An Aspect for Idiom-based Exception Handling

(using local continuation join points, join point properties, annotations and type parameters)

Bram Adams
GH-SEL, INTEC, Ghent University
Bram.Adams@UGent.be

Kris De Schutter
LORE, University of Antwerp
Kris.DeSchutter@UA.ac.be

Abstract

The last couple of years, various idioms used in the 15 MLOC C code base of ASML, the world's biggest lithography machine manufacturer, have been unmasked as crosscutting concerns. However, finding a scalable aspect-based implementation for them did not succeed thusfar, prohibiting sufficient separation of concerns and introducing possibly dangerous programming mistakes. This paper proposes a concise aspect-based implementation in Aspicere2 for ASML's exception handling idiom, based on prior work of join point properties, annotations and type parameters, to which we add the new concept of *(local) continuation join points*. Our solution takes care of the error value propagation mechanism (which includes aborting the main success scenario), logging, resource cleanup, and allows for local overrides of the default aspect-based recovery. The highly idiomatic nature of the problem in tandem with the aforementioned concepts renders our aspects very robust and tolerant to future base code evolution.

Categories and Subject Descriptors D.1.m [*Programming Techniques*]: Aspect-oriented programming; D.2.5 [*Testing and Debugging*]: Error handling and recovery; D.2.7 [*Software Engineering*]: Restructuring, reverse engineering, and reengineering; D.3.2 [*Programming Languages*]: C; D.3.2 [*Programming Languages*]: Prolog; D.3.m [*Programming Languages*]: Local continuation join point

General Terms Design, Languages, Reliability

1. Introduction

Exception handling is one of the more fundamental issues encountered during software development, especially in legacy programming languages like C or Cobol. C in particular lacks any dedicated means to tackle exceptions, so over time people have resorted to all kinds of tricks to emulate them [6]: setjmp/longjmp, global error variables, signals, returning error values, etc.

In [2] the authors discuss a variant of the so-called "return-code idiom" as the preferred means for exception handling in the 15 MLOC C code base of ASML, the world's biggest lithography system manufacturer. This idiom is part of a global idiom-based software development strategy for safeguarding the machines' reliability and functioning.

Workshop SPLAT '07 March 12-13, 2007 Vancouver, British Columbia, Canada.

However, for idioms to work properly, developers need to be disciplined enough to use them correctly and consequently. Also, the choice for one particular idiom ties the code base almost exclusively to the chosen pattern, making it very hard to migrate to or experiment with other approaches.

Exception handling has been identified before [7] as a typical crosscutting concern, i.e. it exhibits excessive scattering (throughout the whole system) and tangling to the point where it severely obscures normal flow. However, as we will see, abstracting the return-code idiom into an aspect requires a combination of concepts which are not mainstream in aspect languages. Without these features, writing a useful, flexible exception handling aspect for systems written in legacy programming languages is hard to achieve.

The contributions of this paper are as follows:

- the introduction of the concept of local continuation join points;
- their application, together with join point properties, annotations and type parameters, to the idiom-based exception handling pattern;
- and a discussion of the benefits of our approach as well as of some implementation issues.

First (section 2), we will introduce the specific details of ASML's return-code idiom using the running example of [2], eventually identifying the core aspect of the problem. This concern is modeled using local continuation join points, which are introduced and applied on the running example in section 3. Section 4 treats the logging part of the idiom using join point properties and annotations, whereas section 5 looks at the resource cleanup concern. Section 6 discusses our approach and some implementation issues. Finally, section 7 concludes this paper.

2. The return-code idiom

As its name implies, ASML's return-code idiom is based on the dedicated use of return values for passing error values up the call chain. Alongside this, errors have to be logged in a so-called "event log" in order to accommodate off-line exception analysis. In the following subsections we will dive deeper into the actual code behind this idiom, based on the reports by Bruntink et al [2, 1].

