the products.

Constraint-based Typing Rules for Delta Modules

The typing rules for delta modules also organize the inferred delta clause constraints in a two-level hierarchy corresponding to the structure of the delta module to support the application of a set of delta clause constraints $\mathcal{D}$ to a set of class constraints $\mathcal{C}$. The syntax of the delta clause constraints is given below

Delta-clause constraints:

`adds C` **with** $\mathcal{K}$	add the constraint "`C` **with** $\mathcal{K}$"
`removes C`	remove constraint "`C` **with** $\cdots$"
`modifies C` **with** $\mathcal{M}$	change the constraint "`C` **with** $\mathcal{K}$" into "APPLY(`modifies C` **with** $\mathcal{M}$, `C` **with** $\mathcal{K}$)"

Delta subclause-constraints:

`adds m` **with** $\mathcal{F}$	add the constraint "`m` **with** $\cdots$"
`removes m`	remove constraint "`m` **with** $\cdots$"
`replaces m` **with** $\mathcal{F}'$	change constraint "`m` **with** $\mathcal{F}$" into "`m` **with** $\mathcal{F}'$"
`wraps m` **with** $\mathcal{F}'$	change constraint "`m` **with** $\mathcal{F}$" into "`m` **with** $\mathcal{F} \cup \mathcal{F}'$"

The typing rules infer a set of *delta adds/removes/modifies-clause constraints* (one for each delta clause in the delta module). A delta adds-clause constraint consists of the keyword `adds` followed by a class

constraint. A delta removes-clause constraint is a removes-clause `removes C`. Each delta modifies-clause constraint consists of the name of the subject class `C` and of a set of *delta adds/removes/replaces/wraps-subclause constraints* that are described as follows:

- An adds-subclause constraint consists of the keyword `adds` followed by a method constraint.
- A delta removes-subclause constraint is of the form `removes m`.
- A wraps/replaces-clause constraint consists of the keyword `replaces/wraps` followed by a method constraint. Wrap-subclause constraints are inferred for modify-subclauses containing `original`, while replace-subclause constraints are inferred for modify-subclauses not containing `original`.

Thus, a set of delta clause constraints $\mathcal{D}$ can be understood as a mapping from class names to delta clause constraints, and a delta modifies-clause constraint can be understood as a mapping from method names to delta subclause constraints.

The constraint-based typing judgment for a delta module is $\vdash \texttt{delta}\ \delta\ \cdots : \mathcal{D}_\delta$, to be read as "the delta module δ has the delta clause constraints $\mathcal{D}_\delta$".

The application of a delta module signature to a class signature table, denoted by $\textsc{apply}(\texttt{DMS}, \texttt{CST})$, performs the alterations specified in `DMS` to `CST`.

The IFΔJ constraint-based type system enables checking the well-typedness of all possible products by analyzing the delta modules in isolation, generating the constraints for the products, and checking the constraints obtained for each product against the class signature table of that product.

Chapter 3

Typing Deltas, Delta Application and Delta Composition

In the previous chapter, an approach to types for delta-oriented programming (DOP) [30] is proposed, which generates a collection of constraints for a delta-oriented product line, but these can only be checked per product. That type system does not reflect the structure of the deltas as types. Moreover it is of exponential complexity.

In this chapter we describe an intuitive and modular approach to type systems for DOP based on the observation that the operations underlying DOP are similar to operations on records [27] and that these can be typed using a row polymorphic type system, as published in [22]. The type system can be used to check that applications of deltas to a core program ensure that a class or method is not added twice or removed when it is not present. The type checking approach described in Chapter 2 is complementary, providing part of the checking omitted in the presented approach. Combining the two approaches is a topic for future work. The results reported in this chapter are the result of a collaboration between BOL, KUL and CTH. BOL provided the expertise in row polymorphism, KUL provided expertise in type systems, and CTH provided expertise in delta-oriented programming.

3.1 Related Work

Thaker et al. [34] describe an informally specified approach to the safe composition of software product lines that guarantees that no reference to an undefined class, method or variable will occur in the resulting products. The approach is presented modulo variability given in the feature model and deals especially with the resulting combinatorics. The lack of a comprehensive formal model of the underlying language and type system was rectified with *Lightweight Feature Java* (LFJ) [9]. Underlying LFJ is a constraint-based type system whose constraints describe composition order, the uniqueness of fields and methods, the presence of field and methods along with their types, and feature model dependencies. The soundness of LFJ's type system was validated using theorem prover Coq.

A formal model of a feature-oriented Java-like language called *Featherweight Feature Java* (FFJ) [2] presents a similar base language also formalizing Thaker et al.'s [34] approach to safe composition, although for this system type checking occurs only on the generated product. *Colored Featherweight Java* [18], which employs a notion of coloring of code analogous to but more advanced than `#ifdef`s, lifts type checking from individual products to the level of the product line and guarantees that all generated products are type safe. More recent work [1] refines the work on FFJ, expressing the code refinements into modules rather than as low-level annotations. The resulting type system again works at the level of the product line and enjoys soundness and completeness results, namely, that a product line is well-typed if and only if all of its derived products are well-typed.

