A Semantics for Execution Levels with Exceptions

Ismael Figueroa[*]
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
ifiguero@dcc.uchile.cl

Éric Tanter[†]
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
etanter@dcc.uchile.cl

ABSTRACT

Aspect-oriented languages are usually formulated as an extension to existing languages, without paying any special attention to the underlying exception handling mechanisms. Consequently, aspect exceptions and handlers are no different than base exceptions and handlers. Conflation between aspect and base exceptions and handlers may inadvertently trigger execution of unintended handlers, changing the expected program behavior: aspect exceptions are accidentally caught by base handlers or vice-versa. Programmers cannot state the desired interaction between aspect and base exceptions and handlers. Specific instances of this issue have been identified by others researchers. We distill the essence of the problem and designate it as the *exception conflation problem*. Consequently, we propose a semantics for an aspect-oriented language with execution levels and an exception handling mechanism that solves the exception conflation problem. By default, the language ensures there is no interaction between base and aspect exceptions and handlers, and provides level-shifting operators to flexibly specify interaction between them when required. We illustrate the benefits of our proposal with a representative set of examples.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features

General Terms: Languages, Design

Keywords: Exception handling, aspect-oriented programming, exception conflation, execution levels

1. INTRODUCTION

Aspect-oriented languages aim at modularizing crosscutting concerns. Well-known aspect-oriented languages, like AspectJ, define crosscutting behavior using pointcuts and advices: join points are interesting events raised during program execution, pointcuts are predicates over join points to determine the application of advice. Advice is code that executes after, before or instead of the computation represented by a join point. An aspect is an abstraction to specify crosscutting behavior.

Exceptions are a mechanism to deal with abnormal states of computation in a uniform and modular way by a lexical separation between normal and error handling code. When an abnormal situation ocurrs, an exception is thrown and then propagates dynamically through the call stack in search for a suitable handler. A handler is a special code section which executes receiving an exception as parameter. If a handler is not found the exception is uncaught, usually aborting program execution.

Subtle interactions between aspect and base exceptions and handlers may arise. Aspect exceptions may be inadvertently handled by base handlers. Conversely, base exceptions may be caught by aspect handlers. For example, consider the AspectJ code of Listing 1. The foo method calls getConfiguration to set configValue. In case of failure, a default value is used. A Logging aspect advises around the getConfiguration method. If the logger object cannot find the log file it shall fail and throw a FileNotFoundException, which is caught by the base handler. Thus, the default value is used because the aspect failed, even in cases where getConfiguration would have returned normally.

```
class A {
   public void foo() {
      Integer configValue;
      try { configValue = getConfiguration();
      } catch(Exception ex) { configValue = DEFAULT}}
}
aspect Logging {
   Object around() : call(Integer getConfiguration()) {
      logger.append("Calling getConfiguration"); // FileNotFoundException
      return proceed();}
}
```

Listing 1: Base handler catches aspect exception

This situation arises because the exception handling mechanism merges aspect and base handlers and exceptions in a flat structure. We call this situation the *exception conflation problem*. The exception conflation problem is a generalization of the *Late Binding Handler Pattern* of Coelho *et al.*, which is described as: "...happens when an aspect is created to handle an exception, but the aspect intercepts a point in the program execution where the exception to be caught was already caught by a handler in the method call chain that connects the exception signaler to the aspect handler" [4].

In this paper we propose a semantics for an aspect-oriented language that discriminates aspect and base exceptions using *execution levels*, extending Tanter's original proposal [10]. The language ensures that by default there is no interference between aspect and base exceptions and provides level-shifting operators to flexibly

[*]Funded by a CONICYT Doctoral Scolarship.
[†]Partially funded by FONDECYT Project 1110051.

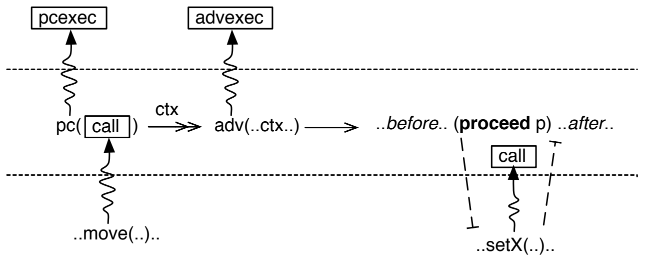

Figure 1: Execution levels in action: pointcut and advice are evaluated at level 1, *proceed* goes back to level 0 (from [10]).

specify their interactions when required. We illustrate the benefits of our proposal showing representative examples of interaction between aspect and base exceptions and handlers.

