Design, Verification and Implementation of Exception Control Flows for Product Line Architectures

Patrick H. S. Brito *Nelio Cacho*
Alessandro Garcia *Cecília M. F. Rubira*
Rogério de Lemos

Technical Report - IC-09-11 - Relatório Técnico

March - 2009 - Março

Design, Verification and Implementation of Exception Control Flows for Product Line Architectures

Patrick H. S. Brito[1] Nelio Cacho[2] Alessandro Garcia[2]
Cecília M. F. Rubira[1]
Rogério de Lemos[3]

[1] University of Campinas (UNICAMP), Brazil
{pbrito, cmrubira}@ic.unicamp.br

[2] Lancaster University, UK
{cacho, garcia}@lancaster.ac.uk

[3] University of Coimbra, Portugal
rdelemos@dei.uc.pt

Abstract

Separation of concerns is one of the overarching goals of exception handling in order to keep separate normal and exceptional behaviour of a software system. In the context of software product lines, this separation of concerns is important for designing software variability related to different exception handling strategies. This technical report presents a tool-supported solution for designing, verifying and implementing exceptional behaviour variability into product lines architectures. Our solution is based on: (i) the adoption of an exception handling model that supports explicit exception control flows and pluggable handlers; (ii) a strategy for designing and automatically verifying the selection of variation points related to exception control flows and handlers; and (iii) an aspect-oriented implementation for exceptional behaviour variability. We evaluate qualitatively and quantitatively our solution through a case study targeting a real mobile application.

1 Introduction

Fault tolerance is the ability of a system to continue delivering its service despite the presence of faults [3]. The provision of fault tolerance relies on the existence of redundancies, which can be implicitly incorporated through the usage of exception handling for supporting error detection and handling. The importance of exception handling is attested by the fact that many mainstream programming languages, such as, Java, Ada, C++ and C#, implement exception handling mechanisms. These languages provide constructs to indicate the occurrence of an error (*raise* or *throw* an exception), and means to incorporate recovery actions (*handle* the exception), including error handling. However, during system design special care has to be taken to avoid entangling normal and exceptional specifications, which can increase system complexity, and compromise system analysis and reuse. Separation of concerns is one of the overarching goals of exception handling in order to keep separate normal and exceptional behaviour of a software system. This separation promotes both adaptability and reuse of normal and error handling code, and it can also lead to an error handling strategy that supports plugging and unplugging of handlers. Moreover, it is also desirable to consider the exceptional control flow from a global perspective in order to promote the rationale of the system's exceptional behaviour. In the context of software product lines (SPL), this separation of concerns between normal and exceptional behaviour is particularly adequate for designing software variability related to different exception handling strategies to perform error recovery. The exceptional behaviour variability should consider both the selection of different exception control flows and different exception handlers according to the resources available for each product. This technical report presents a tool-supported solution for incorporating exception handling into the development of fault-tolerant software systems.

One of the main artefacts in the contexts of a SPL is the product line architecture (PLA), which explicitly represents the commonalities and variabilities of architectural elements and their configurations. While abstracting away from system details, architectures provide a global system perspective that is key for the explicit representation of exceptional control flows, and structural separation between the representation of normal and exceptional behaviour. In a PLA, the commonalities of architectural elements and their configurations are reused in different products, while the variabilities are resolved through design decisions related to the choices captured by the PLA. The contribution of this technical report is the provision of tool support for the architectural design and implementation of fault tolerant software, which employs a novel exception handling model. This new model, explicitly represents exception control flows, and promotes the reuse of the exception handlers of the software. The tool support integrates the architectural design of the software system with the verification process of the exception control flows and handlers. As part of the tool, the variability related to the exceptional behaviour is implemented using an aspect-oriented solution.

The rest of this technical report is organised as follows. Section 2 discusses the liabilities of conventional models for exception handling. Section 3 presents the exception handling model adopted as part of our solution. Section 4 describes the target system that will be used throughout the technical report for exemplifying and evaluating our contribution. Section 5 describes the tool-support for incorporating exceptional behaviour during the development of software systems. Details about the formal representation and verification of exception flows in software architectures are presented in Section 6. Section 7 presents the implementation of our exception handling mechanism using an aspect-oriented solution. In Section 8, the proposed exception handling is evaluated through a quantitative and qualitative comparison between two products derived from the same SPL, but with different exceptional behaviours. Section 9 describes some related work. Finally, Section 10 provides some concluding remarks, and identifies future directions of research.

2 Problem Definition

This section discusses the liabilities of exception handling mechanisms in terms of their limited separation of concerns between the normal and exceptional behaviour (Section 2.1), and implicit exceptional control flows (Section 2.2). We argue that in the context of SPLs it is even more necessary to overcome these limitations.

2.1 Limited Separation of Concerns

Exception handling contexts (EHCs) are regions in a program where the same exceptions are always treated in the same way. For this, EHCs can have a set of associated handlers, which are activated when exceptions are raised within the context. In Java, `try` blocks define EHCs, and `catch` blocks define handlers.

Separation of concerns is one of the overarching goals of exception handling, ans is specially important in the context of software product lines (SPL). In fact, one of the main motivating factors for the appearance of exception handling mechanisms was to separate the normal and exceptional behaviuor [24, 26]. Nonetheless, the kind of separation promoted by the exception handling mechanisms of most mainstream programming languages brings only limited advantages [9, 12, 20]. In particular, it hinders in explicitly separating the implementation of the normal code and its respective exception handlers.

Since fault tolerance activities are intrinsically an application-specific issue, it is extremely difficult to reuse exception handlers that implement activities of system recovery. The high coupling existing between the normal and exceptional behaviour makes it difficult the construction of reliable SPLs, since it does not facilitate the changing of handlers according to the necessity of specific products. It is necessary to have a "low-coupling" exception handling approach that makes it possible the specification of pluggable exception handlers since the system architectural design, where structural properties can be verified, until the system implementation, in order to promote the reuse of source code for the normal behaviour.