In the above mentioned work, the refinement mechanisms are monotonic, so no method/class removal or renaming is possible. Kuhlemann et al. [21] address the problem of non-monotonic refinements, though their

approach does not consider type safety. They consider the presence of desired attributes depending upon which features are selected. Checking is implemented as an encoding into propositional formulas, which are fed into a SAT solver.

Apel et al. [4] present a general, language independent, static analysis framework for reference checking—checking which dependencies are present and satisfied. This is one of the key tasks of type checking a software product line. Similar ideas are applied in a language-independent framework for ensuring the syntactic correctness of all product line variants by checking only the product line itself, again without having to generate all the variants [19]. Padmanabhan and Lutz [25] describe the DECIMAL tool, which performs a large variety of consistency checks on software product line requirements specifications, in particular, when a new feature is added to an existing system.

3.2 Results

Our approach focuses on typing deltas, their application on a program, and the composition of deltas. As deltas modify the *structure* of the program, that is, the class and interface bodies, our type system focuses on these elements, and abstracts away other typing considerations, such as dependencies between classes. In particular, our approach is not concerned with typing method bodies. This highlights the correspondence between deltas and row polymorphism.

3.2.1 Base Example

To illustrate our approach, we use a simple example written in the delta-oriented fragment of full ABS. Let's suppose that we are developing an SPL to control a set of coffee machines: the core product corresponds to the basic coffee machine model, while deltas encode possible variations on this initial model. The core product, named `C`, is presented Figure 3.1a and consists of a simple coffee machine with the capacity of making coffee or tomato juice. The core has two classes `Coffee` and `TomatoJuice` that respectively handle putting coffee (resp., tomato powder) in the machine with method `fill` and making coffee (resp., tomato juice) with method `make`.

Starting from this core product, we can develop a controller for another kind of machine that does not make tomato juice, but does offer sugar and milk and can also mix hot chocolate, with the three deltas presented Figure 3.1b. The first, `D1`, deals with the sugar and the milk. It adds one class to the product, `Settings`, with two methods giving the possibility to set the desired quantity of milk and sugar. The second, `D2`, removes class `TomatoJuice`, for coffee machines that cannot make juice. Finally, the last delta, `D3`, deals with the hot chocolate feature. This delta first adds a new class `Chocolate`, which has method `fill` for when chocolate powder is put in the machine and `make` to prepare the hot chocolate. The delta also modifies class `Settings`, adding methods that offer the possibility of either dark or white chocolate. Applying delta `D1` to the core program will result in the program shown in Figure 3.2a and applying `D1`, `D2` and `D3` in sequence will result in the code shown in Figure 3.2b.

3.2.2 Row Polymorphism and Deltas

Type systems for records [27] guarantee that record operations, such as adding and removing entries, will never fail, for example, preventing the removal of a field that does not exist. Records are typed with *row types* that store information about the structure of the records; record operations are typed with *functional types* that map a row type corresponding to the input record to an output row type. The most important feature of this type system is *row polymorphism*, which gives types of record operations in terms of both *type* and *row variables*. These work together to model the fact that an operation can be applied to records with different structure, for example, it is possible to remove the field a from any record that has this field. In addition, these variables allow the type of the input of operations to be related to their output.

The parallel between deltas and record operations is immediate. We can see class bodies as records, where each field corresponds either to a method of the class (the label being the method's name and the

```
class Coffee {
  Unit fill(Int q) { ... }
  Unit make() { ... }
}
class TomatoJuice {
  Unit fill(Int q) { ... }
  Unit make() { ... }
}
```

(a) Core Program

```
delta D1 {
  adds class Settings {
    Unit setSugar(Int q) { ... }
    Unit setMilk(Int q) { ... }
  }
}

delta D2 {
  removes class TomatoJuice;
}

delta D3 {
  adds class Chocolate {
    Unit fill(Int q) { ... }
    Unit make() { ... }
  }
  modifies class Settings {
     adds Unit setChocolateDark() { ...
     }
     adds Unit setChocolateWhite() { ...
      }
  }
}
```

(b) Deltas

Figure 3.1: A Core Program and Deltas

value its code), or to a field of the class (the label being the field's name and the value its value). Programs can also be seen as records, where each field corresponding to a class, with the label being the class name and the value being the body of the class. Based on this encoding into records, delta application can be modeled as a function manipulating a record corresponding to the core program, and delta composition is simply function composition. And thus type systems for records [27] can be used to check the application and composition of deltas.

Figure 3.3 presents the type of the core program described in the previous section. Row types consist of a finite sequence of field declaration followed by information about the remainder of the record. Field declarations are identified by a label, such as `Coffee` or `fill`, and state either that the field is present in the record, in which case the label is of the form `Pre(`τ`)`, where τ is the type of the value contained in the field, or that the field is absent from the record, in which case the label is `Abs`. The additional row information can either be `Abs`, meaning that there are no more fields in the record, or a *row variable* ρ, meaning that the rest of the record is unknown. The type in Figure 3.3 states that only the two fields `Coffee` and `TomatoJuice` are present, and both contain records corresponding to the bodies of the classes.