The rest of this paper is structured as follows: Section 2 recalls the notion of execution levels, Section 3 defines the semantics of the proposed language, Section 4 shows the applications of this language to exception handling issues, Section 5 discusses related work, and Section 6 concludes.

2. EXECUTION LEVELS IN A NUTSHELL

We first summarize the proposal of execution levels. An aspect observes the execution of a program through its pointcuts, and affects it with its advice. An advice is a piece of code, and therefore its execution also produces join points. Similarly, pointcuts as well can produce join points. For instance, in AspectJ, one can use an *if* pointcut designator to specify an arbitrary Java expression that ought to be true for the pointcut to match. The evaluation of this expression is a computation that produces join points. In higher-order aspect languages like AspectScript [12] and others, all pointcuts and advice are standard functions, whose application and evaluation produce join points as well.

The fact that aspectual computation produces join points raises the crucial issue of the *visibility* of these join points. In most languages, aspectual computation is visible to all aspects—including themselves. This of course opens the door to infinite regression and unwanted interference between aspects. These issues are typically addressed with ad-hoc checks (*e.g.* using **!within** and **cflow** checks in AspectJ) or primitive mechanisms (like AspectScheme's **app/prim**). However, all these approaches eventually fall short for they fail to address the fundamental problem, which is that of *conflating* levels that ought to be kept separate [2].

In order to address the above issue, Tanter proposed execution levels for AOP [10]. A program computation is structured in *levels*. Computation happening at level 0 produces join points observable at level 1 only. Aspects are *deployed* at a particular level, and observe only join points at that level. This means that an aspect deployed at level 1 only observes join points produced by level-0 computation. In turn, the computation of an aspect (*i.e.* the evaluation of its pointcuts and advice) is reified as join points visible at the level immediately above: therefore, the activity of an aspect standing at level 1 produces join points at level 2.

An aspect that acts *around* a join point can invoke the original computation. For instance, in AspectJ, this is done by invoking *proceed* in the advice body. The original computation ought to run at the same level at which it originated![1] In order to address this issue, it is important to remember that when several aspects match the same join point, the corresponding advice are chained, such that calling **proceed** in advice k triggers advice $k+1$. Therefore, the semantics of execution levels guarantees that the *last call* to **proceed** in a chain of advice triggers the original computation at the lower original level.

[1]This issue is precisely why using control flow checks in AspectJ in order to discriminate advice computation is actually flawed. See [10] for more details.

This is shown in Figure 1. A call to a `move` method in the program produces a call join point (at level 1), against which a pointcut `pc` is evaluated. The evaluation of `pc` produces join points at level 2. If the pointcut matches, it passes context information `ctx` to the advice. Advice execution produces join points at level 2, except for **proceed**: control goes back to level 0 to perform the original computation, then goes back to level 1 for the after part of the advice.

Level-shifting operators. The default semantics of execution levels treats aspects as a meta-level computation. However, in some cases, advice execution should be visible to aspects that observe base level execution. To reconcile both approaches, Tanter proposed explicit level-shifting operators: **up** and **down**. Shifting an expression using **up** or **down** moves its computation one level above or below, affecting the visibility of its join points. With these operators the programmer can specify the level at which computation is performed, according to specific needs.

Level-capturing functions. Certain applications require delayed execution of operations, for example: a prioritized command queue may accumulate a number of requests before executing them or a task scheduler may execute certain tasks at determined time intervals. In these cases, the operations may not be performed in the same execution thread in which they were declared, so the control flow checks to avoid regression problems fail. In addition, the execution may be performed at a different level than when it was postponed. To address this issue Tanter introduces level-capturing functions that keep track of the level at which they were declared. When a level-capturing function is executed, the execution level shifts to the function-captured level, and then shifts back afterwards.

By separating execution into levels, unwanted interactions between aspects are avoided. For instance, it becomes possible to reuse off-the-shelf dynamic analysis aspects and apply them to a given program, and/or aspects, with consistent semantics [11].