2.2 Implicit Exception Control Flows

In our view, the most serious problems with exception handling stem from the fact that it is a global design issue [28]. Existing exception handling mechanisms do not appropriately take this into account [16]. They are based on the implicit assumption that it is enough to specify the places where exceptions are raised and the places where they are handled. The main consequence of this limitation is that exceptions introduce implicit control flow [7, 22]. If something changes between the raising and handling of exceptions, the control flow in apparently unrelated parts of the system may change in surprising ways [28]. This creates two direct complications: (i) it becomes difficult to discover where the exceptions raised within a given context will be handled; (ii) it is also difficult to trace a handled exception to the place where it was originally raised. In other words, traditional exception handling mechanisms provide constructs for raising and handling exceptions [15]. However, not much support is provided to the task of understanding the paths the exceptions take from the raising site to the handling site [22, 27, 28].

Some strategies, such as the implemented by Java, try to alleviate these problems by supporting the definition of an explicit *exception interface*. They require the specification of a list of unhandled exceptions that each operation signals to its clients, otherwise, an error is detected. The main problem with this approach is that it hinders software maintenance [31]. If a new exception is added to the exception interface of a single operation at the bottom of the call chain, the exception interfaces of all operations through which the new exception will be propagated also have to be updated. For systems with long operation call chains, this is a time-consuming and error-prone task [31, 28], causing severe problems on the system architecture evolution [23]. Moreover, in the architectural point of view, this local specification of exception propagation hinders the traceability of the exception, which can be mapped to a different type during the propagation.

The bottom line is that, in general, outside of the raising and handling scopes, exceptions work as an undesirable (and difficult to track) side-effect to the normal behaviour. Many of the problems reported in the literature pertaining to exception handling are related to the inability of properly modularizing it, so that the interface between exceptions/exception handling and the normal code is well-documented. Approaches such as exception interface alleviate this problem, but only to a limited degree. Besides, they create new problems of their own. To the best of our knowledge, there are no general approaches to modularize error handling code, so that it can be understood and maintained separately from the normal code.

In the context of software product lines (SPL), the explicit separation between the normal code and the exception handlers is even more necessary. This low-coupling is necessary for specifying the variabilities between the different products of the same SPL. Depending on the design decisions involved in each product, different strategies for exception handling can be implemented. This “exceptional variability” depends on the resources available in each product, and is related to both the selection of proper handlers, and the existence of different exception control flows.

3 An Exception Handling Model for PLA

As discussed earlier, it is necessary to adopt an approach to (i) separate normal code from exception handling code, and (ii) analyse exception flows from an end-to-end perspective. In this context, an exception handling model is the core of our approach since it defines the interaction between the sites raise and handle exceptions. The model presented in this section extends the Java model by introducing two new concepts: *explicit exception channels* and *pluggable handlers*. We describe these concepts in the following two subsections. Then, in Sections 6 and 7, we present the proposed model used at the architectural and implementation level, respectively.

3.1 Explicit Exception Channels

An *explicit exception channel* (*channel*, for short) is an abstract duct through which exceptions flow from a raising site to a handling site. More precisely, an explicit exception channel (*EEC*) is a 5-tuple consisting of: (i) a set of exception types *E*; (ii) a set of raising sites *RS*; (iii) a set of handling sites *HS*; (iv) a set of intermediate sites *IS*; and (v) a function *EI* that specifies the channel's exception interface. *Exception types*, as the name indicates, are types that, at runtime, are instantiated to exceptions that flow through the channel. For simplicity, we use the term "exception" to refer to both exceptions (runtime elements) and exception type (compile-time element). When necessary, we make the distinction explicit. The *raising sites* are loci of computation where exceptions from E can be raised. The actual erroneous condition that must be detected to raise an exception depends on the semantics of the application and on the assumed failure model. For reasoning about exception flow, the fault that caused an exception to be raised is not important, just the fact that the exception was raised. The *handling sites* of an explicit exception channel are loci of computation where exceptions from E are handled. Even when the handler raises exceptions, it is not considered a raising site. In languages such as Java, both raising and handling sites are methods.

If an explicit exception channel has no associated handlers for one or more of the exceptions that flow through it, it is necessary to define its *exception interface*. The latter is a statically verifiable list of exceptions that a channel signals to its enclosing context, similarly to Java's throws clause. In our model, the exception interface is defined as a function ($Ex1 \alpha Ex2$) that translates exceptions flowing ($Ex1$) through the channel to exceptions signalled ($Ex2$) to the enclosing exception handling context (EHC).

Raising and handling sites are the two ends of an explicit exception channel. Handling sites can be potentially any node in the method call graph that results from concatenating all maximal chains of method invocations starting in elements from *HS* and ending in elements from *RS*. To keep the model simple, we do not consider a handling site that throws exceptions a raising site. All the nodes in such graph that are neither handling nor raising sites are considered intermediate sites. *Intermediate sites* comprise the loci of computation through which an exception passes from the raising site on its way to the handling site. Intermediate sites in Java are methods that indicate in their interfaces the exceptions that they throw, i.e., exceptions are just propagated through them, without side effects to program behaviour. Note that the notions of handling, raising, and intermediate site are purely conceptual and depend on the specification of the explicit exception channel.

3.2 Pluggable Handlers

A *pluggable handler* is an exception handler that can be associated to arbitrary EHCs, thus separating error handling code from normal code. A single pluggable handler can be associated, for example, to an operation in a component C1, two different operations in another component, C2, and all operations in a third component C3. In this sense, they are an improvement over traditional notions of exception handler. For example, the exception handling model of Guide [17] allows one to bind handlers to blocks, methods, classes, and exceptions, but not all of them at the same time! Another difference is that a pluggable handler exists independently of the EHCs to which it is associated. Therefore, these handlers can be reused both within an application and across different applications.

4 Description of the Target System

In the following, in order to exemplify and evaluate our contribution, we present as a case study a real software application, called MobileMedia [14]. MobileMedia is a SPL of a mobile application that manipulates photo, music, and video on mobile devices, such as mobile phones. The application uses various technologies based on the Java ME platform, such as SMS, WMA and MMAPI.

We believe that this application is representative of how exception handling is typically used to deal with errors in real software development efforts for two reasons. First, MobileMedia encompasses a large number of exception handlers that implement diverse exception handling strategies ranging from trivial to sophisticated. Second, they present heterogeneous crosscutting relationships involving the normal code, the handler code, the clean-up actions, and other crosscutting concerns.