Figure 3.4 presents the type of delta `D1`. The type of a delta is a *polymorphic* function type that is structured in three parts: the declaration of the type variables used in the type, the row type describing the input of the delta, and the output type of the delta. The row variable describes the part of the program *not* affected by the delta. The input type of the type of `D1` states that `D1` can take as input any program that does not have class `Settings`, while the rest of it is unspecified and thus can be anything. The output type captures the modifications `D1` makes to the input program. The definition of field `Settings` has changed; it is now present, stating that class `Settings` has been added to the program, and that it contains two methods, `setSugar` and `setMilk`. As variable ρ is not changed, all other classes in the input program are not modified by `D1`.

```
class Coffee {
  Unit fill(Int q) { ... }
  Unit make() { ... }
}
class TomatoJuice {
  Unit fill(Int q) { ... }
  Unit make() { ... }
}
class Settings {
  Unit setSugar(Int q) { ...
   }
  Unit setMilk(Int q) { ...
   }
}
```

(a) After the application of D1

```
class Coffee {
  Unit fill(Int q) { ... }
  Unit make() { ... }
}
class Settings {
  Unit setSugar(Int q) { ... }
  Unit setMilk(Int q) { ... }
  Unit setChocolateDark() { ... }
  Unit setChocolateWhite() { ... }
}
class Chocolate {
  Unit fill(Int q) { ... }
  Unit make() { ... }
}
```

(b) After the application of all the deltas

Figure 3.2: Delta Applications

```
{
  Coffee : Pre( { fill : Pre; make : Pre; Abs } );
  TomatoJuice : Pre( { fill : Pre; make : Pre; Abs } );
  Abs
}
```

Figure 3.3: Core Program Type

$\forall\rho$. { Settings : Abs; ρ }
 $\rightarrow$ { Settings : Pre ({ setSugar : Pre; setMilk : Pre; Abs }); ρ }

Figure 3.4: Delta D1 Type

3.2.3 Typing

The row polymorphic type system was applied to a simple delta-oriented programming language that captures the essence of this aspect of the ABS language. The three basic delta operations modeled are: `adds` C, which adds a new class; `removes c`, which removes a class; and `modifies c` O, which modifies a class, where O is a list of operations that remove or add fields.

Typing Core Products The type rules for a core product, i.e., a set of classes, are presented Figure 3.5.

(TClass)
$$\{T_1\ \mathtt{f}_1\ def_1; \ldots; T_n\ \mathtt{f}_n\ def_n\} : \{\mathtt{f}_1 : \mathbf{Pre}; \ldots; \mathtt{f}_1 : \mathbf{Pre}; \mathbf{Abs}\}$$

Figure 3.5: Typing Core Products

Rule (TClass) is used to type the body of a class. This rule takes all the fields and methods defined in the class body, forgets about their types and definitions, and simply states in the output type that these elements are present in the class body.

Typing Deltas The typing rules for deltas are presented in Figure 3.6. Typing deltas is a little more complex than typing core products, as deltas are essentially polymorphic functions.

$$\frac{E : T}{E : \sigma(T)}\ (\textsc{TSubst}) \qquad \frac{E : T_1 \to T_2 \quad E' : T_2 \to T_3}{E; E' : T_1 \to T_3}\ (\textsc{TSeq})$$

$$\textbf{adds}\ T\ \texttt{f}\ \mathit{def} : \forall\rho.\{\texttt{f} : \textbf{Abs}; \rho\} \to \{\texttt{f} : \textbf{Pre}; \rho\}\ (\textsc{TAddField})$$

$$\textbf{removes}\ T\ \texttt{f}\ \mathit{def} : \forall\rho.\{\texttt{f} : \textbf{Pre}; \rho\} \to \{\texttt{f} : \textbf{Abs}; \rho\}\ (\textsc{TRemoveField})$$

$$\frac{C : TC}{\textbf{adds class}\ \texttt{c}\ C : \forall\rho.\{\texttt{c} : \textbf{Abs}; \rho\} \to \{\texttt{c} : \textbf{Pre}(TC); \rho\}}\ (\textsc{TAddClass})$$

$$\textbf{removes class}\ \texttt{c} : \forall\rho, \rho'.\{\texttt{c} : \textbf{Pre}(\{\rho\}); \rho'\} \to \{\texttt{c} : \textbf{Abs}; \rho'\}\ (\textsc{TRemoveClass})$$

$$\frac{O : TC_1 \to TC_2 \quad \rho \text{ fresh}}{\textbf{modifies class}\ \texttt{c}\ O : \forall\rho.\{\texttt{c} : \textbf{Pre}(TC_1); \rho\} \to \{\texttt{c} : \textbf{Pre}(TC_2); \rho\}}\ (\textsc{TModifiesClass})$$

Figure 3.6: Typing Deltas

Rule (TSubst) makes use of *substitutions* σ to replace row variables with more concrete types, thus allowing the input and output types of a function to be made more precise. Rule (TSeq) types the sequential composition of two functions E and E'. Note that this rule enforces that the output of E is a valid input of E' by requiring that the output type of E is the same as the input type of E'—typically achieved using (TSubst). The remaining type rules describe the validity of field and method addition and removal, and class addition, modification and removal. In each case, the input and output parts of the type describe which elements need to be present or absent, and polymorphism handles the remaining ingredients.