3. SEMANTICS

In this section we first recall the core syntax and semantics of Tanter's original proposal [10], which we then extend with exception handling. The complete semantics for the original language and the exception handling extensions are defined using PLT Redex, a domain-specific language for specifying reduction semantics [5]. The complete semantics implementation, along an executable test suite, and examples shown in Section 4 are available at `http://pleiad.cl/research/scope`.

3.1 Original Semantics

Tanter's proposal defines a simple Scheme-like language with higher-order aspects, and execution levels as described in Section 2. The language has booleans, numbers and lists, primitives functions to operate on these, and an *internal* `app/prim` operator to apply functions without generating join points [10]. Figure 2 shows the core syntax and reduction rules of the language. The user-visible expressions are values, identifiers, **if** statements, multi-arity function application, and aspect deployment. These are shown in **bold** font. The other expressions shown in the figure are related to level-shifting operations, and are shown with `typewriter` font.

A reduction relation $\hookrightarrow$ describes the operational semantics of

$$
\begin{array}{llll}
\mathit{Value} & v & ::= & (\lambda(x\cdots)\ e) \mid (\lambda^{\bullet}(x\cdots)\ e) \\
 & & & \mid n \mid \#t \mid \#f \\
 & & & \mid (\textbf{list}\ v\cdots) \mid \mathit{prim} \mid \textit{unspecified} \\
 & \mathit{prim} & ::= & \textbf{deploy} \mid \textbf{list} \mid \textbf{cons} \mid \textbf{car} \mid \textbf{cdr} \\
 & & & \mid \textbf{empty?} \mid \textbf{eq?} \mid + \mid - \mid \ldots \\
\mathit{Expr} & e & ::= & v \mid x \mid (e\,e\cdots) \mid (\textbf{if}\ e\ e\ e) \\
 & & & \mid (\textbf{up}\ e) \mid (\textbf{down}\ e) \\
 & & & \mid (\texttt{in-up}\ e) \mid (\texttt{in-down}\ e) \\
 & & & \mid (\texttt{in-shift}(e)) \\
 & & & \mid (\texttt{app/prim}\ e\ e\cdots) \\
 & v & \in & \mathscr{V}\text{, the set of values} \\
 & n & \in & \mathscr{N}\text{, the set of numbers} \\
 & \mathit{list} & \in & \mathscr{L}\text{, the set of lists} \\
 & x & \in & \mathscr{X}\text{, the set of variable names} \\
 & e & \in & \mathscr{E}\text{, the set of expressions} \\
\mathit{EvalCtx} & E & ::= & [\,] \mid (v\cdots\ E\ e\cdots) \mid (\textbf{if}\ E\ e\ e) \\
 & & & \mid (\textbf{up}\ E) \mid (\textbf{down}\ E) \\
 & & & \mid (\texttt{in-up}\ E) \mid (\texttt{in-down}\ E) \\
 & & & \mid (\texttt{in-shift}(l)\ E) \\
 & & & \mid (\texttt{app/prim}\ v\cdots\ E\ e\cdots)
\end{array}
$$

$$
\begin{array}{lr}
\langle l, J, E[(\textbf{up}\ e)]\rangle \hookrightarrow \langle l+1, J, E[(\texttt{in-up}\ e)]\rangle & \textsc{InUp} \\
\langle l, J, E[(\texttt{in-up}\ v)]\rangle \hookrightarrow \langle l-1, J, E[v]\rangle & \textsc{OutUp} \\
\langle l, J, E[(\textbf{down}\ e)]\rangle \hookrightarrow \langle l-1, J, E[(\texttt{in-down}\ e)]\rangle & \textsc{InDwn} \\
\langle l, J, E[(\texttt{in-down}\ v)]\rangle \hookrightarrow \langle l+1, J, E[v]\rangle & \textsc{OutDwn} \\
\langle l, J, E[(\lambda^{\bullet}(x\cdots)\ e)\rangle & \textsc{Capture} \\
\qquad \hookrightarrow \langle l, J, E[(\lambda^{l}(x\cdots)\ e)]\rangle & \\
\langle l, J, E[(\texttt{app/prim}\ (\lambda(x\cdots)\ e)\ v\cdots)]\rangle & \textsc{AppPrim} \\
\qquad \hookrightarrow \langle l, J, E[e\{v\cdots/x\cdots\}]\rangle & \\
\langle l_1, J, E[(\texttt{app/prim}\ (\lambda^{l_2}(x\cdots)\ e)\ v\cdots)]\rangle & \textsc{AppShift} \\
\qquad \hookrightarrow \langle l_2, J, E[(\texttt{in-shift}(l_1)\ e\{v\cdots/x\cdots\})]\rangle & \\
\langle l_2, J, E[(\texttt{in-shift}(l_1)\ v)\rangle \hookrightarrow \langle l_1, J, E[v]\rangle & \textsc{Shift}
\end{array}
$$

