

# Evaluation of Delta Modelling in the ABS Language

Wouter Seyen

Thesis voorgedragen tot het behalen  
van de graad van Master of Science  
in de ingenieurswetenschappen:  
computerwetenschappen

**Promotor:**

Prof. dr. D. Clarke

**Assessoren:**

Prof. dr. M. Denecker

Dr. E. Truyen

**Begeleiders:**

R. Muschevici

Dr. J. Proença

© Copyright KU Leuven

Without written permission of the thesis supervisor and the author it is forbidden to reproduce or adapt in any form or by any means any part of this publication. Requests for obtaining the right to reproduce or utilize parts of this publication should be addressed to the Departement Computerwetenschappen, Celestijnenlaan 200A bus 2402, B-3001 Heverlee, +32-16-327700 or by email [info@cs.kuleuven.be](mailto:info@cs.kuleuven.be).

A written permission of the thesis supervisor is also required to use the methods, products, schematics and programs described in this work for industrial or commercial use, and for submitting this publication in scientific contests.

Zonder voorafgaande schriftelijke toestemming van zowel de promotor als de auteur is overnemen, kopiëren, gebruiken of realiseren van deze uitgave of gedeelten ervan verboden. Voor aanvragen tot of informatie i.v.m. het overnemen en/of gebruik en/of realisatie van gedeelten uit deze publicatie, wend u tot het Departement Computerwetenschappen, Celestijnenlaan 200A bus 2402, B-3001 Heverlee, +32-16-327700 of via e-mail [info@cs.kuleuven.be](mailto:info@cs.kuleuven.be).

Voorafgaande schriftelijke toestemming van de promotor is eveneens vereist voor het aanwenden van de in deze masterproef beschreven (originele) methoden, producten, schakelingen en programma's voor industrieel of commercieel nut en voor de inzending van deze publicatie ter deelname aan wetenschappelijke prijzen of wedstrijden.

# Preface

I would like to thank my mentors Radu Muschevici and José Proença for their insight and support throughout the year, as well as my promoter Dave Clarke for giving me the opportunity to write this thesis. I also wish to thank all other people who read parts of my thesis and gave helpful comments and insights by discussing the subject with me.

Last but not least, I would like to thank my parents for giving me the opportunity to study all these years at the K.U.Leuven.

*Wouter Seyen*



# Contents

<b>Preface</b>	<b>i</b>
<b>Abstract</b>	<b>v</b>
<b>List of Figures and Tables</b>	<b>vii</b>
<b>List of Listings</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem Statement . . . . .	1
1.2 Context . . . . .	2
1.3 Outline of the Thesis . . . . .	2
<b>2 State of the Art</b>	<b>3</b>
2.1 Delta-Oriented Programming . . . . .	3
2.2 The Abstract Behavioural Specification Language . . . . .	4
<b>3 Development of a Smart Home Product Line in ABS</b>	<b>11</b>
3.1 Feature Diagram . . . . .	11
3.2 Feature Model in $\mu$ TVL . . . . .	13
<b>4 Code Reuse in the ABS Language</b>	<b>17</b>
4.1 Delegation Approach . . . . .	17
4.2 Single Type Approach . . . . .	24
4.3 Multiple Delta Approach . . . . .	28
4.4 Conclusion . . . . .	33
<b>5 Evaluation of the ABS Delta-Oriented Programming Methodology</b>	<b>35</b>
5.1 General Experience with ABS . . . . .	35
5.2 Developing in ABS . . . . .	36
5.3 Performance of ABS Execution . . . . .	42
5.4 Unit Tests . . . . .	45
5.5 Conclusion . . . . .	49
<b>6 Conclusion</b>	<b>51</b>
<b>Bibliography</b>	<b>53</b>
<b>Appendices</b>	<b>1</b>
<b>A Source Code Excerpts</b>	<b>3</b>
A.1 Test of Execution Time of Delegation vs. Inheritance . . . . .	3

## CONTENTS

---

A.2 Test of ABS Execution Time . . . . .	4
A.3 Test of Java Execution Time . . . . .	5
<b>B Poster</b>	<b>7</b>
<b>C Article</b>	<b>11</b>

# Abstract

This thesis evaluates the ABS Modelling Framework which supports delta-modelling. A short introduction to the ABS language is given and the smart home product line on which a part of the evaluation is based is discussed. This thesis presents three approaches to achieve code reuse: the delegation approach, the single type approach and the multiple delta approach. The first one is applicable to most object-oriented programming languages as it does not rely on deltas, the second one relies on deltas but is only applicable when one type of a type hierarchy is needed in a product, the last one also relies on deltas and is the most usable approach to code reuse of the three. ABS also shows to be fairly intuitive to use. It still has to mature a bit, as some annoyances in the language show. These can be improved in the future though. Also, according to a comparative performance test, ABS is about three orders of magnitude slower than Java, which is due to performance being a low priority requirement when ABS was developed.





# List of Figures and Tables

## List of Figures

2.1	Example of a feature diagram . . . . .	7
3.1	Feature diagram of the smart home . . . . .	12
3.2	Full class diagram of the Smart Home . . . . .	15
4.1	Sensor type hierarchy . . . . .	18
4.2	Sensor class diagram for delegation approach . . . . .	19
4.3	Type hierarchy of the fire sensors . . . . .	24
4.4	Type hierarchy of Sensor, TemperatureSensor and FireSensor . . . . .	28
5.1	Average of ten executions of the ABS performance test . . . . .	43
5.2	Average of ten executions of the Java performance test . . . . .	43

## List of Tables

4.1	Execution times of inheritance vs. delegation (in $\mu s$ ) . . . . .	24
-----	---	----



# List of Listings

2.1	Typing in ABS . . . . .	5
2.2	Example of concurrency in ABS . . . . .	6
2.3	Example of $\mu$ TVL . . . . .	7
2.4	Example of the Delta Modelling Language . . . . .	8
2.5	Example of the Product Line Configuration Language . . . . .	8
2.6	Example of the Product Selection Language . . . . .	8
3.1	Feature model of the smart home in $\mu$ TVL . . . . .	13
3.2	PSL code fore a smart home with electric heating . . . . .	14
3.3	Deltas linked to features in the smart home implementation . . . . .	16
4.1	Interfaces of the sensors . . . . .	19
4.2	Implementation of the FireAndMovementSensor class . . . . .	19
4.3	Build a list of different sensors . . . . .	20
4.4	Test of [ <i>Near</i> ] and [ <i>Far</i> ] behaviour of delegation . . . . .	21
4.5	Output of the sensor test-program . . . . .	21
4.6	Problem in the delegation approach . . . . .	23
4.7	Implementation of the fire sensor hierarchy . . . . .	25
4.8	Test program of the fire sensor hierarchy . . . . .	26
4.9	Sensor type hierarchy . . . . .	29
4.10	Reassigning objects to variables of different types . . . . .	32
4.11	if-statement for Sensor . . . . .	33
4.12	Checking type using a function . . . . .	33
4.13	if-statement for Sensor using a function . . . . .	33
5.1	Run() method of the Sensor class . . . . .	37
5.2	Asynchronous call - await - get . . . . .	37
5.3	Problem with await and suspend . . . . .	38
5.4	No intermediate results . . . . .	39
5.5	With intermediate results . . . . .	39
5.6	Conditional statement with method call . . . . .	40
5.7	Conditional statement without method call . . . . .	40
5.8	Example of how a static method declaration could look like . . . . .	41
5.9	ABS speed test . . . . .	42
5.10	ABS code transformed in Java . . . . .	44

5.11	ABSTest example . . . . .	45
5.12	Generated test runner code . . . . .	46
5.13	Testing a private method . . . . .	47
5.14	ABSTest error message . . . . .	47
5.15	Improved error message . . . . .	48
5.16	Output from test runner generator . . . . .	48
A.1	Test program of the different sensor types . . . . .	3
A.2	Test of ABS execution time . . . . .	4
A.3	Test of Java execution time . . . . .	5

# Chapter 1

## Introduction

The ABS Modelling Framework is being developed in the context of the HATS (Highly Adaptable and Trustworthy Software using Formal Models) European project [5, p. 2], and supports developing software product line (SPL) systems following established software product line engineering (SPLE) practices, e.g. feature-oriented development. It allows the precise modelling and analysis of component-based distributed concurrent systems, focusing on their functionality while not taking into account concerns such as concrete resources, deployment scenarios and scheduling policies [20].

### 1.1 Problem Statement

Delta Modelling [15] is a programming approach to developing software product lines in which modifications to a program are encapsulated using *deltas*. Deltas can be seen as patches to a core program. Upon compilation deltas are incrementally applied to the core program, guided by a selection of features of a desired product, thereby adding, removing, or modifying functionality. Delta modelling is supported in the Abstract Behavioural Specification (ABS) language, a modern object-oriented programming (OOP) language currently under development.

Since development of the ABS language is still in progress, not much experience with the ABS language and its approach to delta modelling is available.

The goals of this master's thesis are to evaluate ABS w.r.t. its practical usefulness in developing SPL systems, to identify design patterns and best practices related to delta-oriented programming, to uncover code smells and anti-patterns, to expose weaknesses and limitations, and to propose improvements to the ABS language. The focus will be on researching patterns for code reuse in the ABS language which needs to be accomplished using deltas. A smart home product line will be implemented using the delta modelling constructs in the ABS language. This implementation will serve as the basis for evaluating ABS.

### 1.2 Context

This master's thesis is executed in the context of Software Product Line Engineering (SPLE). This methodology has been proven to allow the development of a diversity of software products and software-intensive systems at lower costs, in shorter time, and with higher quality w.r.t. other approaches [13, p. V].

Most systems of a certain complexity make use of software because of the benefits it offers regarding flexibility and functionality. It is much easier to change or add something in software than it is in hardware, so the amount of embedded systems is steadily growing. The amount of variability in these systems is growing even faster. This is where SPLE comes into play. SPLE allows designers to control this ever increasing variability in a structured way.

*Software product line engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customisation. ([13, p. 14])*

This definition of SPLE covers both the development of pure software products and of software for embedded systems as the principles of product line engineering are the same for both types of software. The definition mentions the use of platforms. These should be build carefully by considering re-usability in advance: the platform should contain the reusable parts of the product line. The platform can then be customised to generate different end products. This entails using the concept of managed variability, which means that commonalities and differences in the applications should be well documented and modelled in a common way. This is consistent with the definition of a software product line given by Schaefer et al. [17, 16]: ‘A software product line (SPL) is a set of software systems with well-defined commonalities and variabilities’. Managed variability also impacts the way software is developed, extended and maintained. Instead of just changing software to suit the new needs, adapting the software should be done only in those places where it makes sense to do so. [13, p. 14-15]

### 1.3 Outline of the Thesis

Chapter 2 discusses the current state of the art by giving descriptions of delta-oriented programming and the ABS language. Chapter 3 presents the smart home product line. Chapter 4 covers three approaches to code reuse in ABS with their advantages and disadvantages. In Chapter 5 other methodologies for delta-oriented programming are given, covering discovered patterns and methodologies as well as experiences with the ABS language. Chapter 6 summarises the thesis and gives a general conclusion.

## Chapter 2

# State of the Art

This chapter elaborates on the state of the art regarding the thesis. In Section 2.1 delta-oriented programming and its relation to other programming paradigms is discussed, Section 2.2 discusses the ABS language.

### 2.1 Delta-Oriented Programming

In an object-oriented programming paradigm, two kinds of approaches to implementing SPLs exist: annotative approaches, in which source code is removed if it does not correspond to a feature in the selected product configuration, and compositional approaches, where code fragments corresponding to different features are assembled to form the required end product [17]. Delta-Oriented Programming (DOP) belongs to the group of compositional approaches. DOP was introduced as a programming language approach for more flexible implementation of SPLs [16]. In DOP, feature modules are generalised to delta modules that allow the adding, refining, and removing of fields, methods, and classes. To create a SPL using DOP, first a feature model should be build. A valid product is then selected from the feature model (i.e. at least the mandatory features and a minimal set of required alternative features) for implementation. Selecting a valid product allows using known software engineering techniques to ensure validity and quality. Delta modules can then be created and applied to this core to add or remove features by adding, changing and removing code. Schaefer and Damiani [17] differentiate between two kinds of DOP: Core DOP and Pure DOP. Core DOP refers to traditional DOP in which a single valid core product must be selected, as a baseline is needed. Pure DOP is introduced as a term referring to DOP where this requirement is dropped: products are assembled solely from delta modules. The requirement is dropped in order for Pure DOP to be a true generalisation of feature-oriented programming. The ABS language on which this master's thesis focuses, uses the Core DOP approach and will be discussed in Section 2.2.

### 2.1.1 Feature-Oriented Programming

DOP was introduced to provide more flexibility and expressiveness than feature-oriented programming (FOP) [14, 3], which can also be used to implement SPLs. FOP focuses on large-scale compositional programming and feature modularity [1].

The main difference between DOP and FOP is that in FOP the finest level of granularity for variability is a feature module, while in DOP this is a delta module. In FOP, feature modules are applied incrementally and can only introduce new classes or refine existing ones by adding fields and methods or by overriding existing methods. In contrast to delta modules, feature modules cannot remove code. Delta modules are also not restricted to one specific feature. This opens the possibility to provide deltas which contain code that is necessary for the interaction between two features. In FOP such interaction code cannot be handled directly, since feature modules are intended to represent exactly one product feature [16].

### 2.1.2 Aspect-Oriented Programming

In contrast to FOP, Aspect-Oriented Programming (AOP) focuses on cross-cut modularity [1]. It is a technique that improves the separation of concerns in software [11]. A frequently used example is logging. Logging occurs in many places in the code base of a program. Therefore it requires much code duplication since the logging code has to be implemented in all the places where logging is required. AOP allows for the separate implementation of such cross-cutting concerns, called aspects. These aspects are then applied at specified joinpoints. This technique effectively decreases code duplications since the code only needs to be defined once and is then applied at the specified locations.

Deltas are similar to aspects in that they also define orthogonal code modifications. However, they differ in several ways. Deltas are more like features in FOP, where features are typically used in an incremental fashion, refining the code and other features. Deltas are guided by features but are more fine grained. They are combined to implement a feature. On the other hand, aspects address crosscutting concerns by taking over control at specified joinpoints, and executing their code. Another difference is that deltas have the possibility to introduce new, independent classes. This cannot be done using aspects. They can only introduce new members to classes, and new superclasses and interfaces. This is because aspects have no architectural model [1].

## 2.2 The Abstract Behavioural Specification Language

The abstract behavioural specification language, or ABS language or just ABS as it will be called throughout the rest of the text, is a language which has a hybrid functional and object-oriented core with built-in concurrency constructs. It comes with extensions that support the development of systems that are adaptable to diversified requirements as well as to future changes, yet capable to maintain a high level of trustworthiness [20]. The full ABS language actually consists of 5 different



languages: Core ABS, the Micro Textual Variability Language ( $\mu$ TVL), the Delta Modelling Language (DML), the Product Line Configuration Language (CL), and the Product Selection Language (PSL), which will be discussed in the following sections.

### 2.2.1 Core ABS

Core ABS is a subset of the full ABS language and does not in itself address SPL, but forms a basis for extensions which will capture SPL artifacts such as features and feature integration [8]. It can be used as a regular OOP language and is used as the language for specifying the core behavioural modules.

DOP was one of the extensions added on top of Core ABS in order to support SPLE. DOP was chosen as a research project for ABS over inheritance because it was fairly new and the developers of ABS wanted to see how well it fared in practice. This is why ABS does not support inheritance, but code reuse should be achieved using deltas. Chapter 3 discusses the feasibility of this.