Figure 1 presents a simplified view of the feature model of MobileMedia, following the representation proposed by Ferber et al. [13]. The core features of the MobileMedia are: Create/Delete Media, Media (photo, music or video), Label Media, and View/Play Media. The multiple features are just the types of media supported: Photo, Music, and/or Video. Finally, the optional features are: send photo via SMS (SMS for short), Copy Media, and set favourite media (Favourites). The core features of MobileMedia are applicable to all the mobile phone devices that are J2ME enabled. The optional and alternative features are configurable on selected mobile phones depending on the API support they provided. MobileMedia was developed for a family of four brands of devices, namely Nokia, Motorola, Siemens, and RIM.

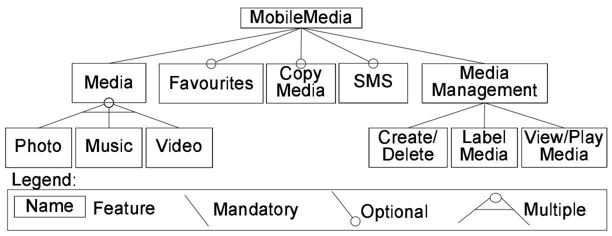

Figure 1: Simplified feature model of the MobileMedia SPL

4.1 Specification of the PLA

The software architecture of the MobileMedia SPL is mainly determined by the use of the Model-View-Controller (MVC) architectural pattern [5]. The exception handling code for the Java implementation followed the design approach described in detail elsewhere [28]. Figure 2 presents a representative partial view of software architecture. The three grey boxes encompass components that realise each of the three roles of the MVC pattern, namely model, view, and controller. Figure 2 also relates the architectural elements with the features in the feature model (Figure 1). This is done by the circles on the left top of the architectural elements. For instance, the SMS on the top of the SMS Controller (Figure 2) indicates that this element contributes to the implementation of the feature SMS in the feature model (Figure 1).

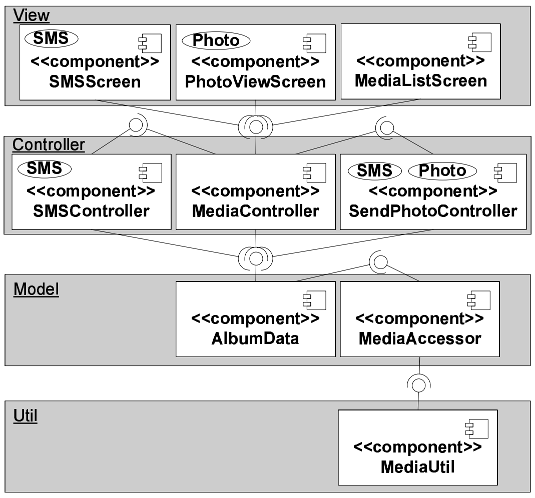

Figure 2: Product Line Architecture of the MobileMedia SPL

4.2 Exception Flows at the PLA

For better illustrating the exceptional behaviour of MobileMedia, Figure 3 depicts four of its 26 exception flows. The figure presents five components: MediaUtil, MediaAccessor, AlbumData, MediaController, and PhotoViewController. Each component is divided in terms of its normal behaviour (white part) and its error handling code (grey part). The ellipses inside the components represent methods. A black arrow from `a` to `b` indicates that method `a` invoked method `b`. A dashed arrow in the opposite direction indicates that, during the execution of `b`, an exception can be raised and this exception will be signalled to `a`. As would be expected, each arrow also indicates that control flow is passed from one method to the other. We also explicitly indicate the types of exceptions that the components encounters and signals.

We can define several different explicit exception channels in terms of the elements of Figure 3. For example, let *EEC2* be the explicit exception channel defined by the tuple {{InvalidArrayFormat}, {mu.getImageInfoFromBytes()}, {ma.loadImageDataFromRMS()}, {ad.getImages(), mc.showImageList()}, {}}. Only exception *InvalidArrayFormat* is raised in this channel. Explicit exception channel *EEC2* has method `mu.getImageInfoFromBytes()` as its sole raising site and `ma.loadImageDataFromRMS()` as its only intermediate site. Methods `ad.getImages()` and `mc.showImageList()` are handling sites, since pluggable handlers *h2* and *h3* are bound to them, respectively. The latter catches exception *InvalidArrayFormat* and maps it to exception *UnavailablePhotoAlbum*, whereas the former catches exception *UnavailablePhotoAlbum* and stops its propagation. It is important to stress that method `ad.getImages()` is not an intermediate site because it is associated to a pluggable handler. *EEC2* does not define an exception interface, as it includes handlers for all of its excep-

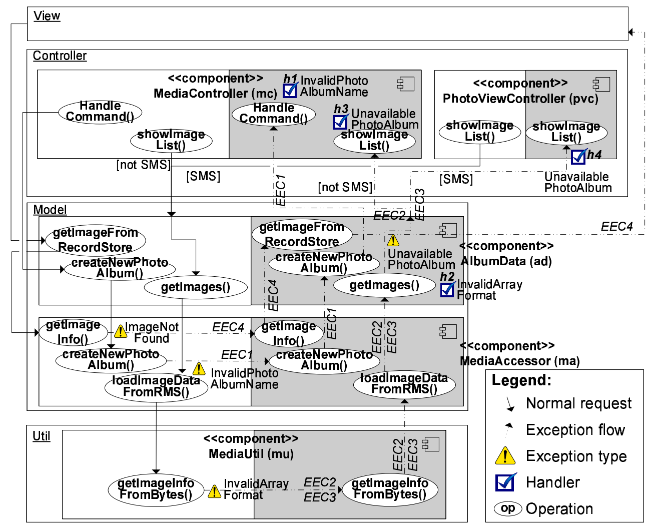

Figure 3: Control flow among components

tions. In total, the figure indicates three other explicit exception channels. It is important to notice that there is a variation point specified between *EEC2* and *EEC3*. It guarantees that a different exception flow (and handler) is activated when feature SMS is being used. Regarding *EEC4*, since it has no handling site, it should include *ImageNotFound* in its interface. It is defined by the tuple ({*ImageNotFound*}, {*ma.getImageInfo()*}, {}, {*ad.getImageFromRecordStore()*}, {*ImageNotFound* α *ImageNotFound*}).