3.2.4 Examples

To illustrate the type system, we present two typing examples. First, we show how to validate the application of `D1` and `D2` (previously introduced in Figure 3.1b) to the core program (presented Figure 3.1a). Then we demonstrate that applying `D3` to the same core program would fail.

Successful Application Let's follow the typing rules to see how to validate the product line composed by the two deltas `D1` and `D2`, followed by the application `D2(D1(P))`, where `P` is the core program of Figure 3.1a. `P` is typed as presented in Figure 3.3. Denote this type by TP. Using typing rules (TClass) to type the class `Settings` and the rule (TAddClass), the delta `D1` is also typed as presented in Figure 3.4. Denote this type by TD_1. Using the same approach, we can see that the delta `D2` can be typed with TD_2 as follows:

$TD_2 = \forall\rho_t, \rho.$ `{ TomatoJuice : Pre({`ρ_t`}); `ρ` } → { TomatoJuice : Abs; `ρ` }`

Now, using the substitution σ that replaces the variable ρ with the content of the type of P, we can instantiate the type of delta `D1` to take as parameter program P, resulting in the following type:

```
{ Settings: Abs;
  Coffee: Pre({fill: Pre; make: Pre; Abs});
  TomatoJuice: Pre({fill: Pre; make: Pre; Abs}); Abs } →
{ Settings: Pre({setSugar: Pre; setMilk: Pre; Abs});
  Coffee: Pre({fill: Pre; make: Pre; Abs});
  TomatoJuice: Pre({fill: Pre; make: Pre; Abs}); Abs }
```

The type of `D1(P)` is the output of the type just presented. To type the application of `D2` to the product `D1(P)`, we can apply the same technique of instantiating TD_2 so that its input type matches the type of `D1(P)`. The output type of the result, i.e. the type of `D2(D1(P))`, is:

```
{Settings: Pre({setSugar: Pre; setMilk: Pre; Abs});
 Coffee: Pre({fill: Pre; make: Pre; Abs}); Abs }.
```

Failing Application Let's now consider what happens when applying `D3` (presented in Figure 3.1b) to program `P`. To check the application, we first need the type of `D3`. This delta consists of two operations: adding class `Chocolate` and modifying class `Settings`. Using the typing rules (TClass), (TAddClass), (TAddField), (TModifyClass) and (TSeq), `D3` can be typed with TD_3 as follows:

$TD_3 = \forall \rho_t, \rho.$
```
 { Chocolate: Abs;
   Settings: Pre({setChocolateDark: Abs; setChocolateWhite: Abs; ρt}); ρ } →
 { Chocolate: Pre({fill: Pre; make: Pre; Abs});
   Settings: Pre({setChocolateDark: Pre; setChocolateWhite: Pre; ρt}); ρ }
```

To be able to type the application of `D3` to `P`, we must instantiate TD_3 so that its input type corresponds to TP, the type of P. But this is impossible, because the class `Settings` is declared as being absent in TP, while it is required to be present in the input type of TD_3. Hence typing fails.

3.2.5 Extending to Full DOP

The presented type system is insufficient to check the validity of a full delta-oriented SPL, as it does not consider the *feature model* or *application conditions* associated with deltas [6]. These additions introduce two difficulties: ensuring that there are no *unresolved conflicts* between deltas [6] and ensuring the validity of the resulting products, whose number is in general exponential in the number of deltas. The calculus can be extended to take into account these features (Figure 3.7). Deltas are annotated with the sets of features that activate the delta. The independence of deltas is expressed using parallel composition. The term $DM \vDash$`class` $\mathtt{c}_1\ C_1\ \ldots$ `class` $\mathtt{c}_m\ C_m$, executed in the context of the feature set $f_1, \ldots, f_n$, corresponds to the computation of a product whose core is classes $\mathtt{c}_i$, under the delta model DM.

$$
\begin{array}{rcl}
PL & ::= & \texttt{delta d activated by}(f_1, \ldots, f_n)\ D\ PL \quad | \quad DM \vDash P \\
DM & ::= & 0 \quad | \quad \mathtt{d} \quad | \quad DM.DM \quad | \quad DM \parallel DM
\end{array}
\qquad \text{Delta Model}
$$

Figure 3.7: Calculus Syntax Extension

Extending the Type System

The extension type terms are of the form $DM \vDash P$, i.e., all possible delta applications resulting from the delta model DM on the core P. The extended syntax of types maps sets of features to types, describing the dependence of deltas on features and which products result from which features. For instance, a delta adding empty class `c` activated by the presence of feature f is typed: $\forall\rho.[f] : (\{\mathtt{c} : \mathbf{Abs}; \rho\} \rightarrow \{\mathtt{c} : \mathbf{Pre}(\{\mathbf{Abs}\}); \rho\}) \vee \forall\rho.[\neg f] : (\{\rho\} \rightarrow \{\rho\})$, where there is one disjunct for when f is activated and one for when it is not. This is a kind of *union type*, where each element is *guarded* by a constraint. The type syntax also captures conflicts ($\bot$). A delta type is only well formed if another delta is provided to resolve the conflict [6]. Conflicts only result in errors if they are not resolved.