In ABS, interfaces define types and the methods available for that type. Classes can implement these interfaces and their methods but are not types themselves. Other ways to create types in ABS are to use algebraic data types by using the **data** keyword, or to define type synonyms which are semantically equivalent to their synonym by using the **type** keyword. Listing 2.1 gives an overview of the possibilities for typing in ABS.

```
1  module Example;
2  data Car =
3      BMW |
4      Mercedes |
5      Ferrari
6      ;
7  interface Foo {
8      Int getInt();
9      Car getCar();
10 }
11 type Bar = Foo;
12 class F(Car car) implements Foo {
13     Int getInt() {
14         return 5;
15     }
16     Bar getBar() {
17         Foo f = new F(BMW);
18         return f;
19     }
20     Car getCar() {
21         return this.car;
22     }
23 }
```

LISTING 2.1: Typing in ABS

ABS is designed to model distributed systems and applies a concurrency model using concurrent object groups (COG) [18] for this. Each COG has its own heap of objects and communicates with other COGs through asynchronous method calls. Calls inside one COG are regular, local, synchronous method calls. Synchronous method calls are made using the normal “.” and can only be performed on *[Near]* references. These are references to objects which belong to the same COG. *[Near]* is an annotation which can be added to variables to indicate that they refer to an object residing in the same COG. Asynchronous method calls are made using a “!” and can only be performed on *[Far]* references. These are references to objects which belong to a different COG. *[Far]* is an annotation similar to *[Near]*, only *[Far]* indicates that the variable refers to an object residing in a different COG. The result of an asynchronous call is called a future. The execution of a method can be suspended by performing an **await** on such a future. When execution resumes, a **get** can be performed on the future and the actual result is returned. Listing 2.2 gives a short example of the use of ABS’ concurrency model.

```

1  [Far]Foo foo = new cog Foo();
2  [Near]Foo foo2 = new Foo();
3  Fut<Int> f = foo!getInt();
4  Int i = foo2.getInt();
5  await f?;
6  i = f.get;
```

LISTING 2.2: Example of concurrency in ABS

On line 1 a new object of class `Foo` is created inside a new COG. On line 2 a new object of class `Foo` is created as well, but this time it is created inside the current COG. On line 3 an asynchronous call is made to the `getInt()` method of the *[Far]* `Foo`, the result of this call is a future parametrised with type `Int`. The future contains information about the completion of the asynchronous call and the result of the `getInt()` method. On line 4 a call to the `getInt()` method of the *[Near]* `Foo` is made. This is a regular, synchronous call since it resides in the same method and the result is immediately returned. On line 5 an **await** is performed on the future created on line 3 the execution will be suspended until the asynchronous call returns. This also gives the opportunity for other tasks inside the current COG to be executed. When the task is resumed, the **get** on line 6 is executed and the result assigned to `i`.

### 2.2.2 Micro Textual Variability Language

$\mu$ TVL is the language used to describe feature models [2, 10] using a textual representation. It allows for describing feature models as a forest of nested features with possibly multiple roots (for orthogonal variability) with the possibility to add boolean or integer attributes to each feature. Additional constraints can be put on the presence of features and on the possible values of attributes.

Listing 2.3 gives a short example of  $\mu$ TVL. This is the textual representation of the feature diagram in Figure 2.1. It defines a feature model with three features, one of them having three subfeatures of which at least two and at most three need to be selected (the large, second constraint in the feature diagram), and one being optional and requiring another feature to be present (the first constraint in the feature diagram). A larger example will be explained in Chapter 3.

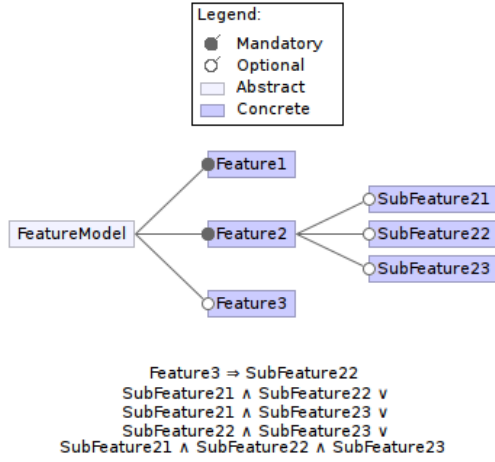


FIGURE 2.1: Example of a feature diagram

```

1  root FeatureModel {
2    group allof {
3      Feature1,
4      Feature2 {
5        group [2..3] {
6          SubFeature21,
7          SubFeature22,
8          SubFeature23
9        }
10     },
11     opt Feature3 { require
12       : SubFeature22 ;
13     }
14   }
15 }

```

LISTING 2.3: Example of  $\mu$ TVL

### 2.2.3 Delta Modelling Language

Delta modules (or deltas in short), which are used to achieve variability in a SPL, are defined using DML. They contain program modifications, adding, removing, and refining classes, methods, and fields. The deltas are applied to the core module, and specified using Core ABS.

When defining a delta, first an identifier and possibly some attributes are given. The use of attributes is not exploited in this thesis. Secondly, the added, modified, or removed class, method, or field is specified. When modifying or adding, the new implementation is given next. Modifications may sometimes want to call the original implementation in order to extend its behaviour. This can be done by calling the special method `original()`.

Listing 2.4 gives an example of the use of DML. It adds an interface `FooBar` with two methods and adds this interface to the `F` class. It also removes the `Foo` interface and `getBar()` method from the `F` class, the `getString()` method is added and the `getInt()` method modified.

```
1  delta D1;
2  uses Example;
3  adds interface FooBar {
4      String getString();
5      Int getInt();
6  }
7  modifies class F adds FooBar removes Foo {
8      adds String getString() {
9          return "String";
10     }
11     removes Bar getBar();
12     modifies Int getInt() {
13         Int i = original();
14         return i + 1;
15     }
16 }
```

LISTING 2.4: Example of the Delta Modelling Language

### 2.2.4 Product Line Configuration Language

To link feature models created using  $\mu$ TVL to delta modules created using DML, the product line configuration language is used. A product line configuration starts with the name of the SPL followed by the applicable features. Next the deltas to be applied are specified together with an optional **after** clause, specifying a partial order on the application of deltas, and a **when** clause, specifying for which features the delta should be applied. Listing 2.5 gives an example.

```
1  productline Examp;
2  features Feature1, Feature2, SubFeature21, SubFeature22,
   SubFeature23, Feature3;
3  delta D1 when Feature1;
4  delta D2 after D1 when SubFeature12;
```

LISTING 2.5: Example of the Product Line Configuration Language

### 2.2.5 Product Selection Language

The different end products are specified using the product selection language. A product description consists of a product name, followed by the features the product has and, if necessary, filled in parameters. Listing 2.6 gives an example.

```
1  product Example1 (Feature1, Feature2, SubFeature21,
   SubFeature23);
```

LISTING 2.6: Example of the Product Selection Language

### 2.2.6 Full Specification of ABS

The previous sections briefly discussed the different language components of ABS. Many more concepts exist in these languages, but the interested reader is referred to [20], [6] and [5] for a full discussion of these languages. It needs to be noted that these texts use another syntax of ABS than is used in this master's thesis. The syntax used here is newer and solves some of the problems the old syntax had.



## Chapter 3

# Development of a Smart Home Product Line in ABS

In this chapter, the model of a smart home product line is explained in more detail. The possible features in this product line and how they relate to each other are discussed here.

### 3.1 Feature Diagram

The SPL implemented for this thesis is a product line for a smart home and is based on the home automation model described in the book Software Product Line Engineering [13]. Figure 3.1 shows a graphical representation of the feature model of the smart home.

- Every product (valid configuration of features) needs to have an interface and optionally a remote interface. Interface in this context means a way of interaction with the system, not an interface of a programming language. A remote interface is then a way of controlling the smart home through a remote device.
- A smart home can optionally have electronic doors which can be sliding or swinging doors. When a smart home has an electronic door, it also needs to have door sensors.
- A smart home can optionally have electronic blinds, which require window sensors to be present as well.
- A smart home can have two types of alarms: a fire alarm and a burglar alarm. When a fire alarm is present, a fire sensor needs to be present as well. When a burglar alarm is present, a movement sensor is mandatory.
- Heating can optionally be controlled by the smart home. Heating is either gas, electric, or oil fuel based. When heating is controlled by the smart home, a temperature sensor needs to be present.

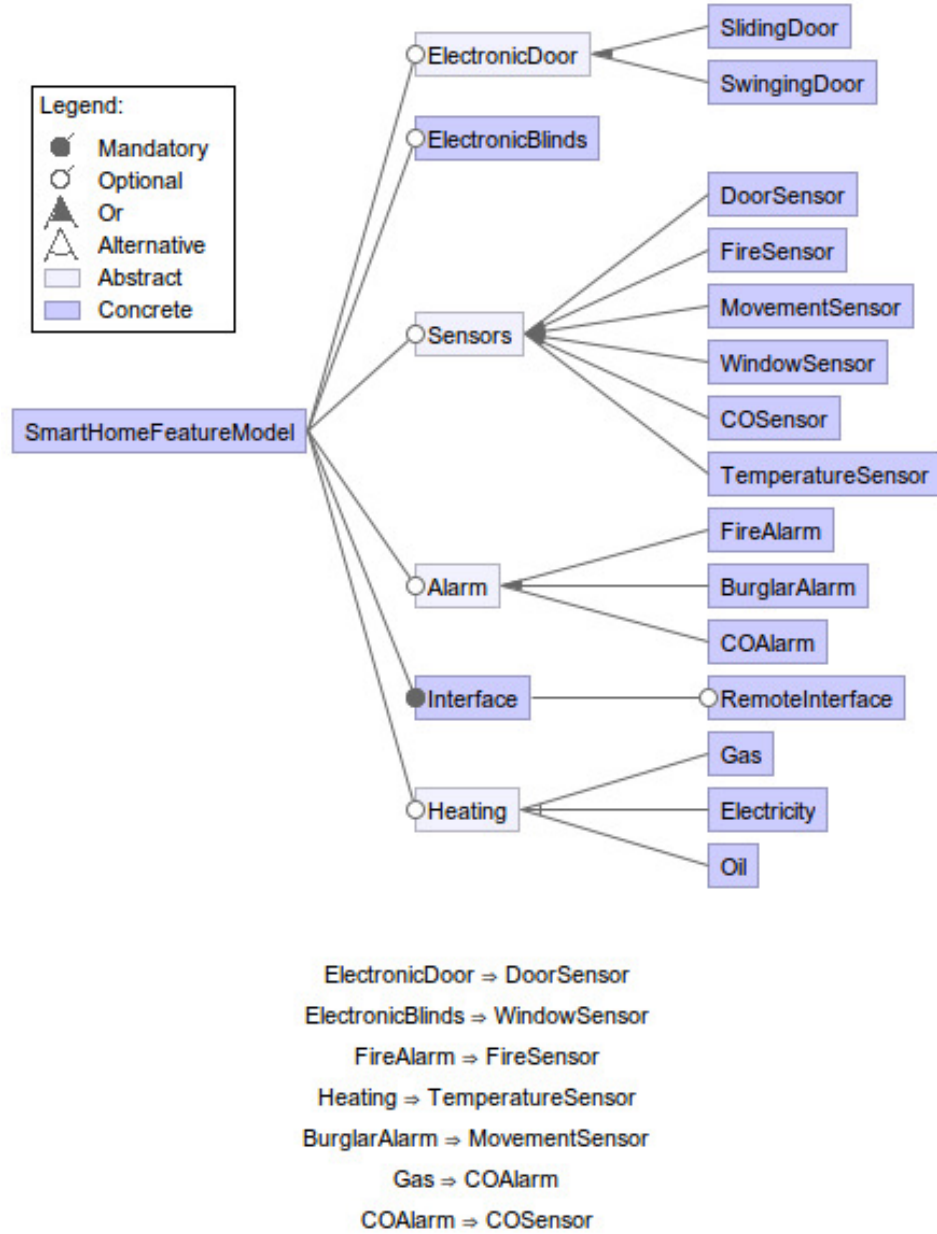


FIGURE 3.1: Feature diagram of the smart home

- Sensors can also be optionally available to provide information on the state of the smart home regardless of whether there are other features depending on these sensors.

This feature model is sufficiently large to provide enough challenges to gain experience with the use of the ABS language, while at the same time, it is sufficiently small to still be manageable for a one person research project. The feature model



also shows the need for SPLs because even though it only has 21 features which are constrained to some degree, it still has 3.510 possible configurations.





### 3.2 Feature Model in $\mu$ TVL

```

1  root SmartHome {
2    group allof {
3      opt ElectronicDoor {
4        group [1..*] {
5          SlidingDoor,
6          SwingingDoor
7        }
8        require: DoorSensor ;
9      },
10   opt ElectronicBlinds { require: WindowSensor ; },
11   opt Sensors {
12     group [1..*] {
13       DoorSensor,
14       FireSensor,
15       MovementSensor,
16       WindowSensor,
17       COSensor,
18       TemperatureSensor
19     }
20   },
21   opt Alarm {
22     group [1..*] {
23       FireAlarm { require: FireSensor ; },
24       BurglarAlarm { require: MovementSensor ; },
25       COAlarm { require: COSensor ; }
26     }
27   },
28   Interface {
29     group allof {
30       opt RemoteInterface
31     }
32   },
33   opt Heating {
34     group oneof {
35       Gas { require: COAlarm ; },
36       Electric,
37       Oil
38     }
39     require: TemperatureSensor ;
40   }
41 }
42 }
```

LISTING 3.1: Feature model of the smart home in  $\mu$ TVL

The definition of the feature model in  $\mu$ TVL is given in Listing 3.1. The **group**

**allof** keywords, e.g. on line 2, mean that all its grouped features without the **opt** keyword need to be selected. Using the **opt** keyword here is equivalent to using  in the feature diagram, not using **opt** here is equivalent to . The **group oneof** keyword on line 34 means that exactly one of the grouped features needs to be selected. This is equivalent to  in the feature diagram. When using **group** in combination with a cardinality, e.g. on line 4, the cardinality determines how many of the grouped features need to be selected. In this feature model, only cardinalities of  $[1..*]$  are used, but they can take any number (e.g.  $[2..4]$ ). The cardinality  $[1..*]$  is equivalent to  in the feature diagram, other cardinalities need to be represented using constraints (like in Figure 2.1). The **require**, e.g. on line 8 is used to state that when that feature is selected, the features after **require** also need to be selected. So in this case, if the feature `ElectronicDoor` is selected, the feature `DoorSensor` also needs to be selected.

## Class Diagram

Figure 3.2 shows the class diagram of the program that would be implemented in regular OOP should every feature in the feature model (in Figure 3.1) be selected. If only a subset of the possible features is selected, say a smart home with electric heating, then only the classes and interfaces in light green - `SmartHomeInterface` (with a subset of it's methods), `Heating`, `GasBurner`, `Observer`, `Observable`, `Sensor` and `TemperatureSensor` - would need to be implemented. To retain the overview, not all methods are filled in in this class diagram.

Defining the product of the smart home with electric heating is done using PSL. Listing 3.2 shows the code for this. The `Interface` feature is selected because it is mandatory, `Heating` and `Electric` are selected because these are the desired features. `Sensors` and `TemperatureSensor` are selected because the `Heating` feature requires this the `TemperatureSensor` to be present.

```
1 product ElectricHeating (Interface, Heating, Electric, Sensors,
    TemperatureSensor);
```

LISTING 3.2: PSL code fore a smart home with electric heating

The selected features are linked to deltas using CL. Listing 3.3 shows an excerpt of the smart home implementation doing exactly this. In this code, the delta `DHeating` is applied when the `Heating` feature is selected, but only after the `DTemperatureSensor` delta is applied (line 20). The `DTemperatureSensor` delta is applied when the `TemperatureSensor` feature is selected (line 17), but since `Heating` requires `TemperatureSensor`, this dependency can never lead to problems.