Figure 2: Core syntax and reduction rules of the language.

our language using reduction steps. The relation $\hookrightarrow$ is defined as follows[2]: $\hookrightarrow: \mathscr{L} \times \mathscr{J} \times \mathscr{E} \rightarrow \mathscr{L} \times \mathscr{J} \times \mathscr{E}$

An evaluation context consists of an execution level $l \in \mathscr{L}$, a join point stack $J \in \mathscr{J}$ and an expression $e \in \mathscr{E}$. The reduction relation takes a level, a join point stack, and an expression and maps this to a new evaluation step. Join point definition, aspect deployment and the weaving mechanism is described in [10]. The exception handling extension does not alter these definitions nor any reduction rules of the original language.

Primitive application. The language features a primitive function application, `app/prim`, that does not generate join points. It performs a simple β_v reduction of the expression. This mechanism is required for tasks such as the initial application of the composed advice chain (and its recursive calls), and to perform the original computation when all aspects (if any) have proceeded. This is shown in rule AppPrim.

Level-shifting operators. The **up** (**down**) level-shifting operator embeds its inner expression in an `in-up` (`in-down`) expression, which increases (lowers) the current execution level. When the embedded expression is reduced to a value, the execution level is decreased (increased) to the original level. This is specified by the rules InUp, InDwn, OutUp, OutDwn. Aspect weaving ensures that pointcuts and advices are evaluated with **up**, and that the last **proceed** in the chain is evaluated with **down** [10].

Level-capturing functions. Level-capturing functions are defined using $\lambda^{\bullet}$. A function value bound to level l is denoted λ^{l}. Rule Capture shows that when a level-capturing function is defined, it is tagged with the current execution level. Rule AppShift shows that when a level-capturing function is applied, it embeds the reduction of the application in an `in-shift` expression. By rule Shift, when the expression reduces to a value, the execution level shifts back to the level embedded in the `in-shift` form.

Aspect deployment. Aspects can be dynamically deployed. The **deploy** expression takes a pointcut and an advice and adds the aspect to the current aspect environment. To deploy aspects of aspects we use the level-shifting operators to shift the level at which the **deploy** expression is evaluated. The complete rules for aspect deployment are shown in Tanter's proposal [10], and are not changed in our proposal.

3.2 Exception Handling Semantics

We introduce a standard exception handling mechanism [9], using the extensions shown in Figure 3. We extend the user-visible syntax with the **raise** and **try-with** expressions and their respective evaluation contexts. We also define an *Exception* normal form annotated with the level at the which the exception was raised. Then we add reduction rules to specify the semantics of the **raise** and **try-with** expression. In essence, our proposal consists in tagging exceptions and handlers with their respective execution level; then, an exception is only caught by a suitable handler if they are both bound at the same level.

Safe default. A **try** e **with** e_h expression contains a protected expression e and a handler expression e_h. If an exception is raised during the reduction of e, e_h is evaluated, and the resulting function is applied to the exception, *only if* the level of the exception matches the level of the handler. This default semantics ensures that there is no interaction between aspect and base exceptions and handler. For instance, if there are no explicit level shifts, the computation of e happens at level 0. If an exception is thrown by the base code, it will be a level-0 exception, which will therefore be caught by the handler. Conversely, if the exception is thrown in an aspect (at level 1), the handler will not catch the exception.

Certainly, there are situations where interaction is desired. To this end, we exploit the fact that the handler is an expression to be evaluated to install a handler in an upper (or lower) level using level-capturing functions and level shifting operators. This is illustrated in Section 4.