Problem with the Extension

This extension is not presented in detail in this deliverable, because it fails to satisfactorily address the way features and conflicts are handled. Typing of features requires types to be duplicated. Let us consider for instance two deltas $\mathtt{d}_1$ and $\mathtt{d}_2$ that respectively add the empty class $\mathtt{c}_1$ for the feature f_1 and the empty class $\mathtt{c}_2$ for the feature f_2. To type the sequential composition of $\mathtt{d}_1$ and $\mathtt{d}_2$, four cases need to be considered (two for $\mathtt{d}_1$ times two for $\mathtt{d}_2$) corresponding to the possible activations of features f_1 and f_2. Moreover, checking

for conflicts required considering all valid orderings of the delta model and checking that they produced the same type. This is factorial, while [6] argues that conflict checking should be in $O(n^2)$.

Chapter 4

Type Systems for Abstract Failure Models

The problem of exception propagation of concurrency-related errors is a well-known issue for concurrent programming languages, UIO has done some foundational research into this problem, published in [15]. This work served as a basis for studying how to statically avoid exception-related errors in the concurrency model of ABS, research performed in collaboration between UIO and BOL.

This chapter summarizes, first, the foundational background study of type systems for concurrency-related errors, in Section 4.1. Then, in Section 4.2 a type system for Abstract Failure Models is discussed.

4.1 Foundational Background

The study is carried out for a concurrent, object-oriented calculus in the tradition of Featherweight Java. Trying to statically avoid run-time errors, the analysis is a form of type-based control-flow analysis characterizing the behavior of one single program. The problem tackled in the work is to prevent "mis-use" of locks as a basic concurrency-control mechanism, in particular in a setting allowing the use of `lock` and `unlock` statements in a non-lexical manner. These operators may lead to run-time errors and unwanted behavior; e.g., taking a lock without releasing it, which could lead to a deadlock, or trying to release a lock without owning it. This freedom of lock-usage is more general than the implicit assurance of concurrency control in ABS, where each object or more generally each concurrent object group assures mutual exclusion and the "critical section" is lexically scoped in that it cannot exceed the execution of the method body. To assure safe use of locking we present a static type and effect system which assures that, e.g., no lock is released more often than it is being held, or released by a thread which does not hold it. We call such erroneous situations *lock errors*.

To prevent lock errors, we basically keep track per thread of the number of locking and unlocking on the individual locks. There are three main challenges to this approach:

Dynamic lock creation: locks can be dynamically created as instances of a lock class.

Aliasing: As locks are accessed via references to a lock instance, they are subject to aliasing: two different variables may refer to the same lock.

Lock passing: Lock references can be passed via method calls (between objects) and via instance fields (between threads).

To capture effects related to locks, the general form of judgments for a single expression, i.e., inside one thread, is of the form

$$\sigma; \Gamma; \Delta_1 \vdash e : T :: \Delta_2 \ . \tag{4.1}$$

It is read as "given the heap σ and under the lock assumptions Δ_1 and type assumptions Γ, expression e has type T and some effect which changes Δ_1 into Δ_2". The *lock environment* keeps the assumptions

for locks, i.e. whether a lock is free (denoted by 0), or taken by some thread in which the environment needs to remember how many times it is taken, to capture re-entrance. The type and effect system is not only concerned with checking expressions, the declarations of methods are generalized, as well. To ensure compositional checking, the interface information contains information about the pre- and post-conditions concerning the lock-use, interpreted in a assume-guarantee manner. The work formalizes the (operational) *semantics*, the *type and effect system* in terms of a formal derivation system following the ideas sketched above, to avoid improper use of lock operations, and proves *soundness* by standard subject reduction.

4.2 Typing Faults in the Future

In [14] we have considered the problem of fault handling inside ABS by presenting an extension of the language in which futures are used to return fault notifications and to coordinate error recovery between the caller and callee. In fact, futures are used in ABS to return the results of asynchronous method calls, and they uniquely identify those method calls. Thus, they provide a natural means to distribute fault notifications and kill requests. More precisely, the callee of a method can return a fault notification to the caller to signal callee-side failures, while the caller can ask to kill/compensate a previous call. These features support distributed error recovery policies programming.

In the Deliverable D2.1 [11] dedicated to "A configurable deployment architecture" we discuss the primitives proposed in [14] to deal with fault handling in ABS. Here we briefly recall such primitives and we discuss the extension of the ABS type system needed to ensure that all the faults that may be raised by a method invocation are managed by the caller.

The primitives for fault handling are as follows. The caller can ask for termination/compensation of a previous call by performing operation $x := f.\texttt{kill}$ (reminiscent of the `cancel` method of Java futures) on the future f identifying the call, while the callee can signal a failure by executing the `abort` n command (n describes the kind of failure). If the callee aborts, then it will definitely terminate its activities. On the contrary, if the caller performs $x := f.\texttt{kill}$, it expects the callee to react by executing some compensating activity (in contrast to Java, where the call is just interrupted). The compensating code s is attached to the `return` statement, that we replace with the new command `return` e `on compensate` s. This is the main novelty of our proposal: when a callee successfully terminates, it has not definitely completed its activity, as it will possibly have to perform its compensation activity in case of failure of the caller. This mechanism is inspired by the compensation mechanisms adopted in service orchestration languages like WS-BPEL [24] or Jolie [12]. A compensation can return a result to the caller: to this aim we use a new future which is freshly created and assigned to x by $x := f.\texttt{kill}$.