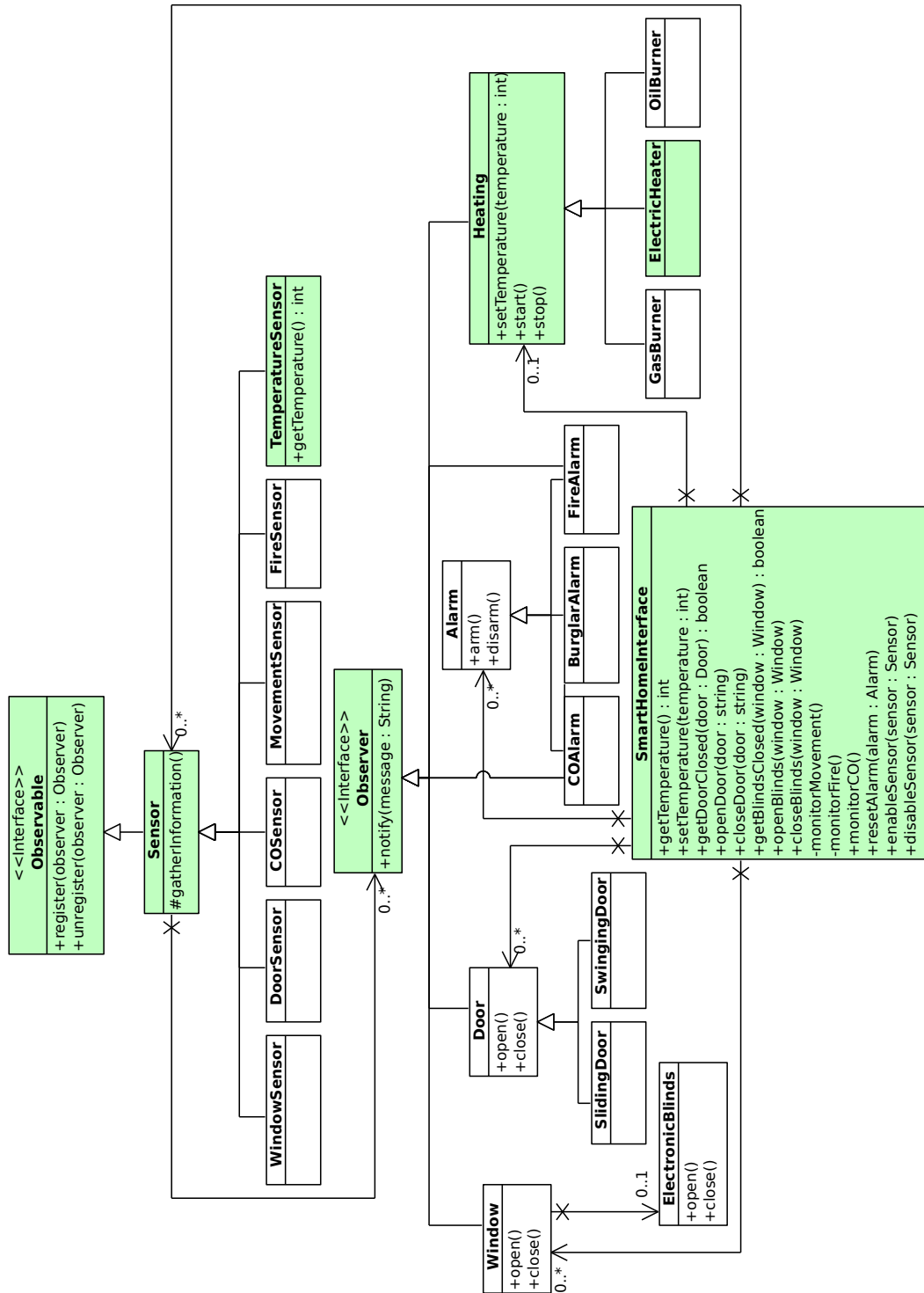


FIGURE 3.2: Full class diagram of the Smart Home

### 3. DEVELOPMENT OF A SMART HOME PRODUCT LINE IN ABS

---

```
1 productline House;
2   features ElectronicDoor, SlidingDoor, SwingingDoor,
   ElectronicBlinds,
3   Sensors, DoorSensor, FireSensor, MovementSensor,
   WindowSensor, COSensor,
4   TemperatureSensor, Alarm, FireAlarm, BurglarAlarm, COAlarm,
   Interface,
5   RemoteInterface, Heating, Gas, Electric, Oil;
6
7   // Deltas concerning House
8   delta DTemperature when TemperatureSensor;
9   ...
10  // Deltas concerning SmartHomeInterface
11  delta DTemperatureMonitor after DTemperatureSensor when
   TemperatureSensor;
12  delta DHeatingInterface after DTemperatureMonitor when
   Heating;
13  ...
14  // Deltas concerning Sensor
15  delta DObserver when Sensors;
16  delta DSensor after DObserver when Sensors;
17  delta DTemperatureSensor after DSensor when
   TemperatureSensor;
18  ...
19  // Deltas concerning Heating
20  delta DHeating after DTemperatureSensor when Heating;
21  delta DElectricHeater after DHeating when Electric;
22  ...
```

LISTING 3.3: Deltas linked to features in the smart home implementation

## Chapter 4

# Code Reuse in the ABS Language

Most object-oriented programming languages (e.g. Java) use inheritance as a mechanism to avoid code duplication. As explained in Section 2.2, there is no support for inheritance in ABS. ABS provides deltas which can be used to achieve code reuse. In the next sections three approaches, the last two of which use deltas, to achieve code reuse in ABS are discussed. The first one is called the *delegation approach* (Section 4.1) and is applicable to other OOP languages as well. The second approach makes use of deltas and can only be used when just one 'subclass' of a class needs to be introduced; this will be called the *single type approach* (Section 4.2). The last, and most difficult one is the *multiple delta approach* (Section 4.3) which allows using more than one type of a type hierarchy in the system. The first two approaches use small implementations of a sensor type hierarchy as working examples. For the third approach, the implementation of the smart home is used.

### 4.1 Delegation Approach

Delegation is a known pattern in software engineering and can be defined as:

*An implementation mechanism in which an object forwards or delegates a request to another object. The delegate carries out the request on behalf of the original object. ([7, p. 360])*

This means that instead of handling a method call itself, an object performs a method call to another object to which it has a reference and which provides an implementation for that method. So it receives a method call and forwards it to another class. Delegation can be used as an alternative to inheritance since the delegation of a method to another class can be viewed as a subclass delegating method calls to its superclass. The difference is that in delegation, the subclass has the superclass as a variable instead of it being an instance of the superclass. To avoid confusion, a superclass in the delegation approach will from now on be called a

*superobject*, as an instantiation of the supertype's class is used as object to delegate its method calls to.

#### 4.1.1 Sensor Implementation

In light of the smart home model, a test using the delegation approach was done by implementing a type hierarchy for sensors. The type hierarchy is shown in Figure 4.1. The `FireAndMovementSensor` is not part of the smart home, it is merely used here to show some properties of the delegation approach.

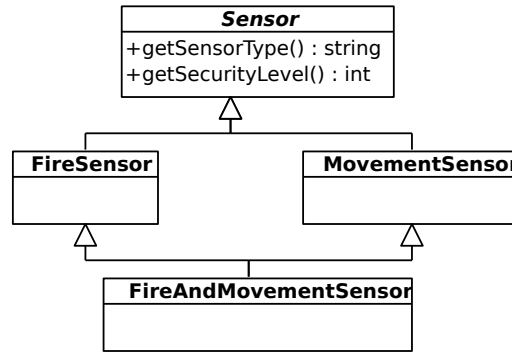


FIGURE 4.1: Sensor type hierarchy

The `Sensor` class is made abstract since no direct instantiations of a sensor should be made. `FireSensor` and `MovementSensor` inherit from `Sensor`, and `FireAndMovementSensor` from both `FireSensor` and `MovementSensor`. If `FireSensor` and `MovementSensor` have different implementations of some methods, this creates a Nixon Diamond [9].

Figure 4.2 shows how the class diagram looks like when the type hierarchy of the sensors is mapped to the delegation approach. Every type is now represented as an interface with an implementing class. These classes have a reference to their superobject(s) instead of inheriting from it/them. This means that for all the methods the superobjects have, a method needs to be created which can either call the superobject's method, or give a new implementation (overriding). When implementing this in ABS, the interfaces of the different sensor types are defined first (Listing 4.1).

Secondly, the classes are implemented. Listing 4.2 shows the implementation of `FireAndMovementSensor`, as this is the most interesting case. The `FireAndMovementSensor` class implements the methods required by `Sensor`: `getSensorType()` and `getSecurityLevel()` which are both overridden. For `getSecurityLevel()` the results of both superobjects' methods are combined to get a new result. This shows that the delegation approach can also mimic multiple inheritance in a straightforward way.

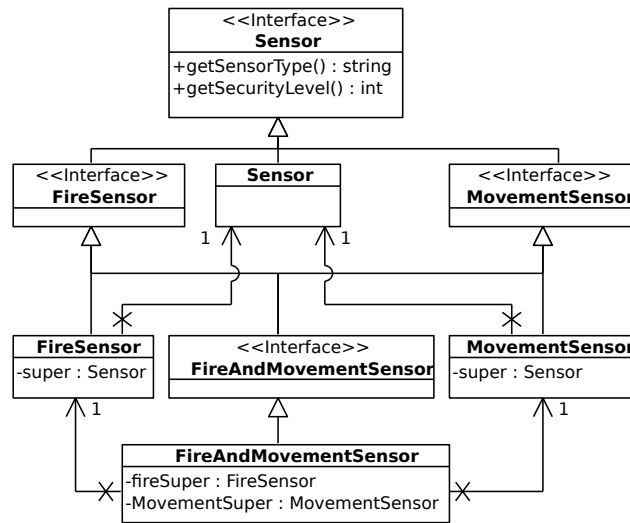


FIGURE 4.2: Sensor class diagram for delegation approach

```

1  interface Sensor {
2      String getSensorType();
3      Int getSecurityLevel();
4  }
5  interface FireSensor extends Sensor {}
6  interface MovementSensor extends FireSensor {}
7  interface FireAndMovementSensor extends FireSensor,
    MovementSensor {}

```

LISTING 4.1: Interfaces of the sensors

```

1  class FireAndMovementSensor implements FireAndMovementSensor {
2      FireSensor fireSuper;
3      MovementSensor movementSuper;
4      {
5          this.fireSuper = new FireSensor();
6          this.movementSuper = new MovementSensor();
7      }
8      String getSensorType() {
9          return "Fire and Movement Sensor";
10     }
11     Int getSecurityLevel() {
12         Int fire = fireSuper.getSecurityLevel();
13         Int movement = movementSuper.getSecurityLevel();
14         return fire + movement;
15     }
16 }

```

LISTING 4.2: Implementation of the FireAndMovementSensor class

### 4.1.2 Testing the Sensor Behaviour

A small program was written to test the behaviour of the different sensor classes. First a list is built containing all the different types of sensors (Listing 4.3), once with the list as a *[Near]* reference and once with the list as a *[Far]* reference. Next, the `getSensorType()` method is called on all the sensors in the list and the result printed to the console. The code for the test is shown in Listing 4.4. The resulting output is shown in Listing 4.5.

```
1  class ListBuilder implements ListBuilder {
2      List<Sensor> buildSensorList() {
3          Sensor s = new Sensor();
4          FireSensor fs = new FireSensor();
5          MovementSensor ms = new MovementSensor();
6          FireAndMovementSensor fms = new FireAndMovementSensor();
7          List<Sensor> sensors = Cons(s, Cons(fs, Cons(ms, Cons(fms,
8              Nil))));
9          return sensors;
10 }
```

LISTING 4.3: Build a list of different sensors

The list containing the different sensors is parametrised so that it contains objects of type `Sensor`. However, when the type of the `Sensor` is asked using the `getSensorType()` method, the type of the originally created object is returned. This is exactly how dynamic binding in inheritance would respond: method calls are bound based on the actual type of the object and not on the declared type. It is easy to see why this happens in ABS. Every kind of sensor implements the interface `Sensor`, so the list can hold it. However the different classes implementing the different interfaces have no direct knowledge of each other (only through delegation), so they can only respond to the method with their own implementation. A discussion of static behaviour will be given in Section 5.2.5.

### 4.1.3 Pattern

The example discussed in Section 4.1.1 can be generalised into a pattern:

1. Define the type hierarchy.
2. For every type in the hierarchy: define an interface which extends its direct supertype's interface.
3. For every interface, define a class which implements that interface.
  - a) Define a field *F* of the supertype's type.
  - b) In the initialisation block, create an object *O* of the supertype's class and assign it to *F*.



```

1  class Test {
2      Unit run() {
3          // Test behaviour of Near sensors
4          Printer printer = new Printer();
5          ListBuilder nearBuilder = new ListBuilder();
6          List<Sensor> sensors = nearBuilder.buildSensorList();
7          Bool continue = ~isEmpty(sensors);
8          printer.print("Near list of Sensors");
9          while (continue) {
10             Sensor s = head(sensors);
11             sensors = tail(sensors);
12             String toPrint = s.getSensorType();
13             printer.print(toPrint);
14             continue = ~isEmpty(sensors);
15         }
16         printer.print("~~~~~");
17         // Test behaviour of Far sensors
18         ListBuilder builder = new cog ListBuilder();
19         Fut<List<Sensor>> f = builder!buildSensorList();
20         await f?;
21         List<Sensor> farSensors = f.get;
22         continue = ~isEmpty(farSensors);
23         printer.print("Far list of Sensors");
24         while (continue) {
25             Sensor s = head(farSensors);
26             farSensors = tail(farSensors);
27             Fut<String> fs = s!getSensorType();
28             await fs?;
29             String toPrint = fs.get;
30             printer.print(toPrint);
31             continue = ~isEmpty(farSensors);
32         }
33     }
34 }

```

LISTING 4.4: Test of [*Near*] and [*Far*] behaviour of delegation

```

1  "Near list of Sensors"
2  "Generic Sensor"
3  "Fire Sensor"
4  "Movement Sensor"
5  "Fire and Movement Sensor"
6  "~~~~~"
7  "Far list of Sensors"
8  "Generic Sensor"
9  "Fire Sensor"
10 "Movement Sensor"
11 "Fire and Movement Sensor"

```

LISTING 4.5: Output of the sensor test-program

- c) Delegate all method calls that don't need overriding to `O`.
- d) Define getters and setters for all variables that need to be available in lower types' implementations (and add these getters and setters to the type's interface).

#### 4.1.4 Discussion

The delegation approach provides an alternative to inheritance in an easy to understand manner. The programmer gains a lot of control over the code using this approach. For instance, multiple inheritance can be simulated using this approach without much difficulty, and problems like the Nixon Diamond do not even exist, since the programmer has to specifically declare which superobject's method, combination of superobjects' methods, or own implementation he wishes to use.

Four main problems exist with the delegation approach: *(a)* using an overridden method, *(b)* coding overhead, *(c)* memory overhead, and *(d)* execution overhead. These are discussed in detail below.

**Overridden Method** The first problem with the delegation approach is that when a method call is performed on a subtype, but this method is implemented in a higher type's implementation and this method calls another method which is overridden in the subtype, the subtype's method implementation will not be called. Instead, the implementation of the higher type will be used (or one of its supertypes if it does not provide a new implementation). Take for instance Listing 4.6. When using regular inheritance, the programmer expects the variable `i` on line 29 to contain 10 after the call to `foo()`. However, because the class `Super`, to which `foo()` was delegated has no knowledge of any overriding of its `bar()` method, `i` will contain 5.

This problem can easily be overcome by having the `foo()` method in `Sub` call `bar()` directly, resulting in code duplication. But this is what this approach tried to avoid in the first place. The delegation approach therefore is far from optimal.

**Coding Overhead** The delegation approach also suffers from a coding overhead. Since methods are not inherited from a supertype, every method of every supertype all the way to the top of the hierarchy needs to be defined and implemented for every type in the type hierarchy. This can cause a large overhead if the top levels of the hierarchy contain many methods which do not need overriding in the lower levels. Also, getters and setters need to be defined for all variables which need to be available lower in the hierarchy since these lower types have no other way of reading or writing to this variables. This seriously deteriorates any encapsulation present since the variables are now publicly available for reading and writing. For instance in Java, this encapsulation problem is handled by making variables `protected`, which means they are only visible for the class and its subclasses.

**Memory Overhead** A third concern is memory. When a hierarchy consists of many levels and many objects of the lowest level need to be created, many more

```
1  interface Super {
2      Int foo();
3      Int bar();
4  }
5  interface Sub extends Super {
6  }
7  class Super implements Super {
8      Int foo() {
9          return bar();
10     }
11     Int bar() {
12         return 5;
13     }
14 }
15 class Sub implements Sub {
16     Super super;
17     Unit run() {
18         this.super = new Super();
19     }
20     Int foo() {
21         return super.foo();
22     }
23     Int bar() {
24         return 10;
25     }
26 }
27 {
28     Sub sub = new Sub();
29     Int i = sub.foo();
30 }
```

LISTING 4.6: Problem in the delegation approach

other objects need to be created as well. Take for instance a type hierarchy of ten levels. If an object of the lowest level is created, an object of the level above that is created as well, and for the level above that, and so on, creating ten different objects in total to only represent one object.

**Execution Overhead** A last possible concern is execution overhead. Non-overriding methods call the method of their superobject, which may possibly call the method in their superobject, and so on. This results in an overhead on the stack as well as in execution time because of the method call overhead. To check whether this overhead in execution time is substantial, a test was performed comparing inheritance and the delegation approach in Java. A method was called 9,223,372,036 times<sup>1</sup> for three levels of inheritance and delegation and for ten levels of inheritance and delegation. The results were averaged over ten runs. These averages are shown in Table 4.1 with execution times expressed in microseconds. The code for this test is available in Appendix A.1.