Raising exceptions. The **raise** expression signals an exception that embeds a value to carry information to the handler. Rule Raise-Capture creates a tagged exception, which holds the execution level at which the exception is raised. An exception bound at level l is denoted ($\textbf{raise}^{l}\,v$).

Exception propagation. The rule Reraise deals with exception propagation in nested **raise** expressions. An exception propagates through the `in-up` and `in-down` expressions maintaining its tagged level. For simplicity we omit the propagation rules here. They are present in the downloadable complete semantics specification.

Handling exceptions. As reflected by rule TryV, if the protected expression of a **try-with** reduces to a value, the whole **try-with** expression reduces to that value. Otherwise, the reduction is determined by one of these rules: HndEx1, HndProp1, HndEx2 and HndProp2. It is important to observe that the **try** ex **with** E execution context ensures that the handler expression is only eval-

[2]The complete semantics we provide includes the aspect environment in the reduction rules, omitted here for simplicity.

$$
\begin{array}{llll}
\mathit{Expr} & e & ::= & \cdots \mid (\textbf{raise}\ e) \mid (\textbf{try}\ e\ \textbf{with}\ e) \\
\mathit{Exception} & ex & ::= & (\textbf{raise}^{l}\ v) \\
\mathit{EvalCtx} & E & ::= & \cdots \mid (\textbf{raise}\ E) \mid (\textbf{try}\ E\ \textbf{with}\ e) \\
 & & & \mid (\textbf{try}\ ex\ \textbf{with}\ E)
\end{array}
$$

$$
\begin{array}{lr}
\langle l, J, E[(\textbf{raise}\ v)]\rangle & \textsc{RaiseCapture} \\
\quad \hookrightarrow \langle l, J, E[(\textbf{raise}^{l}\ v)]\rangle & \\
\langle l, J, E[(\textbf{raise}\ (\textbf{raise}^{l_1}\ e))]\rangle & \textsc{Reraise} \\
\quad \hookrightarrow \langle l, J, E[(\textbf{raise}^{l_1}\ e)]\rangle & \\
\langle l, J, E[(\textbf{try}\ v\ \textbf{with}\ e)]\rangle & \textsc{TryV} \\
\quad \hookrightarrow \langle l_1, J, E[v]\rangle & \\
\langle l, J, E[(\textbf{try}\ (\textbf{raise}^{l}\ v)\ \textbf{with}\ (\lambda(x\cdots)\ e)]\rangle & \textsc{HndEx1} \\
\quad \hookrightarrow \langle l, J, E[((\lambda(x\cdots)\ e)\ v)]\rangle & \\
\langle l, J, E[(\textbf{try}\ (\textbf{raise}^{l_1}\ v)\ \textbf{with}\ (\lambda(x\cdots)\ e))]\rangle & \textsc{HndProp1} \\
\quad \hookrightarrow \langle l, J, E[(\textbf{raise}^{l_1}\ v)]\rangle \text{ where } l_1 \neq l & \\
\langle l, J, E[(\textbf{try}\ (\textbf{raise}^{l_1}\ v)\ \textbf{with}\ (\lambda^{l_1}(x\cdots)\ e)]\rangle & \textsc{HndEx2} \\
\quad \hookrightarrow \langle l_1, J, E[(\texttt{in-shift}(l)\ ((\lambda^{l_1}(x\cdots)\ e)\ v))]\rangle & \\
\langle l, J, E[(\textbf{try}\ (\textbf{raise}^{l_1}\ v)\ \textbf{with}\ (\lambda^{l_2}(x\cdots)\ e))]\rangle & \textsc{HndProp2} \\
\quad \hookrightarrow \langle l, J, E[(\textbf{raise}^{l_1}\ v)]\rangle \text{ where } l_1 \neq l_2 &
\end{array}
$$

Figure 3: Exception handling extensions.

uated when an exception is raised in the evaluation of the protected expression.

The HNDEX1 and HNDPROP1 rules deal with normal function handlers: if the exception level matches the current execution level, the handler is applied; else, the exception propagates in search of a suitable handler. Rules HNDEX2 and HNDPROP2 deal with level-capturing function handlers: regardless of the current execution level, if the exception and handler level match then the handler is applied shifting the current execution level to the level of the handler; otherwise the exception keeps its propagation.