Basically, this model provides the means to specify, in a local manner, non-local information pertaining to exception flows. For example, in Java, to implement explicit exception channel *EEC2*, it would be necessary to: (i) include *InvalidArrayFormat* in the exception interface of `mu.getImageInfoFromBytes()` and `ma.loadImageDataFromRMS()`; (ii) include *UnavailablePhotoAlbum* in the exception interface of `ad.getImages()`; and (iii) to implement `try-catch` blocks in methods `ad.getImages()` and `mc.showImageList()`. This is not a large amount of work, but the information about the exception that `mu.getImageInfoFromBytes()` signals and that `ad.getImages()` and `mc.showImageList()` handle is scattered throughout four different methods.

5 A Tool-Supported Architecture-Centred Development Process

In our approach, the software architecture is considered as first-level unit, guiding the development from the specification to the implementation of the application. An overview of the development method is shown in Figure 4. Activity 1 is responsible for the specification of the software architecture using a CASE[1] tool. After that, the graphical representation of the software architecture should be exported to the XMI format (Activity 2), which

[1]Acronym for Computer Aided Software Engineering.

is also supported by the CASE tool. Then, the XMI specification is used as input for automatically generating the formal specification of the software architecture (Activity 3). Basically, this activity consists on an automatic model transformation from UML (XMI files) to B-Method and CSP. The are two UML artefacts as input: a UML component diagram representing the software architecture, and a UML sequence diagram representing the interaction between architectural elements. Activity 4 is the formal verification of the software architecture, in order to identify and remove design faults related to the exception flow and the selection of features of the SPL. Activity 5 consists on the specification of the interactions amongst architectural elements, which is represented through a sequence graph. Afterwards, in Activity 6, this graph is used for generating integration and robustness test cases.

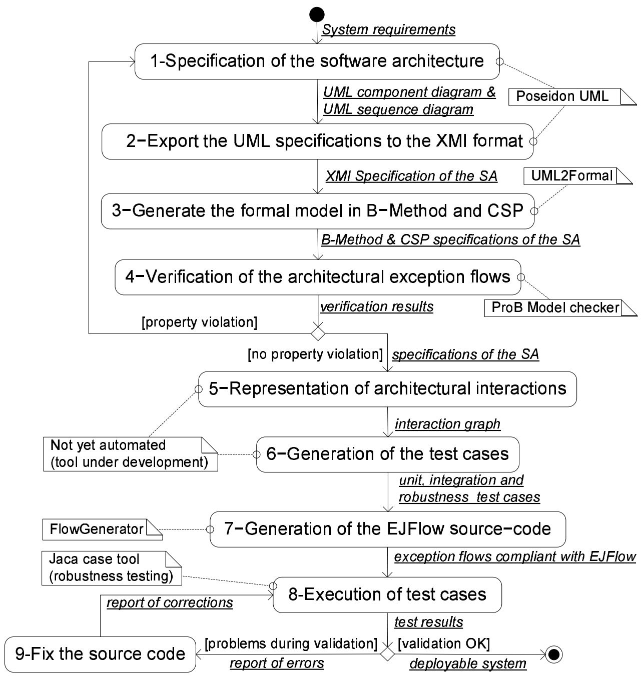
System requirements
1-Specification of the software architecture
Poseidon UML
UML component diagram &
UML sequence diagram
2−Export the UML specifications to the XMI format
XMI Specification of the SA
UML2Formal
3−Generate the formal model in B−Method and CSP
B-Method & CSP specifications of the SA
4−Verification of the architectural exception flows
verification results
ProB Model checker
[property violation]
[no property violation]
specifications of the SA
5−Representation of architectural interactions
interaction graph
Not yet automated
(tool under development)
6−Generation of the test cases
unit, integration and
robustness test cases
FlowGenerator
7−Generation of the EJFlow source−code
exception flows compliant with EJFlow
Jaca case tool
(robustness testing)
report of corrections
8-Execution of test cases
test results
9-Fix the source code
[problems during validation]
report of errors
[validation OK]
deployable system

Figure 4: A Tool-Supported Process for Modelling Exception Control Flows

With the system properly verified and the test cases already generated, Activity 7 consists on the source code generation of the exception flow according to the notation presented in Section 7. In Activity 8, the source code should be validated against its specification through the execution of the previously generated test cases. Finally, if any faults are identified during the testing, they have to be fixed in Activity 9.

6 Formal Verification of Exception Control Flows

6.1 Formal Representation of the PLA

For the representation of the PLA, shown in Figure 2, the B-Method is used for specifying the structure of the PLA, while CSP is employed for specifying its behaviour. It is important to know that the formal specification is automatically generated from XMI files representing the software architecture (see Section 5). The adoption of B-Method and CSP was mainly motivated by the explicit separation between the structural and behavioural specifications. We claim that this specification facilitates the comprehension of the formal model, as well as its adjustment for running extra verification. Moreover, the existence of a compliant model checker tool (ProB [19]) was also considered. For length constraints, the details about the formal models are not presented in this technical report.

Regarding the structural specification, we have defined a hierarchy of B-Method machines, shown in Figure 5, that explicitly separates the specification of the feature model and the specification of the PLA. Following ProB notation, the rectangles represent B-Method machines, and the arrows represent relationships between them. The featureTypes and plaTypes machines contain sets that store the data types necessary for representing the feature model and the PLA, respectively. The featureModel machine `uses` the data of featureTypes to represent the feature model presented in Figure 1 in terms of its variants, and the relations among features. The pla machine `uses` the data of plaTypes, and the information of the feature model (featureModel) to structure the PLA presented in Figure 2 in terms of its architectural configuration, explicit exception channels, and the respective variation points. The `uses` relationship means that the featureModel and pla machines can refer, respectively, to the shared sets of featureTypes and plaTypes for defining its invariants.

It is important to stress that the channels are explicitly represented following the exception model presented in Section 3. For this, the pla B-Method machine defines relations for representing each element of the channels' five-tuple: raised exceptions, raising sites, handling sites, intermediate sites, and functions of explicit propagation. Moreover, the formal model also represents the exception mappings during the exception flow, which is important to provide traceability during the model checking process, in order to identify architectural mismatches.