---

<sup>1</sup>This is `Long.MAX_VALUE / 1000000000` in Java

	Inheritance	Delegation	Difference
3 levels	4,746,942	4,751,934	+0.105 % (4,992)
10 levels	4,749,890	4,882,556	+2.793 % (132,666)
Difference	+0.062 % (2,948)	+2.749 % (130,622)	

TABLE 4.1: Execution times of inheritance vs. delegation (in  $\mu s$ )

The execution times for three levels and ten levels of inheritance are nearly the same with ten levels being 2,948  $\mu s$  slower than three levels, this is a difference of only 0.062 % and is probably due to execution fluctuations on the processor. For three levels of inheritance and delegation, delegation is 4,992  $\mu s$  slower than inheritance, or 0.105 % which is still very small. When looking at ten levels, the difference is already much larger with delegation being 132,666  $\mu s$  slower than inheritance, or 2.793 % which is comparable to ten levels of delegation being 130,622  $\mu s$ , or 2.749 % slower than three levels of delegation. Looking at these results, it can be concluded that delegation is in fact slower than inheritance. However, the difference in execution time is so small that for most applications this is of no concern.

## 4.2 Single Type Approach

When only one type of the type hierarchy may be used in each product, the single type approach can be used. In this approach the top-level type is modelled as a regular class and the subtypes are defined as deltas, changing the original class's implementation to suit their needs by changing implemented methods and adding new methods and fields.

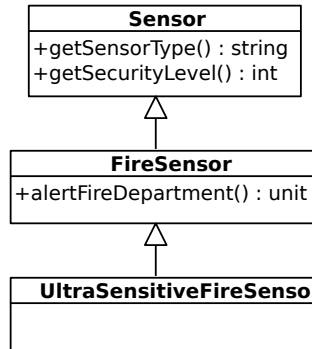


FIGURE 4.3: Type hierarchy of the fire sensors

Figure 4.3 shows the class hierarchy of three types of sensors. The `Sensor` class has two methods which will be implemented in a regular class in ABS. The `FireSensor` class has an extra method, this method will be added to the `Sensor` class in a delta module. It will also be added to the `Sensor` interface so that the new method will be visible outside the class. `UltraSensitiveFireSensor`

has no new methods, but changes the result of a call to `getSensorType()` and `getSecurityLevel()`, also by using a delta module. Since only one type in the hierarchy is available in the entire program, the original interface can be reused and no new interfaces need to be defined.

```

1  module Sensor;
2  export *;
3  import * from Printer;
4  interface Sensor {
5      String getSensorType();
6      Int getSecurityLevel();
7  }
8  class Sensor implements Sensor {
9      String getSensorType() {
10         return "Generic Sensor";
11     }
12     Int getSecurityLevel() {
13         return 5;
14     }
15 }
16 delta DFireSensor;
17 uses Sensor;
18 modifies interface Sensor {
19     adds Unit alertFireDepartment();
20 }
21 modifies class Sensor {
22     modifies String getSensorType() {
23         return "Fire Sensor";
24     }
25     modifies Int getSecurityLevel() {
26         return 7;
27     }
28     adds Unit alertFireDepartment() {
29         Printer printer = new Printer();
30         printer.printS("Fire!");
31     }
32 }
33 delta DUltraSensitiveFireSensor;
34 uses Sensor;
35 modifies class Sensor {
36     modifies String getSensorType() {
37         return "Ultra-Sensitive Fire Sensor";
38     }
39     modifies Int getSecurityLevel() {
40         return 10;
41     }
42 }

```

LISTING 4.7: Implementation of the fire sensor hierarchy

The code for this example is given in Listing 4.7. First the interface for `Sensor` is defined and implemented in the `Sensor` class (lines 4 to 15). Next, the `DFire-`

Sensor delta is defined which adds `alertFireDepartment()` to the Sensor interface and class and modifies the method implementations of Sensor (lines 16 to 32). Finally, the `DUltraSensitiveFireSensor` delta also modifies the original methods of the Sensor class (lines 33 to 42).

```
1  module Test;
2  import * from Sensor;
3  import * from Printer;
4  class Test() {
5    Sensor sensor;
6    Unit run() {
7      sensor = new Sensor();
8      Printer printer = new Printer();
9      String toPrint = sensor.getSensorType();
10     printer.printS(toPrint);
11     Int level = sensor.getSecurityLevel();
12     toPrint = intToString(level);
13     printer.printS(toPrint);
14   }
15 }
16 {
17   new cog Test();
18 }
19 delta DUseFireSensors;
20 uses Test;
21 modifies class Test {
22   modifies Unit run() {
23     original();
24     sensor.alertFireDepartment();
25   }
26 }
27 productline Sensors;
28 features Fire, UltraSensitive;
29 delta DFireSensor when Fire;
30 delta DUseFireSensors when Fire;
31 delta DUltraSensitiveFireSensor after DFireSensor when
    UltraSensitive;
32
33 product FireSensor (Fire);
34 product UltraSensitiveFireSensor (Fire, UltraSensitive);
35
36 root Sensors {
37   group [0..2] {
38     Fire,
39     UltraSensitive { require: Fire; }
40   }
41 }
```

LISTING 4.8: Test program of the fire sensor hierarchy

Listing 4.8 shows a test program for the fire sensor hierarchy. It defines two products, one with a fire sensor and one with an ultra-sensitive fire sensor (lines 33

and 34). For the first one, two deltas are applied, the `DFireSensor` delta (Listing 4.7) and the `DUseFireSensors` delta (lines 19 to 26). The first changes the implementation of the `Sensor` class, the second one changes the implementation of the `run()` method of the `Test` class, so that it makes use of the newly available method `alertFireDepartment()`. When the `UltraSensitiveFireSensor` product is chosen, the `DUltraSensitiveFireSensor` delta (Listing 4.7) is applied as well, after the `DFireSensor` delta. The **after** statement is important, as they both change the methods `getSensorType()` and `getSecurityLevel()`.

This example shows that when new methods are added to a class, existing methods wanting to use the added methods have to be completely rewritten if it is necessary to call the new methods in between the existing code, or if it has to be called on an object created inside the existing method. E.g. printing the result of `getSensorType()`, then calling `alertFireDepartment()`, and then printing the result of `getSecurityLevel()` is impossible without copying the original code and inserting the new method call in between.

#### 4.2.1 Pattern

The example discussed above can be generalised into a pattern:

1. Define the type hierarchy.
2. Define an interface `I` for the top-level type.
3. Define a class implementing `I`.
4. For each subtype `S`, define a delta which modifies the original class to suit the needs of `S`.
5. Apply the delta's in top-down order until the delta of the needed type is reached.

#### 4.2.2 Discussion

The single type approach is easy to use when exactly one type of a type-hierarchy is used throughout the entire product. It makes use of deltas, modifying the classes directly, which alleviates the problem the delegation approach has with methods calling other methods without doing anything besides that. Because all modifications are made to the original class, the resulting compiled code will be much smaller than when the delegation approach would be used.

The single type approach is very easy to use, but having only one type of a type hierarchy available throughout an entire program is very problematic as this is a case that will only occur very sporadically. The single type approach also becomes much harder to use when the types lower in the hierarchy have methods the types higher up do not have. It is possible that code has to be rewritten in this case, creating a coding overhead. However, in regular OOP with inheritance, when a method wants to start using a type lower in the hierarchy, it has to be rewritten as well, so the single type approach is not worse than when inheritance is available in this aspect.

### 4.3 Multiple Delta Approach

The single type approach is only useful when the entire product needs exactly one specific class of a type hierarchy. In this case, the existing class is altered so that it provides the functionality that is necessary in that particular product. When two or more types in a type hierarchy are needed, the single type approach becomes useless. The multiple delta approach provides an answer to this problem.

The first intuition into tackling the problem was to provide a delta for each different type, changing the class to what is needed for that specific type and using that delta in different places in the code. This is not possible however, since deltas are applied to the entire code, not just to specific locations. Hence, a kind of local single type approach is not possible.

A solution can be found in the ABS typing system. The perceived type of an object is dependent on the variable that contains the object. So if for instance a class implements interfaces `Observable` and `Alarm`, an instantiation of this class can be assigned to a variable of the type `Observable` as well as a variable of the type `Alarm`. However, the variable of type `Observable` can only access the methods defined in the `Observable` interface and the variable of type `Alarm` can only access the methods defined in the `Alarm` interface. This provides a large part of the solution, since variables can be declared to be of a specific type and only the methods belonging to the corresponding interface can be accessed.

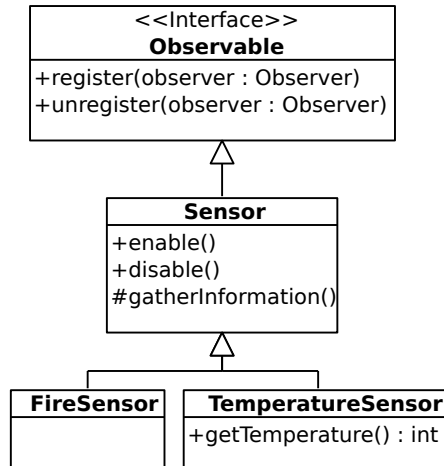


FIGURE 4.4: Type hierarchy of `Sensor`, `TemperatureSensor` and `FireSensor`

Using this observation, the code in Listing 4.9 was written. This is an excerpt from the smart home implementation of the sensor type hierarchy shown in Figure 4.4. In the first delta (`DSensor`, lines 1 to 24), the interface and its implementing class `Sensor` are introduced. This is done in a delta as the `Sensor` feature is not mandatory. Next, in the delta `DTemperatureSensor` (lines 25 to 49), the `TemperatureSensor` interface is defined, extending the `Sensor` interface. This new interface is added to the original `Sensor` class and the new



`getTemperature()` method is added to the `Sensor` class and implemented. Similarly, in the delta `DFireSensor` (lines 50 to 67) the interface `FireSensor` is defined, extending the `Sensor` interface, and added to the `Sensor` class. If all three deltas would be applied, there would be a `Sensor` class implementing the `Sensor`, `TemperatureSensor` and `FireSensor` interfaces. It is then possible to have variables of the three different types, with only variables of the type `TemperatureSensor` able to receive calls to the `getTemperature()` method.

```
1  delta DSensor;
2  uses House;
3  adds data SensorType =
4    Sensor |
5    TemperatureSensor |
6    FireSensor;
7  adds interface Sensor extends Observable {
8    Unit enable();
9    Unit disable();
10 }
11 adds class Sensor(House house, SensorType typE) implements
    Sensor {
12   Bool enabled = True;
13   ...
14   Unit enable() {
15     this.enabled = True;
16   }
17
18   Unit disable() {
19     this.enabled = False;
20   }
21
22   Unit gatherInformation() {
23   }
24 }
25 delta DTemperatureSensor;
26 uses House;
27 adds interface TemperatureSensor extends Sensor {
28   Int getTemperature();
29 }
30 modifies class Sensor adds TemperatureSensor {
31   adds Int temperature = 10;
32
33   adds Int getTemperature() {
34     return this.temperature;
35   }
36   modifies Unit gatherInformation() {
37     if (this.typeE == TemperatureSensor) {
38       Fut<Int> t = this.house!getTemperature();
39       await t?;
40       Int temp = t.get;
41       if (temp != this.temperature) {
42         this.temperature = temp;
43         this.notifyObservers(intToString(temp));
44       } else {
```

```

45         original();
46     }
47 }
48 }
49 }
50 delta DFireSensor;
51 uses House;
52 adds interface FireSensor extends Sensor {
53 }
54 modifies class Sensor adds FireSensor {
55     modifies Unit gatherInformation() {
56         if (this.typeE == FireSensor) {
57             Fut<Bool> f = this.house!getFire();
58             await f?;
59             Bool fire = f.get;
60             if (fire) {
61                 this.notifyObservers("fire");
62             }
63         } else {
64             original();
65         }
66     }
67 }

```

LISTING 4.9: Sensor type hierarchy

#### 4.3.1 Pattern

The example discussed above can be generalised into a pattern:

1. Define the type hierarchy.
2. Define an interface `TopI` for the top-level type.
3. Define a class `C` implementing `TopI`.
4. For each subtype define a delta which:
  - a) defines an interface `I` for the type extending the direct supertype's interface,
  - b) adds `I` to `C`,
  - c) modifies `C` by adding the methods defined in the type's interface.

#### 4.3.2 Multiple Method Implementations

The multiple delta approach as it is described up until now has one big problem: it often happens that different subtypes require the same method to behave differently and override the method implementation of the supertype. Since the proposed approach does not create two classes, only one implementation of a method is available. The attentive reader may already have noticed a solution to this problem in the `gatherInformation()` method of Listing 4.9. A variable of type `SensorType`

is passed as a parameter to the class. This variable should state the type of the type hierarchy this instantiation should have. So for instance, it is possible to have a variable of type `Sensor` to which a new instantiation of the class `Sensor` is assigned, with the new instantiation having `FireSensor` as the `SensorType` parameter. The instantiation will then be treated as being a `FireSensor`. Each delta then modifies the method it wants to override by first putting an if-statement to check whether the modification should be executed, based on the type and calling `original()` in the else-clause. Of course this is not an ideal solution since the wrong type can easily be passed in the parameter, either accidentally or maliciously. Sometimes it is desirable to call the implementation of the supertype while overriding the method. This is still possible by calling `original()` in the if-clause as well. Since only one of the subtypes' implementations should enter the if-clause, the original implementation of the supertype will eventually be called. To make sure a supertype will handle a call to `original()` correctly, the if-clause should respond to all the subtypes of that type as well. Since the subtypes will always be called first (as their deltas are applied later), this does not give rise to new problems.

Taking this problem into account, the pattern in Section 4.3.1 should be extended:

4. d) For every method that will be overridden use following pattern for the method's implementation:

```

1  ReturnType result;
2  if (this.typeE == <Type>) {
3      // <Type specific implementation here>
4  } else {
5      result = original();
6  }
7  return result;
```

The creation of an object now has the elements of object creation in Java. In Java the left hand side of the expression states the static type of the object, which defines the methods available for calling. The right hand side states the dynamic type and defines which method implementation will be used. In ABS the left hand side of the expression also states the static type of the object, i.e. which methods can be used. The dynamic type is defined by the type parameter since this parameter will be used to determine the functionality of the called methods.

Because the same method is used for all implementations, the method signature cannot be changed, so the return type and the type of the parameters are the same for all types in the hierarchy. Of course changing the dynamic type of returned objects is possible, but the static type is not alterable.

### 4.3.3 Discussion

The main advantage of the multiple delta approach over the delegation approach is the correct use of overridden/overriding methods. This was a major issue in the delegation approach. A second important advantage over the delegation approach is the reduction of code overhead. Since the multiple delta approach uses the same

class for all the different types, explicitly implementing the methods defined in the interfaces is only necessary when the behaviour of the supertype needs to be overridden. On the other hand, always modifying the same class is the biggest problem of this approach, since every instantiation contains the functionality of all the types in the type hierarchy. This is mainly a problem when different subtypes have a different behaviour for the same method. This can however be solved by using a combination of if-statements and calls to `original()`.

```
1 interface Sensor {
2 }
3 interface FireSensor extends Sensor {
4 }
5 interface MovementSensor extends Sensor {
6 }
7 class Sensor implements Sensor, FireSensor, MovementSensor {
8 }
9 {
10   Sensor s = new Sensor();
11   FireSensor fs = s; // Error
12   MovementSensor ms = fs; // Error
13   FireSensor fs2 = new Sensor();
14   Sensor s2 = fs2;
15   MovementSensor ms2 = fs2; // Error
16 }
```

LISTING 4.10: Reassigning objects to variables of different types

The fact that the class implements all interfaces poses no problem. Reassigning an object to a variable of another type is only possible according to the type hierarchy defined by the interfaces. For instance, in Listing 4.10 an object of the class `Sensor` is instantiated and assigned to a variable of the type `Sensor` (line 10). On lines 11 and 12 this object is reassigned to a variable of type `FireSensor` and `MovementSensor` respectively. These assignments are invalid, since these types are lower in the hierarchy than `Sensor`. On line 13 an object of the class `Sensor` is instantiated and assigned to a variable of type `FireSensor`, reassigning it to a variable of type `Sensor` (line 14) succeeds, because the interfaces state that a `FireSensor` is also a `Sensor` (due to the `extends` statement). Reassigning it to a variable of type `MovementSensor` (line 15) fails, because that relationship does not apply here.