2.1 Implementation details

Each procedure is given a special local variable in which the current error status (initially `OK`) is stored, although more than one variable is necessary in some cases like parallel execution or errors during resource cleanup. Whenever an error occurs directly inside a procedure (*not* from within a called procedure), the developer has one of two choices:

- recover from the error immediately, or

```
1  int f(int a, int** b){
     int r = OK;
3    bool allocated = FALSE;
     r = mem_alloc(10, (int**) b);
5    allocated = (r == OK);

7    if((r == OK) && ((a < 0) || (a > 10))){
       r = PARAM_ERROR;
9      LOG(r,OK); /*root error*/
     }
11   if(r == OK){
       r = g(a);
13     if(r != OK){
         LOG(LINKED_ERROR,r); /*linked error*/
15       r = LINKED_ERROR;
       }
17   }
     if(r == OK) r = h(b);
19   if((r != OK) && allocated) mem_free(b);
     return r;
21 }
```

Figure 1. Example of the return-code idiom, based on [2].

- abort the procedure and propagate the error back to its caller. This entails logging of the error in an entry called a "root error", followed by transfer of control flow to the procedure's caller.

Fig. 1 shows an example of this (based on [2]). The variable called `r` on line 2 holds the error status of procedure `f`. On line 7, parameter `a` is checked to see whether it lies in the range of $[0, 10]$. If not, the error status gets updated (line 8), and this error is logged as a root error (line 9). The remaining logic is then skipped until line 19, where cleanup takes place.

Any calling procedure at its turn has the same choice of either handling the propagated error or passing it on. In the latter case, it has the possibility to add extra context information by replacing the original error value by another, (possibly) more meaningful one. The change of error value has to be logged, and this kind of entry is now called a "linked error" as it links a higher-level error value to a lower-level one. In the end, this gives rise to an "error link tree", i.e. an exception trace for a particular error. Line 14 of Fig. 1 demonstrates this logging of a linked error (the original error having occurred during the execution of procedure `g` on line 12). Note that a root error is a special case of a linked error, as it links an error to the `OK` value. If ultimately no exception handler is found, the system could go down.

Aside from error variable management, control flow transfer and logging, resource cleanup (memory in this case) plays an important role (line 19 on Fig. 1). If an error has occurred, any previously allocated memory in the current procedure needs to be deallocated. This concern is not treated in [2], but we will look at it to illustrate how the various concerns fit together.

It is clear from the example that the idiom's logic seriously overcrowds the procedure's main control flow. There is both serious tangling and scattering, as the idiom is applied system-wide. Even for such a simple example as Fig. 1 it is hard to deduce e.g. what are all possible execution paths that get through the if-checks of lines 18 or 19. This of course hampers any maintenance and/or re-/reverse-engineering efforts. From the analysis in [2], it turns out that most developer errors regarding the idiom are caused by erroneous guards (i.e. checks) on the error variable `r` and inconsistencies between the logged error value and the one assigned to `r`. Other common errors include forgetting to return an error value, returning the wrong value, logging incorrect things, etc.

We would like to use aspects to relieve the base code developer from the return-code idiom burden, while still giving him the power to override default exception handling if he desires to. In order to do this, we will first take a closer look at Fig. 1's hidden patterns.

2.2 Distinguishing the different concerns

```
1  int f(int a, int** b){
     int r = OK;
3    r = mem_alloc(10, (int**) b);

5    if(r != OK){
       /* no logging needed */
7      /* no deallocation needed */
       return r;
9    }else{
       if((a < 0)||(a > 10)){
11       r = PARAM_ERROR;
         LOG(r,OK);
13       if(r != OK) mem_free(b);
         return r;
15     }else{
         r = g(a);
17       if(r != OK){
           LOG(LINKED_ERROR,r);
19         r = LINKED_ERROR;
           if(r != OK) mem_free(b);
21         return r;
         }else{
23         r = h(b);
           if(r != OK){
25           /* no logging needed */
             if(r != OK) mem_free(b);
27           return r;
           }else{
29           /* no deallocation needed */
             return r;
31         }
         }
33     }
     }
35 }
```

Figure 2. Restructured version of Fig. 1. Crosscutting concerns are marked with colored F-shapes, rectangles and underlining.