It is worth noting that a callee can choose which fault name to return in its corresponding future, depending on the particular cause of failure. In order to allow the caller to properly react to each of these notifications, we had to slightly modify the $x := f.\texttt{get}$ primitive used in ABS to allow the caller to receive the return value. The new construct is $\texttt{on}\ x := f.\texttt{get}\ \texttt{do}\ s\ \overline{\texttt{on fail}\ n_i\ s_i}$, which executes $x := f.\texttt{get}$ as before, but then it executes the statement s if the future f contains a value v, or the statement s_i if f contains a fault name n_i. In the first case the value v is assigned to variable x, otherwise x is unchanged.

Besides presenting the formal definition of the syntax and operational semantics of the new primitives, in [14] we have also discussed how to extend the type system of ABS (some sample rules of it are in Fig. 4.1) to ensure that all the faults that may be raised by a method invocation are managed by the caller.

We use typing contexts which are mappings from names (of variables, interfaces and classes) to types. The reserved name *return* is bound to the return type of the current method. Relation $\preceq$ is the subtyping relation.

To check the correctness of error management, one has essentially to tag a method with the list of failures it can raise. Also, one has to specify the behavior of the compensation, including (recursively) its ability to throw faults. According to this idea, the signature Sg of a method m becomes:

$$\begin{array}{rcl} Sg & ::= & T\ m\ (\overline{T\ x})\ ED \\ ED & ::= & \texttt{throws}\ \overline{n}\ [\texttt{on comp}\ T\ ED] \end{array}$$

$$\frac{\Gamma \vdash x : \texttt{fut}\langle T\rangle}{\Gamma \vdash x.\texttt{get} : T}\ (\textsc{Get}) \qquad \frac{\Gamma \vdash e : \Gamma(return)}{\Gamma \vdash \texttt{return}\ e}\ (\textsc{Return}) \qquad \frac{\Gamma \vdash e : T' \qquad T' \preceq \Gamma(v)}{\Gamma \vdash v := e}\ (\textsc{Assign})$$

Figure 4.1: Sample typing rules for ABS

$$\frac{n \in \Gamma(faults)}{\Gamma \vdash \texttt{abort}\ n}\ (\text{T-}\textsc{Abort}) \qquad \frac{\Gamma \vdash x : T \quad \Gamma \vdash f : \texttt{fut}\langle T'\rangle\ \texttt{throws}\ \overline{n_i}\ CM \quad T' \preceq T \quad \Gamma \vdash s \quad \Gamma \vdash s_i}{\Gamma \vdash \texttt{on}\ x := f.\texttt{get}\ \texttt{do}\ s\ \overline{\texttt{on fail}\ n_i\ s_i}}\ (\text{T-}\textsc{Get})$$

$$\frac{\Gamma \vdash e : \Gamma(return) \qquad \Gamma(comp) = T\ \texttt{throws}\ \overline{n}\ CM \qquad \Gamma[return \mapsto T, faults \mapsto \overline{n}, comp \mapsto CM] \vdash s}{\Gamma \vdash \texttt{return}\ e\ \texttt{on compensate}\ s}\ (\text{T-}\textsc{Return})$$

$$\frac{\Gamma \vdash x : T' \qquad \texttt{fut}\langle T\rangle\ \texttt{throws}\ \overline{m_i}, \texttt{Ann}\ CM \preceq T' \qquad \Gamma \vdash f : \texttt{fut}\langle T''\rangle\ \texttt{throws}\ \overline{n_i}\ \texttt{on comp}\ T\ \texttt{throws}\ \overline{m_i}\ CM}{\Gamma \vdash x := f.\texttt{kill}}\ (\text{T-}\textsc{Kill})$$

Figure 4.2: Sample typing rules for error management in ABS

where T is the syntactic category for types.

Here $\overline{n}$ is the list of names of faults method m may throw. The optional clause `on comp` T ED specifies the typing of the compensation. It is omitted if the compensation is not present. In this case it stands for the (infinite) unfolding of the type `on comp null throws NoC rec` X.`on comp null throws` ε X where `null` is a subtype of any data type, ε the empty list and `NoC` the special fault thrown by a method that defines no compensation when it is required to compensate.

We show in Fig. 4.2 the main typing rules for error recovery. We need two reserved names: $faults$, bound to the list of faults that the current method can throw, and *comp*, bound to the typing of the current compensation. Rule *T-Abort* simply checks that the thrown fault is allowed. Rule *T-Get* verifies that the returned value has the correct type, and that all the faults that may be raised by the callee are managed. Rule *T-Return* checks the type of the returned value, and ensures that the compensation has the expected behavior. Finally rule *T-Kill* controls that variable x can store the result of the `kill`, including the possibility for it to be `Ann`, a special fault returned when the method invocation is annulled before its execution started. In this case the method is not executed.