4. APPLICATIONS

To illustrate the benefits of our proposal, in this section we show a set of four representative examples of interaction between aspect and base exceptions and handlers: no interference between base exceptions and aspect handlers (Listing 2); no interference between base handlers and aspect exceptions (Listing 3); an aspect handler catching a base level exception (Listing 4), and a base handler catching an aspect exception (Listing 5). These minimal examples show that the semantics proposed in Section 3.2 solve the exception conflation problem, while still enabling interesting programming patterns, like aspects explicitly handling base exceptions.

For all the examples we assume that the pointcut associated to the advice matches any function call. The advice is a function which receives at least two parameters: p is the call to proceed, and c is the return value of the pointcut application to the join point. We also assume that the aspect is bound at level 1, so it observes only joint points emitted from base function calls.

Additional parameters are required for each parameter of the advised function. In all the examples each function takes exactly one argument, so each advice has three parameters: p, c, and arg which holds the value of the advised function parameter.

In Listing 2 the function of the normal expression in the **try-with** generates a call join point at level 1; the pointcut matches the join point and the advice raises an exception at level 1. The exception propagates back to the **try-with** expression and as the handler is at level 0, the exception is not caught. In consequence, the base expression reduces to (**raise**^{1} #f).

```
;; Advice
(λ (p c arg) (raise #f))

;; Base code
(try ((λ (x) x) #t) with (λ (ex) ex))
```

Listing 2: By default there is no interference between base exceptions and aspect handlers. Base expression evaluates to (raise^1 #f).

In Listing 3 the base generates a call join point at level 0; when the advice calls the proceed function p, an exception is raised at level 0 (remember that the last call to proceed executes at the original level, see Section 2) and as the handler in the advice is at level 1 the exception is uncaught. Thus, the base expression reduces to (**raise**^{0} #f).

```
;; Advice
(λ (p c arg) (try (p arg) with (λ (ex) ex)))

;; Base code
((λ (x) (raise #f)) #f)
```

Listing 3: By default there is no interference between aspect exceptions and base handlers. Base expression evaluates to (raise^0 #f).

Listing 4 shows the same situation as in Listing 3, except for the handler in the advice code. In this case, we shift down the evaluation of a level-capturing function using the **down** level-shifting operator. This causes the handler function to be bound at level 0. When the call to proceed raises the exception bound at level 0, the handler catches it because they are both bound to the same level. Note that the handler execution is a function application that happens at the level the handler is bound. This application also generates a call join point at level 0, and the advice applies again, but in this case no exception is raised in the call to proceed. Hence the original base code expression reduces to #f.

```
;; Advice
(λ (p c arg) (try (p arg) with (down (λ• (ex) ex))))

;; Base code
((λ (x) (raise #f)) #f)
```

Listing 4: Using a level-capturing function and the down level-shifting operator to catch a base exception in an aspect handler. Base expression evaluates to #f.

Finally, Listing 5 shows the same situation as in Listing 2, but with a different handler. In this case we use the **up** level-shifting operator to bind the handler function at level 1. When the aspect exception propagates to the **try-with** expression, the handler catches the exception and executes at level 1. In this case the advice does not execute again because the call join generates at level 2, and the aspect does not *see* it.

```
;; Advice
(λ (p c arg) (raise #f))

;; Base code
(try ((λ (x) x) #t) with (up (λ• (ex) ex))
```

Listing 5: Using a level-capturing function and the up level-shifting operator to catch an aspect exception in a base handler. Base expression evaluates to #t.

For each example, the handler has a normal function or a level-capturing function. The dual situation in which the handler is of the other kind is omitted. In the examples of Listing 2 and Listing 3 the program evaluates to the same result using a level-capturing function. In the other two examples, using a non level-capturing function the handler level does not match the exception level so the exception propagates.

The examples show the key interactions between aspect and base handlers and exceptions, and the mechanism that our semantics provide to the programmer to specify the desired interaction. Other situations like aspects of aspects can be reduced to one of the examples.

5. RELATED WORK

The first approach dealing with exception handling as a cross-cutting concern was the study done by Lippert and Lopes [8]. They refactored a Java business application framework, called *JWAM*, using an old version of AspectJ to separate normal code from error handling code, in order to promote independent evolution and reusability of each section. They significantly reduced the LOC in the framework, managed to reuse common exception handling code and described several limitations of the AspectJ version used.