What contributes most to the program complexity is the delicate interplay of main logic, exception handling and resource (memory) cleanup. What is more, exception handling actually encompasses various subconcerns: error variable management, control flow transfer, logging linked errors, detecting root errors, etc. To make these patterns stand out in the code, we rewrote Fig. 1's abbreviated programming style into the equivalent, more verbose Fig. 2. Now, we can clearly identify the various concerns:

- the declaration, initialisation and returning of a unique error variable (the doubly underlined code on lines 2 and 30),
- the "assign return values to the error variable"-concern (singly underlined lines),
- the control flow transfer which either continues normal execution of a procedure or aborts it by returning the error value (the three light F-shapes),
- the logging of linked errors (the three dark rectangles),
- memory cleanup (the light rectangles),
- the argument range checking concern (the dark F-shape), and
- the main logic (remaining code).

Our goal now is this: to extract all concerns into aspects, thereby allowing the base code to be reduced to its main logic, as shown on

```
/*@range("a",0,10)*/
int f(int a, int** b){
  mem_alloc(10, (int**) b);

  /*@log("LINKED_ERROR")*/
  g(a);
  h(b);
}
```

Figure 3. Main logic of Fig. 1 to which all aspects and their logic rules and facts are applied.

Fig. 3 (ignore the comments for now). We impose the following restrictions:

- The procedures' signatures remain unaltered in order to allow for gradual, stepwise migration (no broken interfaces!). I.e. all procedures will retain their integer typed return value, which can be used as an error value by some aspect-based exception handling implementations, while others may ignore it.
- In our aspects, we will keep as close to the spirit of the return-code idiom as possible, again keeping in mind a gradual, stepwise migration. However, nobody disallows experimentation with setjmp/longjmp or global error variables later on.

For the purpose of this paper, we implement everything in Aspicere2[1], the new version of Aspicere, our aspect language for C [9]. Advice in Aspicere2 is a special construct featuring a Prolog-based pointcut and a body written in C. Context (types, constants, etc.) gathered during pointcut matching is readily available in the advice body. Weaving happens at link-time and involves transformations on an intermediate representation of the source code. We will present Aspicere2's features as we go, and as needed.

3. Control flow transfer through local continuation join points

From the list of concerns, the one governing the transfer of control flow strikes us as the most fundamental and hardest one. We will therefore discuss it first.

3.1 Fundamental issues

The following pseudo-code forms the heart of the control flow transfer problem:

```
if(r != OK){
  return r;
}else{
  /* continue */
}
```

After each idiomatic (i.e. returning error values) procedure call, the execution of the enclosing procedure either halts or continues depending on the call's returned error value (see light F-shapes of Fig. 2).

We argue that this behaviour is not possible to achieve with current aspect technology for legacy languages. A naive approach would be to put advice around every procedure call in order to only continue the call if no error has occurred yet in the current procedure. Once an error happens all remaining calls within the current procedure will be short-circuited. Unfortunately, accesses of local variables, calculations (more than likely containing bad operands by now), loops, gotos, etc. cannot be skipped and will potentially lead to catastrophe. Would adding extra join point types for loops, gotos, variable accesses, etc. and short-circuiting them help? It would work, but the continuous checks to see whether to skip a join point or not incurs a lot of redundant overhead. Also, it foregoes the more fundamental thing one tries to accomplish here: skipping the remaining execution of a procedure once a call within it contains an error value.

[1] http://users.ugent.be/~badams/aspicere2/

```
1  void f(void){
2    printf("A");
3    do_something();
4    printf("B");
5  }
6  int main(void){
7    f();
8    printf("C");
9    return 0;
10 }
```

Figure 4. Small example highlighting the differences between (local) continuation join points.

The approach that comes closest to this is to rewrite procedures in continuation passing style [5] such that the call to the continuation argument can be circumvented by withholding a proceed-call. Migrating to this style of programming is as huge an undertaking as would be abandoning the act of returning an error value. Instead we prefer to introduce a new concept: *local continuation join points*.