The subtyping relation $\preceq$ extends the corresponding ABS relation to deal also with the new types for futures. It can be defined by:

$$\frac{T \preceq T' \quad \overline{n} \supseteq \overline{n'} \quad T_1 \preceq T_1' \quad ED \preceq ED'}{\texttt{fut}\langle T\rangle\ \texttt{throws}\ \overline{n}\ [\texttt{on comp}\ T_1\ ED] \preceq \texttt{fut}\langle T'\rangle\ \texttt{throws}\ \overline{n'}\ [\texttt{on comp}\ T_1'\ ED']}\ (\textsc{Fut-Sub})$$

Chapter 5

Conclusions

In this deliverable we reported on type systems for software variability, namely types for Delta-Oriented Programming, and for deployment variability, namely types for Abstract Failure Models.

For future work we plan to study the integration of the two type systems for Delta types described in Chapters 2 and 3 with the full ABS language. The first step towards an integrated framework for Deltas will be to improve the type system presented in Chapter 3 to deal with conflicts in an effective way, to add dependencies to the framework in order to type check the products resulting from delta application, and thereby extend the calculus toward a full SPL while keeping polynomial complexity. This work will be done as part of Task **T1.5** *Integrated Tool Platform*, and it will be described in Deliverable **D1.5**.

Behavioral type systems for ABS have been studied and will be described in HATS Deliverable **D4.3** *Correctness*.

Type systems for HATS components are left as future work, to be included in Task **T3.3** *Hybrid Analysis for Evolvability* and described in Deliverable **D3.3**.

Bibliography

[1] Sven Apel, Christian Kästner, Armin Grösslinger, and Christian Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3):251–300, 2010.

[2] Sven Apel, Christian Kästner, and Christian Lengauer. Feature Featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In Yannis Smaragdakis and Jeremy G. Siek, editors, *GPCE*, pages 101–112. ACM, 2008.

[3] Sven Apel and Christian Lengauer. Superimposition: A language-independent approach to software composition. In *Software Composition*, volume 4954 of *Lecture Notes in Computer Science*, pages 20–35. Springer-Verlag, 2008.

[4] Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. Language-independent reference checking in software product lines. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, FOSD '10, pages 65–71, New York, NY, USA, 2010. ACM.

[5] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.*, 30(6), 2004.

[6] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract Delta Modeling. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 13–22, New York, NY, USA, October 2010. ACM.

[7] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Longman, 2001.

[8] Benjamin Delaware, William Cook, and Don Batory. A Machine-Checked Model of Safe Composition. In *FOAL*, pages 31–35. ACM, 2009.

[9] Benjamin Delaware, William R. Cook, and Don S. Batory. Fitting the pieces together: a machine-checked model of safe composition. In Hans van Vliet and Valérie Issarny, editors, *ESEC/SIGSOFT FSE*, pages 243–252. ACM, 2009.

[10] Final Report on Feature Selection and Integration, March 2011. Deliverable 2.2b of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[11] A configurable deployment architecture, February 2012. Deliverable 2.1 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

[12] Claudio Guidi, Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. Dynamic error handling in service oriented applications. *Fundam. Inform.*, 95(1):73–102, 2009.

[13] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[14] Einar Broch Johnsen, Ivan Lanese, and Gianluigi Zavattaro. Fault in the future. In *Proc. of COORDINATION'11*, volume 6721 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2011.

[15] Einar Broch Johnsen, Thi Mai Thuong Tran, Olaf Owe, and Martin Steffen. Safe locking for multi-threaded Java with exceptions. *Journal of Logic and Algebraic Programming*, (0):–, 2011. To appear.

[16] K. Kang, J. Lee, and P. Donohoe. Feature-Oriented Project Line Engineering. *IEEE Software*, 19(4), 2002.

[17] C. Kästner, S. Apel, S.S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the impact of the optional feature problem: Analysis and case studies. In *Proc. Int'l Software Product Line Conference (SPLC). SEI*, 2009.

[18] Christian Kästner and Sven Apel. Type-checking software product lines - a formal approach. In *ASE*, pages 258–267. IEEE, 2008.

[19] Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don S. Batory. Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In Manuel Oriol and Bertrand Meyer, editors, *TOOLS (47)*, volume 33 of *Lecture Notes in Business Information Processing*, pages 175–194. Springer, 2009.

[20] Charles Krueger. Eliminating the Adoption Barrier. *IEEE Software*, 19(4):29–31, 2002.

[21] Martin Kuhlemann, Don S. Batory, and Christian Kästner. Safe composition of non-monotonic features. In Jeremy G. Siek and Bernd Fischer, editors, *GPCE*, pages 177–186. ACM, 2009.

[22] Michaël Lienhardt and Dave Clarke. Row types for delta-oriented programming. In *6th international workshop of Variability Modeling of Software-intensive Systems (VaMoS 2012)*. ACM, 2012.

[23] Roberto E. Lopez-Herrejon, Don S. Batory, and William R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *European Conference on Object-Oriented Programming (ECOOP'05)*, volume 3586 of *Lecture Notes in Computer Science*, pages 169–194. Springer-Verlag, 2005.