For representing the variabilities associated to the exceptional behaviour, the formal model also associates the exception channels and handlers to decisions of the feature model. For example, in the case of *EEC2* and *EEC3* channels presented in Figure 3, the formal model of the MobileMedia case study represents both channels. But when the SMS feature is not selected, EEC2 is considered active, and EEC3 inactive, and the opposite occurs when SMS feature is selected. The same occurs with architectural elements, the respective interfaces, and architectural configurations.

The behavioural specification of the PLA is obtained by restricting the pla B-Method machine with a CSP specification associated to it. This CSP specification defines the interactive behaviour between the architectural elements, including requests, responses and error signalling at the software architecture.

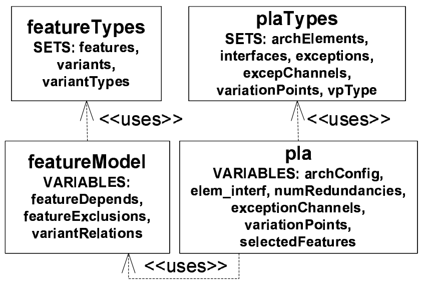
featureTypes
SETS: features, variants, variantTypes
plaTypes
SETS: archElements, interfaces, exceptions, excepChannels, variationPoints, vpType
<<uses>>
<<uses>>
featureModel
VARIABLES: featureDepends, featureExclusions, variantRelations
pla
VARIABLES: archConfig, elem_interf, numRedundancies, exceptionChannels, variationPoints, selectedFeatures
<<uses>>

Figure 5: Hierarchy of B-Method Machines

6.2 Properties of Interest

We have defined three complementary categories of properties to be verified: (i) decision checking that checks the consistency between the variants of the feature model and the variation points of the PLA, according to the respective decision model; (ii) invalid channels checking that checks the consistency between the exceptional flows specified in the PLA and the structural restrictions of the software architecture; and (iii) exceptional architectural mismatches checking that checks the incompatibility between the exceptions propagated from provided interfaces and the respective exceptions expected in required interfaces.

The *decision checking* consists of two complementary checks: (i) feature consistency, which verifies the integrity of the selected features when compared with the rules defined in the feature model (Figure 1); and (ii) structural consistency, which verifies the integrity of the software architecture concerning the selected features. An example of a violation of the feature consistency would be the non-selection of the MediaManagement feature, which is mandatory according to the feature model (Figure 1). An example of a violation of the structural consistency would be the existence of the SMSController component at the software architecture when the feature SMS is not supposed to be selected. We have defined a total of six consistency properties, which are verified for the featureModel B-Method machine. The properties of structural consistency is specific for each PLA, and is verified for the pla B-Method machine. In the context of the MobileMedia case study, we have specified a total of 24 properties, one for each possible decision of the product line architecture. These decisions concerns the possible selection scenarios of alternative and optional features (see Figure 1).

The objective of *invalid channels checking* is to assess if all the exceptional flows can actually occur in the PLA. This checking is conducted in two complementary perspectives: (i) verification of the architectural configuration, in order to identify architectural elements that, although should be part of a flow, it cannot be as a consequence of a missing dependency in the architectural configuration; and (ii) verification of the exceptional masking, in order to identify some architectural elements that, although should be part of a flow, it cannot be as a consequence of exception masking before it should happen (impossible propagation). We have specified four properties for verifying invalid channels.

The *exceptional architectural mismatches checking* focus on the exception propagation between architectural elements, and intends to be sure that all the exceptions propagated from an element to another is proper interpreted. In this way, the propagated exception should either be the same, or an explicit conversion should be specified. Since the behavioural specification states that exceptions should be propagated until the end of the flow, in case of exceptional architectural mismatches, the flow is obliged to stop in an unpredictable way. The model checker detects these mismatches as deadlocks.

7 Exception Flows: From Architecture to Implementation

This section presents EJFlow [6], an AspectJ extension that implements the proposed exception handling model. EJFlow provides means for developers to define explicit exception channels and pluggable handlers in terms of the abstractions supported by AspectJ, namely, pointcuts, advice, and inter-type declarations [18]. More specifically, we define a new pointcut designator, a new kind of advice, and a new inter-type declaration. The pointcut (Section 7.1) defines explicit exception channels in terms of the exceptions that flow through them. The new advice (Section 7.2) implements pluggable handlers and, consequently, handling sites. The inter-type declaration (Section 7.3) allows one to specify the exception interfaces of explicit exception channels. Intermediate sites are computed automatically at compile time by the EJFlow compiler.

7.1 Defining Explicit Exception Channels

EJFlow provides a new pointcut designator, `echannel`, to support the definition of explicit exception channels. This pointcut designator takes a formal parameter, `et`, consisting of the name of an exception type. It matches any join point where the raised exception is a type of `et`. In the EJFlow pointcut language, named pointcut expressions built up using the `echannel` designator are considered explicit exception channels, as described in Section 3.1, where the name of the pointcut expression represents the name of the channel. As a first example, the following simple pointcut declares an initial version of channel *EEC1* (Figure 3):

```
pointcut EEC1():echannel(InvalidPhotoAlbumName)
```

Here, the pointcut matches any statements that may signal exception *InvalidPhotoAlbumName*. Furthermore, EJFlow performs a static analysis [29] on the program's method call graph, in order to identify the raising and intermediate sites of a given explicit exception channel. The implementation of `echannel` attempts to locate methods that raise the exception supplied as argument and considers them raising sites. A method can only be considered a raising site if the act of raising the exception is not a consequence of another exception, neither an implicitly propagated one nor an exception raised by a handler. The analysis then proceeds upwards, through the method call graph, considering every method to be part of the explicit exception channel, either as intermediate or handling sites. For the example of Figure 3, the method `ma.createNewPhotoAlbum()` is identified as raising site, and `ad.createNewPhotoAlbum()`, and `mc.handleCommand()` as either intermediate or handling sites. In summary, *EEC1* matches all calls through which exceptions that were raised as a result of the execution of method `ma.createNewPhotoAlbum()` flow, including calls to methods `ad.createNewPhotoAlbum()`, and `mc.handleCommand()`.