The concept of continuation join points presented here differs from that in [4]. There, Endoh, Masuhara and Yonezawa use the term in reference to a formalisation technique used for defining the semantics of their join point model. We, instead, use the term in the sense of a programming language concept, as will become apparent in the following sections.

3.2 Definition

A *continuation* at any point in the execution of a program can be informally defined as the future execution of that program from that point on. E.g. in:

```
printf (getLine ());
```

the initial continuation is the entire program. The continuation after application and evaluation of `getLine` is the application and evaluation of `printf`.

Certain programming languages are able to turn the concept of a continuation into a first-class entity, which can subsequently be manipulated by the program (mainly re-invocation, which causes the control flow to return to the point at which the continuation was made). This is for instance the case in Smalltalk, where it was used to great effect for the Seaside web application framework [3].

In the context of aspect-oriented programming, we now turn the concept to use in a join point model. As such we define:

> the *continuation* of a join point *p* as (a join point representing) the future execution after conclusion of *p*.

The reification of the continuation of a join point as another join point is what makes it possible for this continuation to become the target of aspectual advice.

Let us apply this definition to the code in Fig. 4. Without any advice this program outputs "ABC". Short-circuiting the continuation of the call to `do_something` on line 3 yields "A" only, as the entire remainder of the program is skipped. The return value of the continuation corresponds to that of the program (i.e. `int`) and could e.g. hold a meaningful error value.

As it is, this construct is too strong for our purposes. Indeed, it captures the entire future execution of a program, whereas we are only interested in this future execution up to the end of the current executing procedure. We therefore introduce a reduced version:

> the *local continuation* of a join point *p* is (a join point representing) the future execution after conclusion of *p*, limited to the control flow of the procedure in which *p* is active.

Let us again apply this to the code in Fig. 4. Short-circuiting the call's *local* continuation join point now yields "AC", as only the remaining execution of procedure `f` is skipped. There is no return value here, as `f` is a `void`-procedure.

So, the concept of a local continuation join point allows us to capture the "remainder of the execution of a procedure", which is needed for the exception handling mechanism. As a sidenote, it seems that only around-advising local continuation join points of calls or variable accesses is useful. An execution join point's local continuation corresponds to nothing (or rather a no-op), while before-/after-advice on a call's local continuation is identical to after-advice on the call or its enclosing execution resp.

3.3 Syntax in Aspicere2

A local continuation join point is coded in Aspicere2 as follows. Given a join point `Jp` the local continuation of that join point is:

```
1  local_continuation(ContinuationJp, Jp)
```

In the spirit of the pointcut language of Aspicere2, `local_continuation` is a logic-based predicate. It takes any join point (here: `Jp`) and deduces the associated local continuation join point (here: `ContinuationJp`). The value for this is then available for further use in the pointcut and advice. In the case of around-advice, calling `proceed` will activate the join point (and hence the continuation) as expected. This (re-)activation can be done as many times as needed, including zero. The latter situation is at the heart of the exception handling mechanism.

3.4 Application to the control flow transfer

The exception handling aspect is shown in Fig. 5 while its accompanying logic rules are depicted in Fig. 6. For now, we focus on the control flow transfer advice of lines 37–45 (code with the shaded background). Advice `error_code_passing` implements the three light F-shapes of Fig. 2. Indeed, the advice body on lines 41–44 is nearly identical to the pseudo-code of section 3.1. The remarkable thing here is that the advice superimposes on join points `Jp` (line 37) which represent the local continuation of join points `JpCall` (line 40). The latter are so-called "idiomatic calls" (line 38), of which exception handling should not be manually overridden (line 39).

What exactly are idiomatic calls? Turning to the predicates shown in Fig. 6, we see (lines 14–20) that idiomatic calls are nothing more than invocations (calls) returning an integer and located inside idiomatic procedures (lines 17–18). The execution join point `JpEncl` of the latter has a property named `error_var` (line 19; more on this in section 4). We want to avoid calls to standard library procedures which accidentally return integers, hence the excluded wildcard pattern on line 16.