Castor Filho *et al.* [7] studied the adequacy of AspectJ for modularizing and reusing exception-handling code. They studied five systems: four object-oriented and one aspect-oriented. The object-oriented systems had their exception-handling code refactored to exception aspects. They obtained quantitative results applying a metrics suite based on four quality attributes: separation of concerns, coupling, cohesion and conciseness, to the systems. They also obtained qualitative results, they discuss several issues like best practices for aspect-oriented exception handling, interaction between exception-handling aspects and other aspects and scalability issues. Their main conclusions are that the mere use of aspects to handle exceptions is not sufficient to improve the quality of software, a careful design from the early phases of development is required to properly aspectize exception handling. A follow-up study addresses different design scenarios to aspectize exception handling in object-oriented systems [6].

Coelho *et al.* [3] study the interaction between aspects and exceptions in three systems with a Java and AspectJ versions available. They categorized the exception paths in the systems and the most common handlers strategies. Using their own static analysis tool they compare the object-oriented and aspect-oriented versions of each system. Using this information they developed a catalogue of exception-handling bug patterns in aspect-oriented programs [4].

6. CONCLUSION AND FUTURE WORK

In this paper we showed that interaction between aspect and base exceptions and handlers is prone to unintended execution of handlers and a lack of flexibility for the programmer. This situation ocurrs because the exception handling mechanisms do not distinguish between aspect and base exceptions. As a solution, we designed and described the semantics of a higher-order aspect language with execution levels and exceptions. Our language by default ensures there is no interaction between aspect and base handlers and exceptions and provides level-shifting operators to specify the interaction between them. We then showed four representative examples of interaction between aspects and base exceptions and how our language solves the exception conflation problem by default, and still provides the necessary flexibility when needed.

To further illustrate the benefits of our proposal, we plan to extend the AspectJ implementation with execution levels described by Tanter *et al.* in [11] with our exception handling semantics. Using this implementation we will make case studies similar to the ones done by Coelho *et al.* in [3], as well as some others like using the Contract4J design-by-contract framework. Other issues to consider are the interactions between exception handling mechanisms and the type system, the contrasts between languages with checked and unchecked exceptions, such as Java and C#; and the presence of finally blocks.

7. REFERENCES

[1] *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, Rennes and Saint Malo, France, Mar. 2010. ACM Press.

[2] S. Chiba, G. Kiczales, and J. Lamping. Avoiding confusion in metacircularity: The meta-helix. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer-Verlag, 1996.

[3] R. Coelho, A. Rashid, A. Garcia, N. Cacho, U. Kulesza, A. Staa, and C. Lucena. Assessing the impact of aspects on exception flows: An exploratory study. In J. Vitek, editor, *Proceedings of the 22nd European Conference on Object-oriented Programming (ECOOP 2008)*, number 5142 in Lecture Notes in Computer Science, pages 207–234, Paphos, Cyprus, july 2008. Springer-Verlag.

[4] R. Coelho, A. Rashid, A. von Staa, J. Noble, U. Kulesza, and C. Lucena. A catalogue of bug patterns for exception handling in aspect-oriented programs. In *PLoP '08: Proceedings of the 15th Conference on Pattern Languages of Programs*, pages 1–13, New York, NY, USA, 2008. ACM.

[5] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[6] F. Filho, A. Garcia, and C. Rubira. Extracting error handling to aspects: A cookbook. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 134 –143, Oct. 2007.

[7] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, and C. M. F. Rubira. Exceptions and aspects: the devil is in the details. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 152–162, New York, NY, USA, 2006. ACM.

[8] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, New York, NY, USA, 2000. ACM.

[9] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[10] É. Tanter. Execution levels for aspect-oriented programming. In AOSD 2010 [1], pages 37–48. Best Paper Award.

[11] É. Tanter, P. Moret, W. Binder, and D. Ansaloni. Composition of dynamic analysis aspects. In *Proceedings of the 9th ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE 2010)*, pages 113–122, Eindhoven, The Netherlands, Oct. 2010. ACM Press.

[12] R. Toledo, P. Leger, and É. Tanter. AspectScript: Expressive aspects for the Web. In AOSD 2010 [1], pages 13–24.