As mentioned in Section 3, exhaustive definition of exception types `et` can impair the usefulness of our approach. Hence, `echannel` supports AspectJ patterns to match exceptions related to a single class, a full class hierarchy, a class with a wildcard, or a combination of classes using logical operators. Therefore, rather than defining channels for exceptions `SocketException`, `FileNotFoundException`, `EOFException`, and so on, one can use just `echannel(IOException+)` to match all subtypes of `IOException`.

An explicit exception channel like the one defined above is too general to be useful. It is possible to specify more clearly-defined channels by explicit indicating the raising site of a channel. The code snippet below illustrates the definition of the explicit exception channel *EEC1* including its respective raising site:

```
pointcut rSite1 : withincode(public void createNewPhotoAlbum(..));
pointcut EEC1() : echannel(InvalidPhotoAlbumName, rSite1);
```

The second parameter of a channel definition identifies its raising site. The example above define the raising sites as separate pointcuts that the definition of *EEC1* use (Figure 3). Notice that the `echannel` designator only supports the specification of channels that have a single raising site. At the implementation level, we see explicit exception channels with multiple raising sites as compositions of simpler channels, each containing a sole raising site. EJFlow supports the definition of multi-raising site channels by means AspectJ's union operator (`||`).

In this manner, the specification of both simple and complex channels remains very simple, in accordance to AspectJ's semantics, and avoids too much syntax overload.

In some cases, a channel might fork at intermediate sites, resulting in two or more different propagation paths for the same exceptions. For example, suppose that there is an extra dashed arrow from `ad.getImages()` to `ad.createNewPhotoAlbum()` in Figure 3. If we wanted to define *EEC1* to be exactly the same as *EEC1* in Figure 3, it would be necessary to exclude this extra propagation "branch". In EJFlow, to be more specific about a channel, a developer can indicate some of its intermediate sites in its definition. In a similar vein, one can exclude some intermediate sites. In both cases, the semantics is to include or exclude the entire subtree of the channel whose root is the provided intermediate site. Intermediate sites (both included and excluded) are supplied as extra arguments to `echannel`. The following snippet presents a simple example:

```
pointcut rSite1 : withincode(public void createNewPhotoAlbum(..));
pointcut iSite1 : !withincode(public ImageData[] getImages(..));
pointcut EEC1() : echannel(InvalidPhotoAlbumName, rSite3, iSite1);
```

Pointcut *EEC1* above defines an explicit exception channel through which exception *InvalidPhotoAlbumName* flows, that has `rSite1` as its raising site, and that **does not** include the branch that starts in method `ad.getImages()` and continues to `mc.handleCommand()` (notice the "!" symbol in the definition of `iSite1`).

Explicit exception channels defined using only `echannel` are obviously incomplete, as they do not include handling sites nor an exception interface. If one compiles a program that defines such a channel, the EJFlow compiler will indicate a compilation error because there are exceptions that should be propagated to the enclosing EHC but are not part of the exception interface of the channel. Unlike Java, EJFlow verifies if exceptions flowing through an explicit exception channel are handled or declared in its exception interface regardless of the exception type, i.e., these rules apply to both checked and unchecked exceptions.

7.2 Plugging Handlers to Exception Channels

In order to specify the handling site of an explicit exception channel, EJFlow provides the `ehandler` advice. This advice is an implementation of pluggable handlers. It encapsulates the exception handling code that is executed when a certain point in an explicit exception channel is reached. Each `ehandler` advice consists of: (i) a set of parameters, like any other advice; (ii) a `boundto` clause specifying the explicit exception channel to which the handler is bound; (iii) an associated pointcut that determines the join point, within the channel, at which the advice executes; (iv) a `catching` clause that indicates the exception to be handled; and (v) a body, the actual handler implementation. To give an example of basic `ehandler` functionality, the following code snippet presents a useful handler (*h1*) in the context of channel *EEC1*:

```
void ehandler() boundto(EEC1()) catching(InvalidPhotoAlbumName e):
    withincode(public boolean handleCommand(..){ ... }
```

This advice handles exceptions flowing through explicit exception channel *EEC1*. Specifically, the handler is activated when such exceptions are caught within method `mc.handleCommand()`. A pluggable handler can be associated with multiple explicit exception channels by means of AspectJ's set union operator (`||`).

It is also possible to define pluggable handlers that are not associated with channels. In this case, they work as `after throwing` advice, with the difference that, like `around` advice, they can stop the propagation of an exception. The code snippet below shows a simple example:

```
void ehandler() catching(InvalidPhotoAlbumName e):
    withincode(public boolean handleCommand(..)){ ... }
```

7.3 Specifying Exception Interfaces

When a program cannot handle all the exceptions that flow through an explicit exception channel, it is necessary to declare these exceptions in the channel's exception interface. The `declare einterface` inter-type declaration serves this purpose. The following code snippet illustrates the definition of exception interfaces:

```
pointcut rSite4() : withincode(public ImageData getImageInfo(..));
pointcut EEC4(): echannel(ImageNotFound, rSite4);
declare einterface : ImageNotFound
   echannel EEC4(): execution(Image getImageFromRecordStore(..));

pointcut rSite4() : withincode(public ImageData getImageInfo(..));
pointcut EEC4(): echannel(ImageNotFound, rSite4);
declare einterface
    echannel EEC4(): execution(Image getImageFromRecordStore(..));
```

The first inter-type declaration (lines 3-4) explicitly indicates the exception to be declared in the exception interface of the channel. Alternatively, the second one (lines 8-9) specifies only the explicit exception channel to which the exception interface is associated. This second format is more general and states that every exception that flows through channel *EEC4* and is not handled is part of the channel's exception interface.

The `declare einterface` inter-type declaration avoids the need to specify the exception interface of each method that acts as a raising or intermediate site within a channel. Therefore, in the example of Figure 3, with the use of the two declarations above, exception *ImageNotFound* does not need to be declared in the `throws` clauses of methods `ma.getImageInfo()`, and `ad.getImageFromRecordStore()`. However, the compiler still performs the static checks. If one of these declarations is removed, the compiler issues an error message.