The only thing to explain now are idiomatic procedures (lines 9–12). It is possible that some in-house modules or external libraries deliberately do not take part in the return-code idiom. Their return values should not be interpreted as error values. The `idiomatic_proc`-predicate remedies this by limiting the aspects' scope to the relevant modules (e.g. "main.c" on line 11).

To summarise, whenever an error value is returned from an idiom-participating procedure call, the remaining procedure execution is skipped. Otherwise, it is business as usual. This is clearly illustrated in Fig. 7, which gives an overview of the role of each advice described in this paper and the join points it applies at.

```
1  /*necessary imports*/

3  int* intro error_var() on Jp:
     idiomatic_proc(Jp);
5
   int around error_code_mgmt(int* R) on Jp:
7    idiomatic_proc(Jp)
     && property(Jp,error_var,R){
9     *R=OK;
      proceed();
11    return *R;
   }
13
   void after error_code_resetting(int* R) on Jp:
15   idiomatic_call(Jp,R)
     && manual(Jp){
17    *R = OK;
   }
19
   void after error_code_logging(int* R,
21  int ErrorCode) returning (int* Return) on Jp:
     idiomatic_call(Jp,R)
23   && log(Jp,ErrorCode){
      if(*R != OK){
25      LOG (ErrorCode, *R);
        *R = ErrorCode;
27      *Return = ErrorCode;
      }
29 }

31 void after error_code_update(int* R)
    returning (int* Result) on Jp:
33   idiomatic_call(Jp,R){
      *R=*Result;
35 }

   int around error_code_passing(int* R) on Jp:
38   idiomatic_call(JpCall,R)
     && !!manual(JpCall)
40   && local_continuation(Jp,JpCall){
      if(*R!=OK)
42      return *R;
     else
44      return proceed();
   }
```

Figure 5. The idiom-based exception handling aspect. The shaded area corresponds to Fig. 2's light F-shapes (control flow transfer).

4. Logging and customisation through join point properties and annotations

The previous section's control flow transfer advice is only part of the story. In this section we will consider the logging concern, as well as the possibility of overriding the default aspect behaviour.

4.1 Concepts

Developers should be able to override flow transfer in case the default idiom does not suffice, e.g. to perform extra resource cleanup or to locally recover from expected errors right after the erroneous procedure call. We also need a way to automatically log linked errors, and here again developers should be able to decide whether linking is necessary and if so what the new error value should be. Both cases are concerned with providing the developers the necessary power to control advice execution.

A second issue relates to advice interaction. How does each advice know when an error has occurred and what its value is? Using a global error variable (or a stack thereof) will lead to race conditions in multi-threaded architectures. This problem will get even worse for the resource cleanup aspect. Also, it is at odds with the actual semantics of an error status, as these are tied to procedures.

To solve the second issue, we will use join point properties [8]. Join point properties are (name,value)-pairs attached to individual join points. The familiar `thisJoinPoint`-object of AspectJ e.g. is actually a container of join point properties provided by the system. In [8], the authors propose to let the user add custom properties as a means to communicate between various advices

```
1  error_code("LINKED_ERROR",0).

3  int_invocation(Jp,FName):-
     invocation(Jp,FName),
5    type(Jp,Type),
     type_name(Type,"int")
7    .

9  idiomatic_proc(Jp):-
     execution(Jp,_),
11   filename(Jp,"main.c")
     .
13
   idiomatic_call(JpCall,R):-
15   int_invocation(JpCall,FName),
     \+wildcard(".*printf",FName),
17   enclosingMethod(JpCall,JpEncl),
     idiomatic_proc(JpEncl),
19   property(JpEncl,error_var,R)
     .
21
   manual(JpCall):-
23   annotation(JpCall,manual,_)
     .
25
   log(JpCall,ErrorCode):-
27   annotation(JpCall,log,[ErrorName]),
     error_code(ErrorName,ErrorCode)
29   .
```

Figure 6. Accompanying Prolog metadata of the aspect in Fig. 5.

and, even more importantly, independently developed aspects. This eases advice interactions, and, if the properties are thread-local, it fits nicely with multi-threading environments.