The solution for the multiple method implementations proposed in Section 4.3.2 still has some issues. For instance, the if-statements needs to check that the provided type parameter is the type for the implementation of that level in the hierarchy as well as any level lower in the hierarchy. So for `Sensor` this would be Listing 4.11.

This cannot be trivially shortened, since the `data` construct in ABS does not support a hierarchical structure. One possible solution is combining all the checks in one function, this would then look like Listing 4.12 and the new if-statement looks like Listing 4.13.

```

1  if (typE == Sensor || typE == FireSensor || typE ==
    MovementSensor)

```

LISTING 4.11: if-statement for Sensor

```

1  def Bool isSensorType(SensorType toCheck, SensorType t) =
2    case toCheck {
3      t => True;
4      Sensor => True;
5      TemperatureSensor => case t {
6        TemperatureSensor => True;
7        _ => False;
8      };
9      FireSensor => case t {
10       FireSensor => True;
11       _ => False;
12     };
13   };

```

LISTING 4.12: Checking type using a function

```

1  if (isSensorType(this.typE, Sensor))

```

LISTING 4.13: if-statement for Sensor using a function

This does not solve the problem of course, but by concentrating the code in one place, it becomes more manageable. On the downside, functions cannot be changed using deltas, so the function to check types has to be written up front. However, this is the same for the types itself, since just like functions, data declarations cannot be modified or removed by deltas either, only added. Since ABS is still under development, it could be that modifying functions and/or data declarations will be possible in the future.

## 4.4 Conclusion

In this chapter, three approaches for code reuse in ABS were discussed. The delegation approach tries to mimic inheritance by delegating method calls to a superobject which represents its superclass. The biggest problem of this approach is that the superobjects have no knowledge of their subobjects and therefore of possible overriding of their methods. This may result in unexpected results since the programmer will expect the overriding method to be used instead of the overridden one. Also, a coding overhead is present due to the delegation. Every non-overridden method still has to be implemented in every class, calling the same method in its superobject. A memory problem exists as well, since an object lower in the type hierarchy is represented by one object for every level in the hierarchy. One positive

point about the delegation approach is that it is usable in most OOP languages since it uses main concepts of OOP (classes and interfaces).

The single type approach is a first step into solving the problems of the delegation approach. In the single type approach, one class is defined and deltas modify the functionality of this class. This solves the delegation and its associated coding overhead. Its simplicity makes it very easy to use, but also makes it useless in many cases. It can only be used when one type in the type hierarchy is needed throughout the entire program, a rare condition.

The third, or multiple delta approach solves the problem of the single type approach by exploiting the ABS typing system. Interfaces are again defined for every type and are added to the original class implementation in deltas. These deltas also modify the functionality of the class by adding new methods and overriding other methods using a specific pattern in order to handle multiple method implementations. Variables containing objects of this class have a specific type and only the methods defined in the interface of this type are available. The downside is that this pattern needs typing information which the programmer needs to fill in manually on creation of the object and methods always have the same unalterable signature.

## Chapter 5

# Evaluation of the ABS Delta-Oriented Programming Methodology

In this chapter the experience gained with ABS is discussed. Section 5.1 gives a short general experience with ABS, and Section 5.2 discusses development in ABS. In section 5.3 the performance of ABS is discussed, and Section 5.4 covers unit testing in ABS. Section 5.5 closes the chapter with a conclusion.

### 5.1 General Experience with ABS

Using Core ABS as a programming language will for most programmers not be a very big step as it is an object-oriented programming language with a syntax resembling e.g. Java. The major difference to use it as a programming language like Java is that classes require interfaces in order to be assignable to variables since the classes do not define types, their interfaces do. Classes just provide the implementation of methods.

The facet of Core ABS that will be new for most programmers is the concurrency model of ABS: using COGs, asynchronous method calls, futures, **await**, **suspend**, and **get** (although the new version of C# (C# 5) will use similar constructs for asynchronous method calls [12]). Using these features may be a bit tricky for novices to concurrency, as they may result in some unexpected results which will be discussed in Section 5.2.3. The problems existing here are common to concurrent programming languages and the reader is referred to literature focusing on the subject of concurrent programming [4, 21].

Using the languages for variability ( $\mu$ TVL, DML, CL, and PSL) will be new to almost all programmers. This does not imply that they are hard to use. These languages are very intuitive and a programmer new to the concepts of DOP, only needs a few examples to grasp the usage of these languages.

## 5.2 Developing in ABS

In this section development in ABS is discussed. It starts with some implementation guidelines, continues with patterns and best practices, some anti-patterns and code smells. Also some annoyances in ABS are discussed, some of these can be improved in later versions of ABS, others are the consequence of design decisions and cannot be changed.

### 5.2.1 Implementing a Software Product Line

When starting the implementation of an SPL in ABS, it is a good idea to begin with describing the feature model. This gives a first degree of granularity of the SPL.

A selection can be made from the feature model. This selection should be implemented in core ABS and forms the basis for the deltas. (For instance, in the implementation of the smart home, this core implementation contained all mandatory features and nothing more (so only the feature `Interface`). It was very small, consisting of only one interface and one class declaration with only one method.) It is possible to select more than the mandatory features, but using the code reuse approaches described in Chapter 3 becomes more difficult as some deltas will have to remove code instead of adding code, which may make it harder to reason about the results.

After the implementation in core ABS is finished, the deltas can be defined. They will provide a second degree of granularity as features can be split into several deltas and one delta can be used by several features or link features together by providing communication between them. The deltas are then linked to their respective features.

Products can be defined anytime after the feature model has been described. So the products, which will be the end products of the SPL, can be defined directly after defining the feature model. Other products which serve merely to test the implementation of deltas can be defined on the fly.

### 5.2.2 Patterns and Best Practices

#### Code Reuse

Chapter 4 gives an extensive overview of three patterns for code reuse: the delegation approach, the single type approach and the multiple delta approach.

#### Repeating Method

During the development of the smart home, while-loops were often used to have active objects simulate certain behaviour. For instance, the sensors need to check whether the state of the house has changed in fixed time intervals. This example is shown in Listing 5.1, with the `run()` method of the `Sensor` class.

In this piece of code a method call is inserted on line 7. The called method implements the execution steps which need to be performed every time the sensor



becomes active. This way, deltas can add to or modify this behaviour without the programmer having to rewrite the code required to keep the sensor active.

```
1  class Sensor implements Sensor {
2      ...
3      Unit run() {
4          sleeper = new cog Sleeper();
5          while (True) {
6              if (enabled) {
7                  this.gatherInformation();
8              }
9              Fut<Unit> s = sleeper!sleep(100);
10             await s?;
11         }
12     }
13     ...
14 }
```

LISTING 5.1: Run() method of the Sensor class

### 5.2.3 Code Smells and Anti-Patterns

#### Asynchronous Call - Await - Get

When an asynchronous call, an **await** statement on the future of that call and a **get** statement on that future occur consecutively (e.g. Listing 5.2) chances are that this is suboptimal. The asynchronous call should be moved up so that it gets executed as early as possible and the **await** and **get** statements should be deferred as long as possible. By splitting these statements, the time the method is idling because the asynchronous call has not returned yet is minimised. Of course it is not always possible to split these three statements, but if anything can be put between an asynchronous call and its respective **await** statement, it should be there.

```
1  ...
2  Fut<Int> f = foo!bar();
3  await f?;
4  Int i = f.get;
5  ...
```

LISTING 5.2: Asynchronous call - await - get

#### Await and Suspend

A programmer needs to use caution when adding **await** or **suspend** statements. These statements halt the execution of the method and allow calls to other methods to be made within the same COG. This could lead to unexpected results, since these

other method calls could change the value of a variable. Listing 5.3 gives an example of how things can go wrong.

```
1  interface Foo {
2      Unit setVal(Int i);
3      Int  getVal();
4      Unit fooBar();
5  }
6  interface Bar {}
7  class Foo implements Foo {
8      Int val = 0;
9      Unit setVal(Int i) {
10         this.val = i;
11     }
12     Int getVal() {
13         return this.val;
14     }
15     Unit fooBar() {
16         if (this.val < 5) {
17             // do some stuff
18             suspend;
19             Printer printer = new Printer();
20             printer.printS("Value was: " + intToString(this.val));
21         }
22     }
23 }
24 class Bar implements Bar {
25     Unit run() {
26         [Far]Foo foo = new cog Foo();
27         foo!fooBar();
28         foo!setVal(10);
29     }
30 }
```

LISTING 5.3: Problem with await and suspend

When the object of class Bar makes an object of Foo and calls the `fooBar()` method, `val` is still 0 and the `if` statement evaluates to `True`. The execution of the method is then suspended giving Bar the opportunity to call `setVal(10)`. This will cause `val` to be set to 10 and "Value was: 10" will be printed instead of the expected "Value was: 0". It is in no way certain that this will occur due to non-deterministic behaviour of the execution of tasks. This can make it harder to detect these problems as tests may succeed even though the problem still exists. In the described example, the problem can easily be solved by adding an `await` between lines 27 and 28 halting the execution of Bar until `fooBar` has finished running. However, scenarios exist where the call to `fooBar()` and `setValue(10)` come from different objects.

In general, using `suspend` and `await` statements between two uses of the same global variable should be avoided unless the programmer can be certain that the variable won't be changed by any other method call.

### 5.2.4 Annoyances and Shortcomings of ABS

#### Initialisation

In ABS, it is possible to add parameters to a class definition which can be used to initialise an object. The initialisation code for this is written in a nameless block in the class. It is however not possible to call methods from inside this block, and the block cannot be altered using deltas. As a consequence, the initialisation of objects cannot be altered in any way using deltas. If method invocation was possible inside this initialisation block, an initialisation method could be written which could then be modified using deltas.

The `run()` method of a class could be used to overcome this problem. If a class has a `run()` method, this method will be executed as a new task, so there is no guarantee when it will be executed. If it is in the same COG as the current task, it has to wait until the current task has finished or is suspended, and then other tasks could still be executed first. If the `run()` method is not executed before the new object is used, the object may behave unexpectedly.

#### Intermediate Results

The ABS compiler does not allow the programmer to perform a method call on the result of a method call. It only allows calling methods on variables. Hence, intermediate results always have to be stored in a variable. Listing 5.4 gives an example of code that is not accepted by the compiler, while Listing 5.5 shows how the code should be written.

```

1  interface Foo {
2    Bar getBar();
3  }
4  interface Bar {
5    Foo getFoo();
6  }
7  class Foo implements Foo {
8    Bar getBar() {
9      return new Bar();
10   }
11 }
12 class Bar implements Bar {
13   Foo getFoo() {
14       return new Foo();
15   }
16 }
17 {
18   Foo foo = new Foo();
19   Foo foo2 =
20       foo.getBar().getFoo();
    // Syntax error!

```

LISTING 5.4: No intermediate results

```

1  interface Foo {
2    Bar getBar();
3  }
4  interface Bar {
5    Foo getFoo();
6  }
7  class Foo implements Foo {
8    Bar getBar() {
9      return new Bar();
10   }
11 }
12 class Bar implements Bar {
13   Foo getFoo() {
14       return new Foo();
15   }
16 }
17 {
18   Foo foo = new Foo();
19   Bar bar = foo.getBar();
20   Foo foo2 = bar.getFoo();
21 }

```

LISTING 5.5: With intermediate results

This is an annoying trait of ABS since lots of variables need to be created which serve to nothing more than hold a result between two method executions. There is no apparent reason why calling methods directly on other methods results should not be possible.

### Conditional Statements

ABS does not allow the programmer to call methods inside a conditional statement. Only statements without side-effects are valid. Listing 5.6 gives an example of code that is not accepted by the compiler, Listing 5.7 shows how this code should be written.

<pre> 1  interface Foo { 2    Bool getBool(); 3  } 4  class Foo implements Foo { 5    Bool getBool() { 6      return True; 7    } 8  } 9  { 10   Foo foo = new Foo(); 11   if (foo.getBool()) { 12     // Do something 13   } 14 }</pre>	<pre> 1  interface Foo { 2    Bool getBool(); 3  } 4  class Foo implements Foo { 5    Bool getBool() { 6      return True; 7    } 8  } 9  { 10   Foo foo = new Foo(); 11   Bool test =         foo.getBool(); 12   if (test) { 13     // Do something 14   } 15 }</pre>
--	---

LISTING 5.6: Conditional statement with method call

LISTING 5.7: Conditional statement without method call

This looks like the same problem as the intermediate results problem of the previous section. However in this case a reason for this exists. A quote from Rudi Schlatter, one of the developers of ABS [19]:

*ABS distinguishes between pure (side-effect-free) and side-effecting expressions, and disallows side effects in lots of places. This is for the benefit of the modelling aspect of the ABS language / methodology, making it much easier to develop proof theories and proof checkers. Also, it makes the semantics a bit easier: consider*

```
if (b && foo.someMethod(this)) { ... }
```

*where someMethod calls back and changes b. These things can be specified, but make language semantics messy and difficult to analyse.*

### Import and Export Statements in Deltas

Core ABS code is written inside a module. To make interfaces and classes available outside this module, the **export** statement is used. To use interfaces and classes

from other modules, the `import` statement is used. They can both select a certain interfaces or class by stating its name or use “`*`” to export/import everything.

However, it is not possible to incorporate import and export statements in deltas. It could sometimes be handy to add new import and export statements in a delta since it is possible that this delta uses a new module or that it introduces new interfaces and classes which a delta in another module needs to use. It is possible to solve this problem by using “`import *;`” and “`export * from x;`” statements, but this requires all interfaces and classes of a module to be imported or exported, which is usually not desirable.

### 5.2.5 Static Behaviour in ABS

Static binding (where method calls are bound to the declared type) is not possible in ABS, since ABS does not support static methods. However, ABS has a functional core and consequently supports functions, which are somewhat like static methods, as they also do not require the initialisation of an object. Functions also do not need to be declared inside a class. This also means that functions cannot use variables outside their local scope, so global variables are not available. So as long as global variables are not required, functions suffice and are even preferable, since functions are side effect free and static methods may have side effects.

Static methods could be made available in ABS if it was allowed to implement certain methods in the interface declaration. These methods would then be the same for every implementation of that interface, and could not depend on the run-time class of an object since it has no reference to actual implementations of the interface. A code example of this is given in Listing 5.8 for a square. The interface definition states that implementations of the `Square` interface needs to implement the `getArea()` method, it also defines a static `calculateArea(Int)` method, which calculates the area of a square based on the parameter provided when calling the method.

```
1  interface Square {
2      Int getArea();
3      Int calculateArea(Int a) {
4          return a*a;
5      }
6  }
7  {
8      Int area = Square.calculateArea(4);
9  }
```

LISTING 5.8: Example of how a static method declaration could look like

As mentioned before, for full functionality of static methods, static variables would be needed as well, otherwise functions could be used. This is where problems could arise when trying to use the delegation approach to code reuse described in Section 4.1. If both an object and its superobject implement an interface with a

static variable, this static variable is present in both objects. A solution could be to always use the static variable of the superobject since both classes can have access to this variable (either directly or through a getter/setter-pair). This would mean that the subobject has a field which would never be accessed and thus wastes memory.

### 5.3 Performance of ABS Execution

In Section 4.1 a test was performed to measure the difference in execution times between regular inheritance and the delegation approach when applied to Java. At first the test was set up so that the delegation approach was applied in ABS code (which is translated to Java) and compared to an inheritance implementation in Java. It was decided that this was not a good measure for this test and that both the delegation approach and regular inheritance had to be implemented natively in Java, with the results in Section 4.1 as a result. It did however give rise to a new test on the difference in execution times between an ABS implementation and a native Java implementation. ABS has different back-ends which translate ABS code to another programming language. Java is the most mature and supported of these back-ends and was therefore chosen as reference.