8 Case Study Evaluation

Overall, this case study has shown that using explicit exception channels and pluggable exception handlers it is possible to develop and evolve software product lines with variable exceptional behaviour. Thus providing the means to apply model checking for verifying key properties of the architecture, such as the feasibility of the channels specified, the existence of uncaught exceptions, and the explicit declaration of exception propagation. Due to space restrictions, we were not able to present in great detail the proposed solution and the many benefits it can offer. In the following, in order to complement the material presented so far, we summarise our experience when employing the solution for developing the MobileMedia system.

During the verification of the MobileMedia, the model checker has detected two deadlocks and the violation of two explicit exception channels. In both cases, the analyses of the counter-examples that were provided helped to identify the cause of the violation. The two deadlocks were caused by architectural mismatches in two channels: *EEC2* and *EEC3*. Since we forgot to specify in the model the conversion from *InvalidArrayFormat* to *UnavailablePhotoAlbum* at *ad.getImage()* (see Figure 3), those exceptions were considered incompatible during the propagation checking. Moreover, two channels from the Model layer to the View layer (see Figure 2) were considered impossible, for the software architecture didn't define dependencies between the architectural elements of these layers. The architecture has been fixed afterwards.

Regarding the existence of exceptions that are not caught by any handler (uncaught exceptions), the model checker has detected only one case, when we forgot to associate an exception handler in the View layer; but after fixing the association, no other violation has been found. In contrast, in a previous static analysis of an aspect-oriented implementation of the MobileMedia [11], approximately 67,5% of the exceptions were not caught by any handler. In part, we believe that the end-to-end perspective provided by the exception model reduces the chance of forgetting exceptions with no handler. As a whole, we consider that the proposed rigorous solution has helped to identify and correct design faults in earlier stages of the software development. Although most of these problems were simple to correct, if they were left to be corrected in the later phases of the development, it would have been much harder.

In order to assess the evolvability of the proposed solution towards the addition of new features, we have considered two different releases of the MobileMedia system [11]: (i) *release V4*, which permits the user to view photos and to make a catalogue of favourites; and (ii) *release V6*, which also permits the sending of photos through SMS. First, we have specified, verified, and implemented the PLA of release V4. Then, we evolved the PLA (and the respective formal model) for attempting release V6. For evolving the PLA from release V4 to release V6, first of all it was necessary to add new architectural elements (elements with a SMS ellipse on the left top, in Figure 2). In addition, the decision model of the PLA should also be updated in order to consider the possibility of selecting the SMS feature.

To evolve the formal model of the PLA (see Figure 5), it was necessary three punctual changes, both in the pla B-Method machine. First, pla was updated in order to reflect the new decision model and make it possible the selection of the SMS feature with no warning from the model checker. Second, the pla was updated in order to add new architectural elements, the respective interfaces, and the new explicit exception channels. Finally, it was also necessary to change the content of two relations of pla: (i) association between features and exception channels; and (ii) association between features and architectural elements. As presented in Section 6, these relations make it possible the activation and deactivation of explicit exception channels and architectural configurations, depending on the features that are being selected for a specific product.

Finally, regarding the exception model used in this technical report, it was considered essencial for the proposed solution, since it provides an end-to-end view of the exception propagation, and a clear separation of concerns between the normal and abnormal behaviour of the system [6]. Moreover, another characteristic that may help the evolution of products (different from the PLA evolution discussed so far) is that the separation of concerns remains at the implementation-level, what improves the modularity, high-cohesion and low-coupling of the system's source-code [6].

9 Related Work

In this section, we present work that is directly related to our own. For simplicity, we place related work in two categories: (i) exception flow analyses and verification; and (ii) formal verification of PLAs.

9.1 Exception Flow Analyses and Verification

Several contributions propose static analyses of source code for analysing exception flow [10, 30]. Usually, flow analysis consists on identifying propagation paths in a program to discover, for example, uncaught exceptions in languages with polymorphic types, such as Java.

Our approach leverages previous proposals for exception flow analysis, but differs in focus. The proposed approach targets the early phases of development and is broader in scope.

Several approaches have been proposed for promoting the automated analysis of software architectures. Wright [2] specifications can be translated to CSP and analysed for deadlock freedom and interface compatibility. Abowd and his coleagues [1] use Z to formalise and compare architectural styles. Ours is yet another work along this direction. It emphasises the specification of exception flow at the architectural level, and the satisfaction of some behavioural properties about exception handling.

Recent work by Castor et al. [8], in the Aereal framework, leverages existing languages and tools to support the description and analysis of exception flow in software architectures. That work is similar to ours in its focus, but it differs on the way exception flows are represented and on the goal of verification. While we represent a flow involving many architectural elements, the Aereal framework represents flows between two interconnected architectural elements. Moreover, besides the properties verified by Aereal (e.g., the existence of uncaught exceptions and useless handlers), we are also interested on verifying the viability or impossibility of each flow occurs.

9.2 Formal Verification of PLAs

Several contributions have proposed techniques for verifying software product lines. Most of the existing solutions focus on the verification of the feature model, intending to check the consistency of the choices used for instantiating specific products [4, 25]. These works are complementary to ours, since we focus on the representation of variability and verification of the exceptional behaviour of PLAs.

Lutz and Gannod [21] describe experiences with tool-assisted architectural analysis of a mission-critical software product line. The authors use model checking to determine the level of fault tolerance based on architectural scenarios. However, this approach only considers properties related to scenarios which are common to all products of a PLA, and does not provide a way to reuse the formal model in order to verify application-specific properties. Moreover, the verification solution that is proposed does not consider the exceptional behaviour of the software architecture.

Finally, to the best of our knowledge, there is no other approach for representing and verifying variability and correctness of the exceptional behaviour of product line architectures.

10 Conclusions and Future Work

This technical report has presented a development solution for supporting the design, verification and implementation of exception control flows in product line architectures. This solution allows the explicit representation of exceptional control flows at the software architecture, and the specification of pluggable exception handlers. We claim that these characteristics bring four main advantages to the software developer: (i) it makes exception flow understandable in a localised way, without the need to examine other parts of the program; (ii) it enhances program modularization by improving the error handling code reuse;

(iii) it promotes better maintainability of normal and error handling code by separating the handlers and eliminating annoying exception interface declarations; and (iv) it allows the specification of variability regarding exception flows and the respective error handlers for product line architectures. The verification approach combines the use of B-Method and CSP for checking properties of interests regarding both the correct selection of features, and the consistency between the exceptional flows specified in the product line architecture and its structural and behavioural restrictions. Finally, we have implemented the exception model with small syntactic additions to AspectJ. The feasibility of the proposed approach was demonstrated in the context of a mobile system. Our ongoing work encompasses the empirical evaluation of the error proneness on the use of the overall development approach when compared to conventional proposals discussed in this technical report. Furthermore, we intend to overcome a limitation of our solution regarding the explicit representation of exception variability at the implementation-level. For this, one possibility is the use of abstract aspects for representing alternative and optional sites of exception channels (raising sites, intermediate sites and handling sites), as well as alternative and optional handlers.