[24] Oasis. *Web Services Business Process Execution Language Version 2.0.* `http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html`.

[25] Prasanna Padmanabhan and Robyn R. Lutz. Tool-supported verification of product line requirements. *Autom. Softw. Eng.*, 12(4):447–465, 2005.

[26] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer-Verlag, 1997.

[27] Didier Rémy. *Type inference for records in natural extension of ML*, pages 67–95. MIT Press, Cambridge, MA, USA, 1994.

[28] Ina Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Proc. of 4th Intl. Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, 2010.

[29] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proc. of 14th Software Product Line Conference (SPLC 2010)*, September 2010.

[30] Ina Schaefer, Lorenzo Bettini, and Ferruccio Damiani. Compositional type-checking for delta-oriented programming. In *10th International Conference on Aspect-Oriented Software Development, AOSD 2011*, pages 43–56. ACM, 2011.

[31] Ina Schaefer and Ferruccio Damiani. Pure Delta-oriented Programming. In Sven Apel, Don Batory, Krzysztof Czarnecki, Florian Heidenreich, Christian Kästner, and Oscar Nierstrasz, editors, *Proc. 2nd International Workshop on Feature-Oriented Software Development (FOSD'10) Eindhoven, The Netherlands*, pages 49–56. ACM Press, 2010.

[32] Ina Schaefer, Alexander Worret, and Arnd Poetzsch-Heffter. A Model-Based Framework for Automated Product Derivation. In *Proc. of Workshop in Model-based Approaches for Product Line Engineering (MAPLE 2009)*, 2009.

[33] Rok Strniša, Peter Sewell, and Matthew Parkinson. The Java module system: core design and semantic definition. In *OOPSLA*, pages 499–514. ACM, 2007.

[34] Sahil Thaker, Don S. Batory, David Kitchin, and William R. Cook. Safe composition of product lines. In Charles Consel and Julia L. Lawall, editors, *GPCE*, pages 95–104. ACM, 2007.

Glossary

Terms and Abbreviations

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

Abstract Failure A user-introduced failure without a specific interpretation in the model.

Application Condition A condition in propositional logic (alternately represented as a set of sets of features) indicating to which feature configurations a delta is applicable.

Application Function A function mapping deltas to their application condition.

Class Refinement Like class inheritance, except that the new 'subclass' completely supplants the original, being instantiated everywhere the original was instantiated before.

Compatibility of Feature Fodels Condition indicating if two features, products, or product lines can be reconciled.

Conflict The condition between two incompatible, non-ordered deltas.

Conflict Resolving Delta The delta that resolves a given conflict between two other deltas. It has to be greater in the partial order than the conflicting deltas and equalize the two possible orderings between them.

Consistent Conflict Resolution A condition of a deltoid which simplifies the conflict resolver property (and thus the unambiguity property).

Core In delta modeling, the core is the basic product that may be modified by product-deltas. The core may also be seen as a delta modifying the empty product.

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

Delta A unit of functionality and conflict resolution in delta modeling, able to modify a product using invasive composition of code or other content.

Delta Model Generally, a means for expressing the semantics of features within product lines. In the formalism, a delta model is defined more specifically as $(D, \prec)$, a partially ordered set of deltas.

Deployment Component A modeling abstraction for deployment choices, restricting the execution capacity of different parts of ABS models.

Deployment Architecture The architectural description of a model; i.e., how COGs are allocated to deployment components.

Deployment Scenario A deployment architecture with given amounts of resources for the different deployment components.

Deployment Variability Changes in the deployment scenarios, either in the deployment architecture or in the resources allocated to different deployment components.

DOP Delta-Oriented Programming.

FDeltaJ Featherweight Delta Java, a programming language with deltas based on Featherweight Java.

Featherweight Java A reduced version of Java in which almost all features are dropped to obtain a small calculus for rigorous proofs of language properties. It has often been extended to explore new language constructs such as context oriented programming, traits and deltas.

Feature Generally, an increment in software functionality. On the level of feature models it is merely a label with no inherent semantic meaning.

Feature Configuration A subset of available features. It identifies a single product in a product line if it is valid in the feature model.

Feature Interaction The phenomenon in which two features interact on a semantic level. The term is used for both intentional interaction of features and for unintentional and unwanted interaction.

Feature Model An expression of the variability within product lines. Abstractly it may be seen as a system of constraints on the set of possible feature configurations.

Feature Module Comparable to a delta, a feature module may contain code modifications to be applied to a software product in order to fully implement a single feature.

FOP Feature Oriented Programming.

Product One member of a product line, with well-defined commonalities and variabilities to other products.

Product Line Generally, a family of products with well-defined commonalities and variabilities. In the delta modeling formalism, a product line is more specifically defined as $(\Phi, c, D, \prec, \gamma)$, a feature model, a core product, a delta model and an application function.

Software Family See Software Product Line.

Software Product Line A family of software systems with well-defined commonalities and variabilities. See also Product Line.

SPL Software Product Line

Unambiguous Delta Model The property in a delta model that every conflict is properly resolved, such that a unique implementation is guaranteed.