The new test was set up to measure the execution time of calling a method in both ABS (translated to Java) and Java. The most important code for the ABS speed test can be found in Listing 5.9. For Java, the multiplication (line 10) is surrounded by a loop running ten times. This is done to get a representable measurement since the execution in Java was too fast to measure accurately otherwise. This code was executed ten times and the results written to file. The total code can be found in Appendix A.2. The analogous code for the Java speed test can be found in Appendix A.3.

```
1 Multiplier m = new Multiplier();
2 Int x = 1;
3 int y = 0;
4 Int start = 0;
5 Int end = 0;
6 List<Int> times = Nil;
7 while(x < 100000) {
8     start = currentms();
9     while (y < x) {
10         m.multiply(123456789 + y, 987654321 + x);
11         y = y + 1;
12     }
13     end = currentms();
14     times = Cons(end - start, times);
15     y = 0;
16     x = x + 1;
17 }
```

LISTING 5.9: ABS speed test

### 5.3.1 Results

The average results of the ten executions are shown, with data points in red and the trend line shown in green, in Figure 5.1 for ABS and in Figure 5.2 for Java.

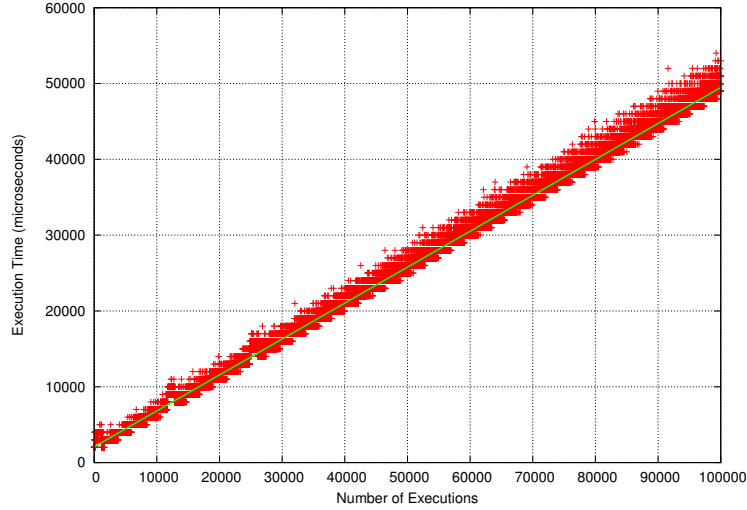


FIGURE 5.1: Average of ten executions of the ABS performance test

Figure 5.1 clearly shows a linearly increasing execution time in function of the number of method calls, which is expected. The y-intercept of the trend line lies at 2077.26 and the trend line has slope of 0.473966. The y-intercept indicates a rather large start-up time of 2077.26  $\mu\text{s}$ . The slope indicates that per 2 method calls, the execution time increases with about 1  $\mu\text{s}$ .

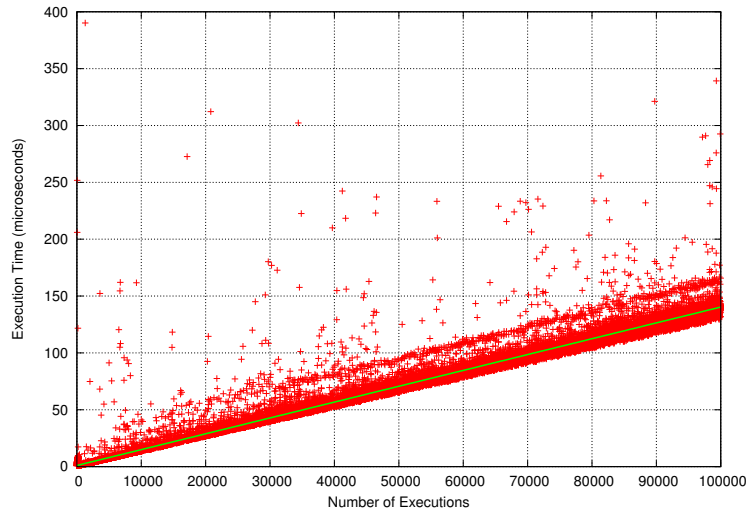


FIGURE 5.2: Average of ten executions of the Java performance test

Figure 5.2 also shows a linearly increasing execution time in function of number of method calls. Here the trend line has an y-intercept of 1.03522 and a slope of 0.00139297, meaning a start-up time of 1.03522  $\mu$ s and an increase of execution time of roughly 1  $\mu$ s per 700 executions.

### 5.3.2 Discussion

As is already clear from the difference in scale on the y-axis (0 to 400 for Java compared to 0 to 60,000 for ABS), the execution of a program written in ABS is much slower than its equivalent program written in Java. The graph for Java shows much more outliers compared to the graph for ABS, which has none. This is due to the greater speed of the Java program. Interrupts in the program have a much greater impact on measured execution time when the normal execution time is so small.

The graphs show that ABS and Java have the same behaviour with increasing execution indicating that ABS is in fact slower and this is not due to a possible large start-up time for ABS. The most probable cause for this slow execution lies in the way the ABS code is transformed into Java. Listing 5.10 shows an excerpt from the performed test for ABS transformed in Java. Lines 3 and 8 show that integers in ABS are transformed to ABSInteger objects. This is done by parsing the textual representation of the integer. As the parsing of strings is slow, this could explain the slow execution in ABS. Another possible answer is the large amount of method calls happening in the transformed code. However, as can be concluded from the test performed in Section 4.1.4, increasing the number of method calls increases the execution time only slightly, certainly not 6 orders of magnitude.

```

1  if (__ABS_getRuntime().debuggingEnabled())
2    __ABS_getRuntime().nextStep("abs/SpeedTest.abs",26);
3  abs.backend.java.lib.types.ABSInteger a = abs.backend.java.lib.
    types.ABSInteger.fromString("123456789");
4  if (__ABS_getRuntime().debuggingEnabled())
5    __ABS_getRuntime().getCurrentTask().setLocalVariable("a",a);
6  if (__ABS_getRuntime().debuggingEnabled())
7    __ABS_getRuntime().nextStep("abs/SpeedTest.abs",27);
8  abs.backend.java.lib.types.ABSInteger b = abs.backend.java.lib.
    types.ABSInteger.fromString("987654321");
9  if (__ABS_getRuntime().debuggingEnabled())
10   __ABS_getRuntime().getCurrentTask().setLocalVariable("b",b);

```

LISTING 5.10: ABS code transformed in Java

ABS aims at specification and analysis. Generation of efficient code is not a primary goal of the project. The results of this test may however encourage the developers of ABS to keep efficiency in mind when working on ABS in order to reduce the difference between the execution time of an ABS program and the execution time of an analogous native program.



## 5.4 Unit Tests

It is possible to test implementations in ABS by performing unit tests. The `AbsUnit` module was written by ABS developers for this purpose. Unit tests are used to test a unit of functionality of a system, usually at the level of public class methods [20].

Unit tests in ABS generally look like the code in Listing 5.11. The `[Fixture]` annotation defines the fixture for the test and is added to the test interface. The `[Test]` annotation specifies which methods in the interface are tests, while `[DataPoint]` defines which methods in the interface provide data input for test methods. Finally, the `[Suite]` annotation is added to the class implementing the tests. The actual tests are performed by making method calls to an instance of `ABSAssert` which is provided by the `AbsUnit` module.

```

1  module Test;
2  import * from AbsUnit;
3  [Fixture] interface TestInterface {
4    [Test] Unit test1(Int ints);
5    [Test] Unit test2();
6    [DataPoint] Set<Int> getInts();
7  }
8  [Suite]
9  class TestImplementation implements TestInterface {
10     ABSAssert aut;
11     Unit run() {
12         aut = new ABSAssertImpl();
13     }
14     Set<Int> getInts() {
15         return Insert(1, Insert(2, EmptySet));
16     }
17     Unit test1(Int ints) {
18         TestedClass tc = new TestedClass();
19         Int i = tc.testedMethod1();
20         Comparator cmp = new IntComparator(i, ints);
21         aut.assertNotEquals(cmp);
22     }
23     Unit test2() {
24         TestedClass tc = new TestedClass();
25         String s = tc.testedMethod2();
26         Comparator cmp = new StringComparator(s, "expected string");
27         aut.assertEquals(cmp);
28     }
29 }

```

LISTING 5.11: ABSUnit example

By running a test runner generator, code is generated, which can be run like any other ABS program. The generated code will call all tests it found in the program asynchronously (so all tests will run concurrently). Tests which take parameters are called once for every item in the respective set. So in the example (Listing 5.11), `test1(Int)` is called twice, once with 1 as parameter and once

with 2 as parameter. Listing 5.12 shows code generated by the test runner generator for the tests in Listing 5.11.

```

1  module AbsUnit.TestRunner;
2  import * from Test;
3  import * from ABS.StdLib;
4  import * from AbsUnit;
5  {
6    Set<Fut<Unit>> futs = EmptySet;
7    Fut<Unit> fut;
8    TestInterface testImplementationdataPoint = new
        TestImplementation();
9    Set<Int> testImplementationdataPointSet =
        testImplementationdataPoint.getInts();
10   TestInterface testImplementation0 = new cog
        TestImplementation();
11   fut = testImplementation0!test2();
12   futs = Insert(fut, futs);
13   while (hasNext(testImplementationdataPointSet)) {
14     Pair<Set<Int>, Int> nt = next(
        testImplementationdataPointSet);
15     Int d = snd(nt);
16     testImplementationdataPointSet = fst(nt);
17     TestInterface testImplementation1 = new cog
        TestImplementation();
18     fut = testImplementation1!test1(d);
19     futs = Insert(fut, futs);
20   }
21   while (hasNext(futs)) {
22     Pair<Set<Fut<Unit>>, Fut<Unit>> nt = next(futs);
23     fut = snd(nt);
24     futs = fst(nt);
25     fut.get;
26   }
27 }

```

LISTING 5.12: Generated test runner code

First a set of futures is defined (line 6) which will contain the futures created by calling the test methods. Next, the set of data points is generated (line 9). After that the actual tests are run. Line 10 to line 12 executes `test2()` and adds the future of the result to the future set. Line 13 to line 20 runs `test1(Int)` for each value in the data point set and adds the future of each of these calls to the future set. In Line 21 to 26 a `get` is performed on each future in the set to make sure each test has finished.

#### 5.4.1 Testing Non-Public Methods

As mentioned earlier, unit tests normally are used to test public methods. However, in ABS it is also possible to test private methods due to the power of deltas. If a method is private to a class, a delta can be written which adds this method to an

interface of the class. The test case can now access the method in order to test its functionality. The test itself will also have to be written in a delta. Listing 5.13 gives an example of this. Being able to test private methods is a major advantage of ABS unit testing over unit testing where deltas are not available (e.g. JUnit).

```

1  module Test;
2  import * from AbsUnit;
3  interface Foo {
4  }
5  class Foo implements Foo {
6      Int bar() {
7          return 5;
8      }
9  }
10 delta DFooTest;
11 uses Test;
12 modifies interface Foo {
13     adds Int bar();
14 }
15 adds [Fixture] interface FooTest {
16     [Test] testBar();
17 }
18 adds [Suite] class FooTestImpl implements FooTest {
19     ABSAssert aut;
20     Unit run() {
21         aut = new ABSAssertImpl();
22     }
23     Unit testBar() {
24         Foo foo = new Foo();
25         Int bar = foo.bar();
26         Comparator cmp = new IntComparator(bar, 5);
27         aut.assertEquals(cmp);
28     }
29 }

```

LISTING 5.13: Testing a private method

## 5.4.2 Discussion

### Error Messages

When the code generated by the test runner generator is executed, the program exits without any output if all assertions succeed. If an assertion fails, an error message is shown. These error messages look like Listing 5.14.

```

1  Error in abs.backend.java.lib.runtime.ABSRuntime@635b9e68:
2  absunit.abs:68:4: Assertion failed

```

LISTING 5.14: ABSUnit error message

A message like this is all but informative. The line of code the error message points to (`absunit.abs:68:4`) refers to the `assert` statement in the `ABSUnit` library and consequently gives no information on what went wrong, only that something did. This is OK when only a very small amount of tests are present, but when the number of tests increases, these error messages are no longer feasible. To be truly useful, the error messages should contain at least the name of the file containing the test and the name of the test. In the best case the line number and parameters of the call in the test are included. For instance, should `test1(Int)` in Listing 5.11 fail, the error message could look something like Listing 5.15.

```
1 Assertion Error: Test.abs:39 test1(Int)
2 assertEquals failed for IntComparator(2, 2)
```

LISTING 5.15: Improved error message

With this information the programmer immediately knows exactly which test failed. To be able to generate this error message, some reflection is needed in order to know the test method that failed and the file it resides in. The information from the `Comparator` could be extracted by using a `toString()` method. This method would then return the name of the `Comparator` implementation and its parameters. Of course, showing the parameters is only possible if they too have a textual representation.

### Test Runner Generator

It seems that the test runner generator still has some bugs. Listing 5.16 shows the console output of running the test runner generator on a product in the smart home product line for which a test was implemented. It clearly shows that the delta is applied and the interface and class are added to the program (lines 2 and 3). However, line 5 shows that generating the test runner failed. According to the code of the test runner generator, this error message means that no unit tests were found which is contradictory with the application of the delta.

```
1 ...
2 *** applying ModuleModifier AddInterfaceModifier(
    SmartHomeInterfaceTest)
3 *** applying ModuleModifier AddClassModifier(
    SmartHomeInterfaceTestImpl)
4 ...
5 An error occurred during compilation: Cannot generate test
    runner
6 make: *** [unit] Error 1
```

LISTING 5.16: Output from test runner generator

In order to identify the possible cause(s) of this bug, Some test implementations were set up:

- Testing a class fully defined inside a delta,
- Modifying the class under test in a delta, before and after the application of the delta with the test in it. With the modifications in the same and in another delta module.
- Modifying the interface of the class under test in a delta, before and after the application of the delta with the test in it. With the modifications in the same and in another delta module.

Each to no avail, none of these circumstances triggered the failure of generating the test runner. The cause(s) of this bug thus still remain unknown and are subject to further investigation.

## 5.5 Conclusion

In this chapter, the ABS language was evaluated. ABS incorporates constructs for concurrent programming (futures, **await**, **suspend** and **get**) which are fairly new to most developers, although they are also getting picked up by commercial programming languages (e.g C# 5 [12]). Together with the languages for variability ( $\mu$ TVL, DML, CL and PSL) this makes ABS more challenging to learn than regular OOP languages. The constructs are fairly intuitive however and after playing around with them for a bit, they soon start to become clear. Implementing in ABS can therefore be learned with relative ease, although some caution needs to be taken when using the concurrent programming constructs. These are common to the other concurrent programming languages though.

ABS is still young and this can also be seen in some annoyances the language has: initialisation blocks cannot be changed by deltas, results of method calls need to be stored in variables before other method calls can be performed on them, conditional statements do not allow side effects (this is a conscious choice) and import/export statements cannot be changed using deltas. ABS provides functions which can be used as a kind of static methods. They however provide less functionality than static methods as they are not defined inside classes and global static variables are thus not available.

When ABS was developed, performance was not a great issue. This is clear from the performance test carried out, which compared the execution of a short program written in ABS to an analogous program written in Java. This test showed ABS to be three orders of magnitude slower than Java.

Unit testing for ABS has also been evaluated. ABSUnit has a very nice feature for testing non-public methods, which is in general not possible, thanks to the deltas which make it possible to make private methods public by adding them to one of the class's interface. The error messages the unit tests give could be improved to give

## 5. EVALUATION OF THE ABS DELTA-ORIENTED PROGRAMMING METHODOLOGY

---

developers more information on which test failed and the test runner generator has some unresolved issues breaking it in some unidentified cases.

## Chapter 6

# Conclusion

In this thesis a product line for a smart home was described. This product line served as basis for the evaluation of delta modelling in the ABS language.

Three approaches to code reuse in ABS were introduced: the delegation approach, the single type approach and the multiple delta approach. The delegation approach has as the advantage that it can be applied to most object-oriented programming languages since it is based on interfaces and classes, two important concepts in OOP. It does however have issues making it unsuitable in some cases. The single type approach alleviates the problems with the delegation approach but has the major downside of only allowing one type of a type hierarchy to be available throughout an entire program. The multiple delta approach fixes this problem. It also has a minor issue with multiple method implementations, but a workarounds exist for this: using a type parameter on construction of an object. Although this workaround is not optimal since all the possible types are available as input for the parameter, even if the type itself is not available (because its delta was not applied), it makes the multiple delta approach work.

After the introduction of the three code reuse approaches, the ABS language in itself was evaluated. ABS showed to have a set of features concerning concurrency and variability, which were fairly new. This does not mean that is very hard to learn ABS when another OOP language is already known, since the features ABS has are rather intuitive. ABS is still young which showed when developing using it. Some things are not possible that should be (like calling methods directly on other methods' results), but these are things that can be improved in future versions of ABS. It was a goal of this thesis to expose weaknesses and limitations of ABS and to propose improvements, which was achieved with respect to object initialisation, intermediate method results, etc.