Acknowledgements

This work is supported in part by the European Commission grant IST-33710 - Aspect-Oriented, Model-Driven Product Line Engineering (AMPLE), grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), grant 479395/2004-7: Brazilian Council for Scientific and Technological Development (CNPq). Patrick is supported by Fapesp/Brazil, grant 06/02116-2 and CAPES/Brazil, grant 0722-07-3.

References

[1] G. D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Trans. Software Engineering and Methodology*, 4(4):319–364, 1995.

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Software Engineering and Methodology*, 6(3):213–249, 1997.

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1):11–33, January-March 2004.

[4] C. Blundell, K. Fisler, S. Krishnamurthi, and P. V. Hentenryck. Parameterized interfaces for open system verification of product lines. *Automated Software Engineering (ASE'04)*, 0:258–267, 2004.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: a system of patterns.* John Wiley & Sons, Inc., New York, NY, USA, 1996.

[6] N. Cacho, F. C. Filho, A. Garcia, and E. Figueiredo. Ejflow: Taming exceptional control flows in aspect-oriented programming (to appear). In *7th International Conference on Aspect-Oriented Software Development (AOSD'08)*, 2008.

[7] T. Cargill. Exception handling: a false sense of security. pages 423–431, 1996.

[8] F. Castor Filho, P. H. S. Brito, and C. M. F. Rubira. Specification of exception flow in software architectures. *Journal of Systems and Software*, 79(10):1397–1418, 2006.

[9] F. Castor Filho, N. Cacho, E. Figueiredo, R. M. ao, A. Garcia, and C. M. F. Rubira. Exceptions and aspects: the devil is in the details. In *Proc. of the 14th ACM SIGSOFT international symposium on Foundations of software engineering (SIGSOFT '06/FSE-14)*, pages 152–162, 2006.

[10] B.-M. Chang, J.-W. Jo, and S. H. Her. Visualization of exception propagation for java using static analysis. In *SCAM '02: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02)*, page 173, Washington, DC, USA, 2002. IEEE Computer Society.

[11] R. Coelho, A. Rashid, A. Garcia, F. Ferrari, N. Cacho, U. Kulesza, A. von Staa, and C. Lucena. Assessing the impact of aspects on exception flows: An exploratory study. In *ECOOP'08: Proceedings of the European Conference on Object-Oriented Programming*, page (to appear), July 2008.

[12] Q. Cui and J. Gannon. Data-oriented exception handling. *IEEE Transacrions on Software Engineering*, 18(5):393–401, 1992.

[13] S. Ferber, J. Haag, and J. Savolainen. Feature interaction and dependencies: Modeling features for reengineering a legacy product line. In *Proc. of the Second International Software Product Lines Conference (SPLC), LNCS 2379*, pages 37–60, 2002.

[14] E. Figueiredo et al. Evolving software product lines with aspects: An empirical study on design stability. In *Proc. of the 30rd international conference on Software engineering (ICSE'08)*, page (to appear), 2008.

[15] A. F. Garcia, C. M. F. Rubira, A. B. Romanovsky, and J. Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, 2001.

[16] P. Greenwood, T. T. Bartolomei, E. Figueiredo, M. Dósea, A. F. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, and A. Rashid. On the impact of aspectual decompositions on design stability: An empirical study. In *Proc. of 21st European Conference on Object-Oriented Programming (ECOOP'07), LNCS 4609*, pages 176–200, 2007.

[17] S. Lacourte. Exceptions in guide, an object-oriented language for distributed applications. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 268–287, London, UK, 1991. Springer-Verlag.

[18] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications Co., July 2003.

[19] M. Leuschel and M. J. Butler. Prob: A model checker for b. In *Proc. of Interfational Conference on Formal Methods (FME'2003), LNCS 2805*, pages 855–874. Pisa, Italy, 2004.

[20] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proc. of the 22nd international conference on Software engineering (ICSE'00)*, pages 418–427, 2000.

[21] R. R. Lutz and G. C. Gannod. Analysis of a software product line architecture: an experience report. *Journal of Systems and Software (JSS)*, 66(3):253–267, 2003.

[22] D. Malayeri and J. Aldrich. Practical exception specifications. pages 200–220. 2006.

[23] A. Molesini, A. F. Garcia, C. von Flach Garcia Chavez, and T. V. Batista. On the quantitative analysis of architecture stability in aspectual decompositions. In *Proc. of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pages 29–38, 2008.

[24] D. L. Parnas and H. Würges. Response to undesired events in software systems. pages 231–246, 2001.

[25] M. Poppleton. Towards feature-oriented specification and development with event-b. In P. Sawyer, B. Paech, and P. Heymans, editors, *13th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 07), LNCS 4542*, pages 367–381, 2007.

[26] B. Randell. The evolution of the recovery block concept. In Lyu, editor, *Software Fault Tolerance*, chapter 1, pages 1–21. 1995.

[27] D. Reimer and H. Srinivasan. Analyzing exception usage in large java applications. In *Proc. of ECOOP'2003 Workshop on Exception Handling in Object-Oriented Systems*, July 2003.

[28] M. P. Robillard and G. C. Murphy. Designing robust java programs with exceptions. *SIGSOFT Softw. Eng. Notes*, 25(6):2–10, 2000.

[29] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. on Software Engineering and Methodology*, 12(2):191–221, 2003.

[30] C. F. Schaefer and G. N. Bundy. Static analysis of exception handling in ada. *Softw. Pract. Exper.*, 23(10):1157–1174, 1993.

[31] M. van Dooren and E. Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. *Proc. of*

the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications, 40(10):455–471, 2005.