The goal of discovering design patterns was also met by introducing the code reuse approaches. This took a lot of time; as a consequence the code smells, anti-patterns and also more general design patterns were neglected a bit.

### **Future Work**

This thesis did not make use of the possibility to add attributes to deltas and features. This possibility is consequently not evaluated. It may however make many more things possible and this subject is worthwhile to put more research to.

A second subject that was not included in the thesis, but is open to investigation is a comparison of delta-oriented programming to aspect-oriented and feature-oriented programming. These three programming paradigms each have their advantages and disadvantages and it could be interesting to see in which circumstances it is best to use each of these paradigms or even if a combination of them is possible.



# Bibliography

- [1] Sven Apel and Don Batory. When to use features and aspects? A case study. In *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, pages 59–68, New York, NY, USA, 2006. ACM.
- [2] Don Batory, David Benavides, and Antonio Ruiz-Cortes. Automated analysis of feature models: challenges ahead. *Commun. ACM*, 49(12):45–47, December 2006.
- [3] Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.
- [4] Jean-Pierre Briot. Object-oriented concurrent programming: Introducing a new programming methodology. In *Proceedings of the 7th International Meeting of Young Computer Scientists*, IMYCS'92. Gordon & Breach, 1993.
- [5] Dave Clarke, Stijn de Gouw, Reiner Hähnle, Einar Broch Johnson, Ilham W. Kurnia, and Radu Mushevi. Deliverable D1.2 full ABS modelling framework, March 2011.
- [6] Dave Clarke, Nikolay Diakov, Reiner Hähnle, Einar Johnsen, Ina Schaefer, Jan Schäfer, Rudolf Schlatter, and Peter Wong. Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In Marco Bernardo and Valérie Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 417–457. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-21455-4\_13.
- [7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, September 2007.
- [8] Reiner Hähnle, Einar Broch Johnsen, Bjarte M. Østfold, Jan Schäfer, Martin Steffen, and Arild B. Torjusen. Deliverable D1.1A report on the core ABS language and methodology: Part A, April 2010.
- [9] Paul-Th. Kandzia. Nonmonotonic reasoning in FLORID. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Logic Programming And Nonmonotonic Reasoning*, volume 1265 of *Lecture Notes in Computer Science*, pages 399–409. Springer Berlin / Heidelberg, 1997. 10.1007/3-540-63255-7\_30.

- [10] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.
- [11] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of Aspectj. In Jorgen Knudsen, editor, *ECOOP 2001 Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45337-7\_18.
- [12] Eric Lippert. Asynchrony in C# 5, part one. <http://blogs.msdn.com/b/ericlippert/archive/2010/10/28/asynchrony-in-c-5-part-one.aspx>. viewed on May 22nd 2012.
- [13] Klaus Pohl, Günther Böckle, and Frank van der Linden. *Software Product Line Engineering*. Springer, 2005.
- [14] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, chapter 18, pages 419–443. Springer-Verlag, Berlin/Heidelberg, 1997.
- [15] Ina Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, 2010.
- [16] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 77–91. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15579-6\_6.
- [17] Ina Schaefer and Ferruccio Damiani. Pure delta-oriented programming. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, FOSD '10, pages 49–56, New York, NY, USA, 2010. ACM.
- [18] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: generalizing active objects to concurrent components. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 275–299, Berlin, Heidelberg, 2010. Springer-Verlag.
- [19] Rudi Schlatte. HATS mailing list, May 2012.
- [20] Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte. The ABS tool suite: Modelling, executing and analysing distributed adaptable object-oriented systems. Submitted for publication, September 2011.

- [21] A. Yonezawa and M. Tokoro. *Object-oriented concurrent programming*. The MIT Press, January 1986.



# Appendices



# Appendix A

## Source Code Excerpts

### A.1 Test of Execution Time of Delegation vs. Inheritance

```
1  public class Main {
2      public static void main(String[] args) {
3          long max = Long.MAX_VALUE / 10000000000;
4          System.out.println(max);
5          int runs = 10;
6          long start = 0;
7          long end = 0;
8          long diff1 = 0;
9          long diff2 = 0;
10         long diff3 = 0;
11         long diff4 = 0;
12         for (int i = 0; i < runs; i++) {
13             // Test for 3 levels of delegation
14             DSS dss = new DSS();
15             start = System.nanoTime();
16             for (long x = 0; x < max; x++) {
17                 dss.multiply(123456789, 987654321);
18             }
19             end = System.nanoTime();
20             diff1 += end - start;
21             // Test for 3 levels of inheritance
22             ISS iss = new ISS();
23             start = System.nanoTime();
24             for (long x = 0; x < max; x++) {
25                 iss.multiply(123456789, 987654321);
26             }
27             end = System.nanoTime();
28             diff2 += end - start;
29             // Test for 10 levels of delegation
30             DSSSSSSSSS dssssssssss = new DSSSSSSSSS();
31             start = System.nanoTime();
32             for (long x = 0; x < max; x++) {
33                 dssssssssss.multiply(123456789, 987654321);
```

## A. SOURCE CODE EXCERPTS

---

```
34     }
35     end = System.nanoTime();
36     diff3 += end - start;
37     // Test for 10 levels of inheritance
38     ISSSSSSSSS issssssssss = new ISSSSSSSSS();
39     start = System.nanoTime();
40     for (long x = 0; x < max; x++) {
41         issssssssss.multiply(123456789, 987654321);
42     }
43     end = System.nanoTime();
44     diff4 += end - start;
45 }
46 // Output the results
47 System.out.println("With delegation lvl 3: " + diff1/runs);
48 System.out.println("With inheritance lvl 3: " + diff2/runs);
49 ;
50 System.out.println("With delegation lvl 10: " + diff3/runs);
51 ;
52 System.out.println("With inheritance lvl 10: " + diff4/runs);
53 }
54 }
```

LISTING A.1: Test program of the different sensor types

## A.2 Test of ABS Execution Time

```
1 module SpeedTest;
2 import * from Foreign;
3 import * from Multiplier;
4 {
5     Int runs = 0;
6     while (runs < 10) {
7         Multiplier m = new Multiplier();
8         Int x = 1;
9         Int y = 0;
10        Int start = 0;
11        Int end = 0;
12        List<Int> times = Nil;
13        Int startTotal = currenttms();
14        while (x < 100000) {
15            // Track execution time of number of method calls
16            start = currenttms();
17            while (y < x) {
18                m.multiply(123456789 + y, 987654321 + x);
19                y = y + 1;
20            }
21            end = currenttms();
22            times = Cons(end - start, times);
23            y = 0;
24            x = x + 1;
25        }
26        Int endTotal = currenttms();
27        // Write results to file
```



```

28     FileWriter fw = new FileWriter();
29     fw.setFileName("results" + intToString(runs) + ".dat");
30     List<Int> temp = Nil;
31     while (~isEmpty(times)) {
32         temp = Cons(head(times), temp);
33         times = tail(times);
34     }
35     times = temp;
36     while (~isEmpty(times)) {
37         Int toWrite = head(times);
38         times = tail(times);
39         fw.write(intToString(toWrite));
40     }
41     fw.flush();
42     runs = runs + 1;
43 }
44 }

```

LISTING A.2: Test of ABS execution time

### A.3 Test of Java Execution Time

```

1  import java.io.File;
2  public class Main {
3      public static void main(String[] args) {
4          int runs = 0;
5          while (runs < 10) {
6              Multiplier m = new Multiplier();
7              int x = 1;
8              int y = 0;
9              long start = 0;
10             long end = 0;
11             Vector<Long> times = new Vector<Long>();
12             while (x < 100000) {
13                 // Track execution time of number of method calls
14                 start = System.nanoTime();
15                 while (y < x) {
16                     for (int i = 0; i < 10; i++) {
17                         m.multiply(123456789 + y, 987654321 + x);
18                     }
19                     y = y + 1;
20                 }
21                 end = System.nanoTime();
22                 times.add(end - start);
23                 y = 0;
24                 x = x + 1;
25             }
26             // Write results to file
27             write("results" + runs + ".dat", times);
28             runs = runs + 1;
29         }
30     }
31     /**
32     * Writes elements to a file

```

## A. SOURCE CODE EXCERPTS

---

```
33  * @param name The name of the file to write to
34  * @param toWrite The elements to write to the file
35  */
36  public static void write(String name, Vector<?> toWrite) {
37      File file = new File(name);
38      try {
39          FileWriter fw = new FileWriter(file);
40          for (int i = 0; i < toWrite.size(); i++) {
41              fw.write(toWrite.get(i) + "\n");
42          }
43          fw.flush();
44      } catch (IOException e) {
45          e.printStackTrace();
46      }
47  }
48 }
```

LISTING A.3: Test of Java execution time

## Appendix B

### Poster





KATHOLIEKE UNIVERSITEIT  
**LEUVEN**

FACULTEIT  
INGENIEURSWETENSCHAPPEN

Master  
Computer-  
wetenschappen

Masterproef  
Wouter Suyen

Promotor  
Dave Clarke

Academiejaar  
2011-2012



# Code reuse in the ABS Language

## Context

- **Software Product Line Engineering**  
Software product lines consist of a set of similar products. They have a common core, but each product has a different set of features. These features are units of functionality in the software system. They represent requirements of the product, specified when designing the system, and provide potential configuration options. Selecting different features thus effectively leads to different end products.
- **ABS Language**  
The HATS project is concerned with highly adaptable software which at the same time needs to be very trustworthy. It develops a formal method for the design, analysis and implementation of such systems. At the core of the HATS project lies the Abstract Behaviour Specification (ABS) Language. The Full ABS Language actually consists of five languages which can describe an entire product line when combined. A core concept in these languages is delta modelling. In delta modelling a distinction is made between the core implementation containing the common code to each product, which is provided by the Core ABS Language, and deltas, which specify changes that have to be made to the core implementation and are specific to some feature(s). Defining the deltas and features and linking them together as well as specifying the end products is achieved by the other four languages.



## Goals

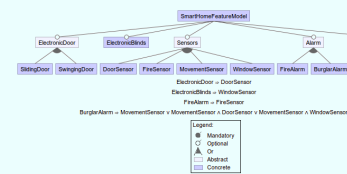
In spite of being object-oriented, the Core ABS language does not support the widely used notion of inheritance. This leads to a problem in code reuse which in most object-oriented languages is achieved using code inheritance. The subject of this research is consequently to develop methodologies for code reuse in the ABS Language.

## Applications

The applications of this research are straightforward. Being able to reuse code reduces the workload for the programmer by creating a smaller typing or copy/paste overhead hereby saving time, energy and development cost. It also reduces errors in consistency between the places where the code is used because the code only needs to be adjusted in one place.

## Case Study

In order to discover the possibilities of code reuse in the ABS Language, a case study is being performed. The implementation of a 'Smart Home' was chosen based on the 'Home Automation Model' in the book "Software Product Line Engineering" by Klaus Pohl, Günther Böckle and Frank van der Linden. The picture at the bottom of this frame shows the feature model (the possible features in the product line and their constraints) of the product line. A nice advantage of the Smart Home is that the number of features in the model is an amount that is small enough to be manageable in a research project or too small to not give any results.



## Preliminary Results

In the ABS Language, types are defined by interfaces which are implemented by classes and although no inheritance is available, interfaces can still extend each other, allowing type hierarchies to be created. By using this handy feature and the delegation pattern, code reuse can be accomplished. An object of certain type can have a variable containing an object of its super type. If the implementation of the super type does not need to be overridden, the object can just call the method in the super type. Because type hierarchies are supported, it is possible to have a variable of a certain type which contains an object implementing an interface which extends (or is) the interface of the variable type. This results in a dynamic binding-like behaviour of the ABS Language since the method implementation of the contained object is used; it has no notion of any other implementation because there is no code inheritance.

## Future Research

It would be feasible to extract a pattern that makes use of the deltas provided by the ABS Language out of the implementation using the delegation approach. Hence the Smart Home will be further implemented and analysed in order to discover such a pattern.

At the moment the behaviour of static binding has not yet been achieved using the delegation approach. Although in most cases dynamic binding is feasible, sometimes there is a need for static binding so this is worth looking into.



## Appendix C

### Article





# Code Reuse in the ABS Language

Wouter Seyen

Department of Computer Sciences

Faculty of Engineering

Katholieke Universiteit Leuven

Email: wouter.seyen@student.kuleuven.be

**Abstract**—The ABS Modelling Framework supports delta-modelling, which opens a path to code reuse. This paper presents three approaches to achieve code reuse: the delegation approach, the single type approach and the multiple delta approach. The first one is applicable to most object-oriented programming languages as it does not rely on deltas, the second one relies on deltas but is only applicable when one type of a type hierarchy is needed in a product, the last one also relies on deltas and is the most usable approach to code reuse of the three.

**Index Terms**—ABS, delta modelling, code reuse, design patterns

## I. INTRODUCTION

The ABS Modelling Framework is being developed in the context of the HATS (Highly Adaptable and Trustworthy Software using Formal Models) European project [3, p. 2], and supports developing software product line (SPL) systems following established software product line engineering (SPLE) practices, e.g. feature-oriented development. It allows the precise modelling and analysis of component-based distributed concurrent systems, focusing on their functionality while separating from the concerns such as concrete resources, deployment scenarios and scheduling policies [13].

The abstract behaviour specification language (ABS) is a language which has a hybrid functional and object-oriented core. It comes with extensions that support the development of systems that are adaptable to diversified requirements as well as to future changes and yet is capable to maintain a high level of trustworthiness [13]. It makes use of the delta-oriented programming (DOP) paradigm [11] in which a core product is implemented and deltas are applied to this core in order to get different end-products based on selected features. ABS consists of five languages to support this: (a) *Core ABS* which is used to describe the core behavioural modules and can be used as a regular object-oriented programming language, (b)  $\mu$ TVL or Micro Textual Variability Language, which is used to describe feature models using a textual representation, (c) *DML* or Delta Modelling Language, which is used to describe the deltas that are applied to the core product in order to change it, (d) *CL* or Product Line Configuration Language, which is used to link deltas to features, the deltas belonging to a feature are applied if the feature is selected, (e) *PSL* or Product Selection Language, which is used to define products by selecting features.

Core ABS is a subset of the full ABS language and does not in itself address SPL development, but forms a basis for

extensions which will capture SPL artifacts such as features and feature integration [7]. DOP was one of the extensions added on top of Core ABS in order to support SPLE. DOP was chosen as a research project for ABS over inheritance because it was fairly new and the developers of ABS wanted to see how well it fared in practice. ABS does not support inheritance, but code reuse can be achieved using deltas, as will be shown in this paper.

In this paper, first an introduction to ABS is given (Section II). Next, three approaches to code reuse in ABS will be discussed: the delegation approach (Section III), the single type approach (Section IV), and the multiple delta approach (Section V). The last two approaches use deltas. Section VII gives a general conclusion of the paper.

## II. THE ABS LANGUAGE

Section I already introduced the five languages which make up ABS. In this section they will be discussed a bit more in depth.

### A. Core ABS

Core ABS is the language used for specifying the core behavioural modules and can be used as a regular OOP language. What is important in Core ABS is that interfaces define types and the methods available for that type. Classes can implement these interfaces and their methods but are not types themselves. Other ways to create types in ABS are to use algebraic data types using the `data` keyword or by defining type synonyms which are semantically equivalent to their synonym, using the `type`.

ABS is designed to model distributed systems and applies a concurrency model using concurrent object groups (COG) [12]. Each COG has its own heap of objects and communicates with other COGs through asynchronous method calls. Calls inside one COG are regular, local, synchronous method calls. Synchronous method calls are made using the normal “.” and can only be performed on `[Near]` references. These are references to objects which belong to the same COG. `[Near]` is an annotation which can be added to variables to indicate that they refer to an object residing in the same COG. Asynchronous method calls are made using a “!” and can only be performed on `[Far]` references. These are references to objects which belong to a different COG. `[Far]` is an annotation similar to `[Near]` only that `[Far]` indicates that the variable refers to an object residing in a different

COG. The result of an asynchronous call is called a future. The execution of a method can be suspended by performing an **await** on such a future. When execution resumes, a **get** can be performed on the future and the actual result is returned.

### B. Micro Textual Variability Language

$\mu$ TVL is the language used to describe feature models [1] [10] using a textual representation. It allows describing feature models as a forest of nested features with possibly multiple roots (for orthogonal variability) with the possibility to add boolean or integer attributes to each feature. Additional constraints can be put on the presence of features and on the possible values of attributes.

### C. Delta Modelling Language

Delta modules (or deltas in short), which are used to achieve variability in a SPL, are defined using DML. They contain program modifications, adding, removing, and refining classes, methods, and fields. The deltas are applied to the core module, specified using Core ABS.

When defining a delta, first an identifier and possibly some attributes are given. Secondly, the added, modified, or removed class, interface, method, or field is specified. When modifying or adding, the new implementation is given next. Modifications may sometimes want to call the original implementation in order to extend its behaviour. This can be done by calling the special method **original**().

### D. Product Line Configuration Language

To link feature models created using  $\mu$ TVL to delta modules created using DML, the product line configuration language is used. A product line configuration starts with the name of the SPL followed by the applicable features. Next the deltas to be applied are specified together with an optional **after** clause, specifying a partial order on the application of deltas, and a **when** clause, specifying for which features the delta should be applied.

### E. Product Selection Language

The different end products are specified using the product selection language. A product description consists of a product name, followed by the features the product has and, if necessary, feature attributes with assigned values.

### F. Full Specification of ABS

The previous sections briefly discussed the different language components of ABS. Many more concepts exist in these languages, but the interested reader is referred to [13], [4] and [3] for a full discussion of these languages.

### G. Code Reuse

Most object-oriented programming languages (e.g. Java) use inheritance as a mechanism to avoid code duplication. As explained in Section I, there is no support for inheritance in ABS. ABS provides deltas which can be used to achieve code reuse. The next three chapters will discuss three approaches to code reuse in ABS, with the last two depending on deltas.

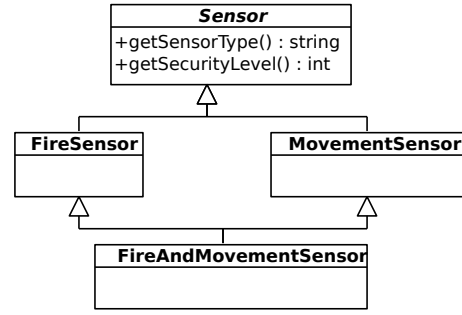


Fig. 1. Sensor type hierarchy

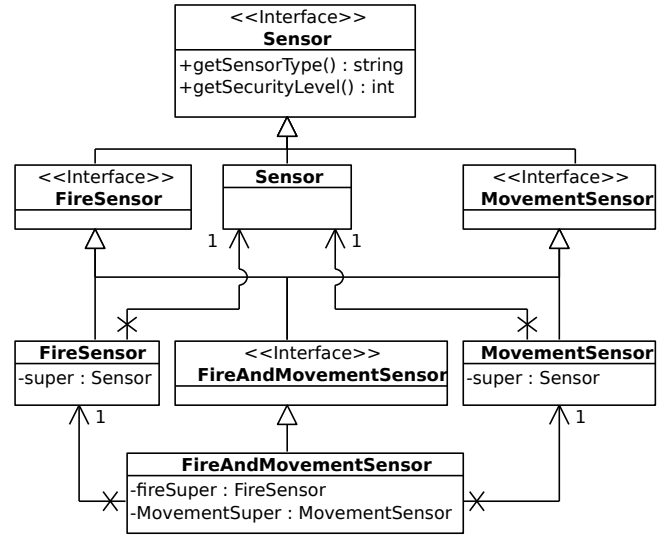


Fig. 2. Sensor class diagram for delegation approach

Each approach starts from an existing type hierarchy, like in Fig. 1.

## III. DELEGATION APPROACH

The delegation approach is based on the well known delegation pattern of software engineering, which can be defined as: “An implementation mechanism in which an object forwards or delegates a request to another object. The delegate carries out the request on behalf of the original object.” [6] Delegation can be used as an alternative to inheritance since the delegation of a method to another class can be viewed as a subclass delegating method calls to its superclass. The difference is that in delegation, the subclass has the superclass as a variable instead of it being an instance of the superclass. To avoid confusion, in the delegation approach a superclass will be called a *superobject* and a subclass a *subobject*. A type hierarchy like in Fig. 1 can be transformed to the class diagram in Fig. 2. A simple pattern describes this procedure:

- 1) For every type in the hierarchy: define an interface which extends its direct supertype’s interface.
- 2) For every interface, define a class which implements that interface.
  - a) Define a field of the supertype’s type.

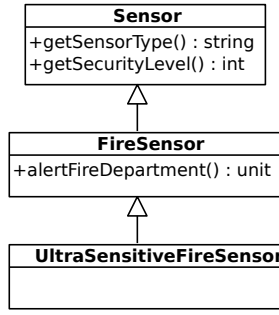


Fig. 3. Type hierarchy of fire sensors

- b) In the initialisation block, create an object of the supertype's class and assign it to the field defined in the previous step.
- c) Delegate all method calls that don not need overriding to the object created in the previous step.
- d) Define getters and setters for all variables that need to be available in lower type's implementations (and add these getters and setters to the type's interface).

The delegation approach provides a simple alternative to inheritance. It can simulate overriding, and even multiple inheritance. Because the delegation approach uses main concepts of object-oriented programming (OOP), it is applicable to most other OOP languages as well.

Three main problems exist with the delegation approach. (a) When a method call is performed on a subtype, but this method is delegated to a supertype and this implementation calls another method which is overridden in the subtype, the supertype's implementation will be used since it has no knowledge of this overriding. To overcome this problem, the method implementation from the superclass would have to be copied to the subtype resulting in code duplication. (b) The delegation approach also suffers from a coding overhead since every method of every supertype all the way to the top has to be implemented for every type in the hierarchy. Also, variables defined higher in the hierarchy need getters and setters to be accessible for implementations lower in the hierarchy, breaking a part of the encapsulation. (c) Memory is also a concern since multiple objects reside in memory to actually only represent one. If for instance a type hierarchy has ten level, an object of the lowest level has ten objects representing it, more if multiple inheritance is simulated.

#### IV. SINGLE TYPE APPROACH

As the name suggest, the single type approach is only applicable when one type of a type hierarchy to is needed in a program. Take the type hierarchy in Fig. 3. If a SPL would use this type hierarchy and only the *Sensor*, or the *FireSensor*, or the *UltraSensitiveFireSensor* is needed in the program because of selected features, the single type approach can be used. It is described in the following pattern:

- 1) Define an interface for the top-level type.

- 2) Define a class implementing the interface defined in the previous step.
- 3) For each subtype, define a delta which modifies the original class the suit the subtype's specific needs.
- 4) Apply the delta's in a top-down order until the delta of the needed type is reached.

The obvious downside of this approach is that only allowing one type of a type hierarchy is a very limiting constraint. However, because it uses deltas to change the behaviour of a class based on the required type, the overhead in coding (writing method headers for each type) and memory (only one object is instantiated here) is alleviated. These advantages over the delegation approach make it worth considering this approach when possible.

#### V. MULTIPLE DELTA APPROACH

When two or more types of a type hierarchy are needed in a program, which is the most frequent case, the single type approach is rendered useless.

The multiple delta approach is a sort of extension to the single type approach. It does allow multiple types of a type hierarchy to exist in one program and uses the type system of ABS to accomplish this. The perceived type of an object is dependent on the variable that contains the object, not on the class of the object. So if a class implements the interfaces *Observable* and *Alarm*, an instantiation of this class can be assigned to a variable of the type *Observable* or to a variable of the type *Alarm*. Only calls to methods in the interface belonging to the type are accepted though. Listing 2 illustrates this. Lines 23 and 26 are invalid as they call methods that are not available for the variable's type.

```

1  interface Observable {
2      Unit register(Observer observer);
3      Unit unregister(Observer observer);
4  }
5  interface Alarm {
6      Unit setAlarm();
7  }
8  class Alarm implements Observer, Alarm {
9      Unit register(Observer observer) {
10         ...
11     }
12     Unit unregister(Observer observer) {
13         ...
14     }
15     Unit setAlarm() {
16         ...
17     }
18 }
19 {
20     Observer o = new Observer();
21     Observable ob = new Alarm();
22     ob.register(o);
23     ob.setAlarm(); // Error!
24     Alarm al = new Alarm();
25     a.setAlarm();
26     a.register(o); // Error!
27 }
  
```

Listing 1. Example of types in ABS

So if the type hierarchy in Fig. 4 is used, a variable of type *Observer* will only accept calls to *register(Observer)* and

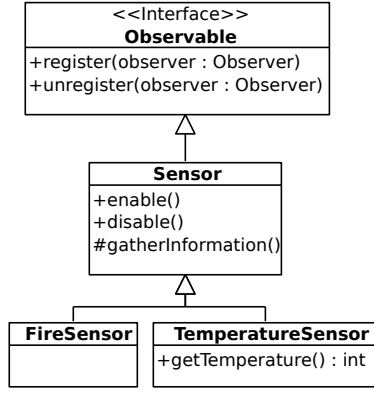


Fig. 4. Type hierarchy of Sensor, TemperatureSensor and FireSensor

`unregister(Observer)`, a variable of type *Sensor* will accept calls to the two methods of *Observable* (as it extends this interface), as well as to `enable()` and `disable()`.

From this observation, the following pattern was developed:

- 1) Define an interface for the top-level type.
- 2) Define a class implementing the interface defined in the previous step.
- 3) For each subtype define a delta which:
  - a) defines an interface for the type extending the direct supertype's interface,
  - b) adds the newly defined interface to the class defined in the third step,
  - c) modifies the class defined in the third step by adding the methods defined in the type's interface.

Because all method implementations for a type hierarchy reside in the same class, the multiple delta approach has a problem when two types require a different implementation for the same method. To alleviate this problem a type parameter is introduced in the class construction, that states of which type in the hierarchy the object is. When overriding a method in a delta, the following template should then be used:

```

1  ReturnType result;
2  if (this.typeE == <Type>) {
3    // <Type specific implementation here>
4  } else {
5    result = original();
6  }
7  return result;

```

Listing 2. Example of types in ABS

This executes the type specific implementation if the type of the object matches, otherwise it executes the implementation of the supertype by calling `original()`.

The main advantage of the multiple delta approach over the delegation approach is the correct use of overridden/overriding methods. This was a major issue in the delegation approach. A second important advantage over the delegation approach is the reduction of code overhead. Since the multiple delta approach uses the same class for all the different types, explicitly implementing the methods defined in the interfaces is only necessary when the behaviour of the supertype needs

to be overridden. On the other hand, always modifying the same class is the biggest problem of this approach, since very instantiation contains the functionality of all the types in the type hierarchy. This is mainly a problem when different subtypes have a different behaviour for the same method. This can however be solved by using a combination of if-statements and calls to `original()`. Real overriding like in Java is not possible in this approach due to the same class factor. All method implementations for the different types reside in the same method so changing the method signature is not possible.

## VI. RELATED WORK

This paper discusses three approaches to code reuse, two of which are applicable to ABS (and possibly other DOP languages), and one which is more generally usable in OOP. Other constructs for code reuse have been researched as well, for instance Findler and Flatt [5] discuss *units* and *mixins* as a means for solving complex reuse problems in a natural manner; Biemand and Xia [2] provide a quantitative study of the use of inheritance in C++.

Helvensteijn et al. [9] evaluate the use of the delta modelling methodology as a means to accurately model and implement SPLs. It reports on the implementation of an industrial scale product in ABS for which the Delta Modelling Workflow (DMW) [8] was used. This paper reports on the practical use of the DMW and therefore also of deltas for implementing a full SPL whereas this paper focuses on using deltas for code reuse which only implicates a subset of the implementation.

## VII. CONCLUSION

In this paper, three approaches for code reuse in ABS were discussed. The delegation approach tries to mimic inheritance by delegating method calls to a superobject which represents its superclass. The biggest problem of this approach is that the superobjects have no knowledge of their subobjects and therefore of possible overriding of their methods. This may result in unexpected results since the programmer will expect the overriding method to be used instead of the overridden one. Also, a coding overhead is present due to the delegation. Every non-overridden method still has to be implemented in every class, calling the same method in its superobject. A memory problem exists as well since an object lower in the type hierarchy is represented by one object for every level in the hierarchy. One positive point about the delegation approach is that it is usable in most object-oriented programming languages since it uses main concepts of object-oriented programming (classes and interfaces).

The single type approach is a first step into solving the problems of the delegation approach. In the single type approach, one class is defined and deltas modify the functionality of this class. This solves the delegation and its associated coding overhead. Its simplicity makes it very easy to use, but also makes it useless in many cases. It can only be used when one type in the type hierarchy is needed throughout the entire program, a rare condition.

The third, or multiple delta approach solves the problem of the single type approach by exploiting the ABS typing system. Interface are again defined for every type and are added to the original class implementation in deltas. These deltas also modify the functionality of the class by adding new methods and overriding other methods using a specific pattern in order to handle multiple method implementations. Variables containing objects of this class have a specific type and only the methods defined in the interface of this type are available. The downside is that this pattern needs typing information which the programmer needs to fill in manually on creation of the object and methods always have the same unalterable signature.

#### ACKNOWLEDGEMENT

The author would like to thank José Proença and Radu Muschevici for reviewing and proofreading initial drafts and their help and insightful views on the research performed.

#### REFERENCES

- [1] Don Batory, David Benavides, and Antonio Ruiz-Cortes. Automated analysis of feature models: challenges ahead. *Commun. ACM*, 49(12):45–47, December 2006.
- [2] James M. Bieman and Josephine Xia Zhao. Reuse through inheritance: a quantitative study of C++ software. *SIGSOFT Softw. Eng. Notes*, 20(SI):47–52, August 1995.
- [3] Dave Clarke, Stijn de Gouw, Reiner Hänle, Einar Broch Johnson, Ilham W. Kurnia, and Radu Muschevici. Deliverable D1.2 Full ABS modelling framework, March 2011.
- [4] Dave Clarke, Nikolay Diakov, Reiner Hähnle, Einar Johnsen, Ina Schaefer, Jan Schäfer, Rudolf Schlatte, and Peter Wong. Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In Marco Bernardo and Valérie Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 417–457. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-21455-4\_13.
- [5] Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. *SIGPLAN Not.*, 34(1):94–104, September 1998.
- [6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, September 2007.
- [7] Reiner Hähnle, Einar Broch Johnsen, Bjarte M. Østvold, Jan Schäfer, Martin Steffen, and Arild B. Torjusen. Deliverable D1.1A report on the core ABS language and methodology: Part A, April 2010.
- [8] Michiel Helvensteijn. Delta modeling workflow. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 129–137, New York, NY, USA, 2012. ACM.
- [9] Michiel Helvensteijn, Radu Muschevici, and Peter Y. H. Wong. Delta modeling in practice: a Fredhopper case study. In *VaMoS*, pages 139–148, 2012.
- [10] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.
- [11] Ina Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, 2010.
- [12] Jan Schäfer and Arnd Poetzsch-Heffter. Jacobox: generalizing active objects to concurrent components. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 275–299, Berlin, Heidelberg, 2010. Springer-Verlag.
- [13] Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte. The ABS tool suite: Modelling, executing and analysing distributed adaptable object-oriented systems. Submitted for publication, September 2011.



## Fiche masterproef

*Student:* Wouter Seyen

*Titel:* Evaluation of Delta Modelling in the ABS Language

*Nederlandse titel:* Evaluatie van delta modellering in de ABS taal

*UDC:* 681.3

*Korte inhoud:*

This thesis evaluates the ABS Modelling Framework which supports delta-modelling. A short introduction to the ABS language is given and the smart home product line on which a part of the evaluation is based is discussed. This thesis presents three approaches to achieve code reuse: the delegation approach, the single type approach and the multiple delta approach. The first one is applicable to most object-oriented programming languages as it does not rely on deltas, the second one relies on deltas but is only applicable when one type of a type hierarchy is needed in a product, the last one also relies on deltas and is the most usable approach to code reuse of the three. ABS also shows to be fairly intuitive to use. It still has to mature a bit, as some annoyances in the language show. These can be improved in the future though. Also, according to a comparative performance test, ABS is about three orders of magnitude slower than Java, which is due to performance being a low priority requirement when ABS was developed.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen

*Promotor:* Prof. dr. D. Clarke

*Assessoren:* Prof. dr. M. Denecker  
Dr. E. Truyen

*Begeleiders:* R. Muschevici  
Dr. J. Proença