Overriding the default behavior will be achieved using the concept of annotations. Annotations are well-known by now as AspectJ 1.5[2] has had them for some time. Put briefly, they correspond to metadata associated to program elements by means of a kind of tags. They convey information which is not expressible in the programming language itself. Of course, C does not have annotations by default, so we will abuse comments for this (as was the style in Java before annotations were turned into first-class entities). Our annotations are associated with the first statement or (nested) expression which follows them.

We prefer manipulation of advice through annotations above custom advice written by the developer for various reasons:

- During examination of advice interaction, one can focus on the few known system-wide aspects (guided by annotations) instead of (possibly) hundreds of small ones added by individual developers every day. This makes proving correctness of the aspect-enabled system more reasonable.
- Existing development tools (IDEs) do not need to change. Care must be taken that the weaver for system-wide aspects can be integrated nicely into the compiler chain[3].
- Aberrant exception handling behaviour remains localised.

4.2 Example

The exception handling aspect of Fig. 5 shows the application of join point properties and annotations for the developer overriding and logging advices (lines 1–35). Lines 3–4 contain the declaration of a join point property called `error_var` representing an error variable[4] (cf. the idiom itself) in each idiomatic procedure `Jp`. Of course, the variable should be initialised to the `OK` status every time a procedure starts executing and should be used as the procedure's return value at the end. This is handled by advice `error_code_mgmt` (lines 6–12). As shown here, a join point property (`R`) can be accessed via the `property`-predicate (line 8) by supplying both the join point and the right property name.

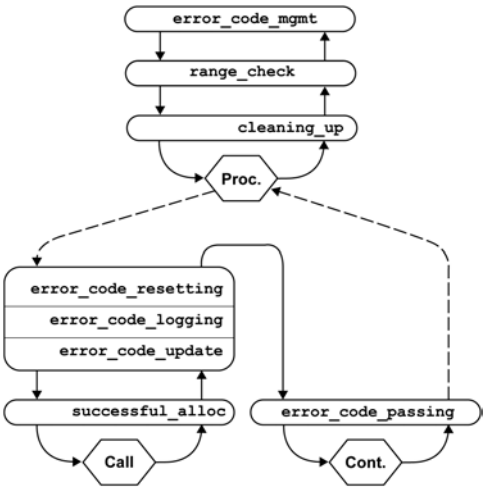

Figure 7. Schematic order of execution of a woven idiomatic procedure. Full arrows denote execution flow, while dashed ones indicate that the sequence in between them can show up zero or more times. Rounded rectangles represent advice, while hexagons indicate execution, call and local continuation join points. The advice name is centered for around-advice, while it is right-aligned for after-advice.

In Aspicere2, the order of before-advice corresponds to the actual run-time order, while after-advice is written down in reverse order[5]. This means that advice `error_code_update` (lines 31–35) is the first to execute after an idiomatic call (see Fig. 7), to record the call's error status into the enclosing procedure's `error_var` property.

Now, annotations will kick in. First, logging of linked errors can only be meaningful if a developer provides the new error value. Hence, the absence of a log annotation could be interpreted as if logging is not necessary. Line 5 in Fig. 3 shows such an annotation, consisting of a name ("log") and some attributes (here only "LINKED_ERROR"). Advice `error_code_logging` (lines 20–29 on Fig. 5) tells us that when an annotated (line 23), idiomatic procedure call (line 22) returns, there will be a linked error log if the `error_var` property signals an error (line 24). Both the error variable and the call's return value need to be updated to the newly provided error value. Logging only happens when needed.

To override the `error_code_passing`-advice for manual recovery (lines 14–18), it suffices to reset the error variable (line 17) when a developer uses a `@manual`-annotation. Indeed, as the error variable is `OK` again, the `error_code_passing`-advice will not notice anything and just proceed (see Fig. 7). The developer should then catch the procedure call's error return value, check it for the

[2] `http://www.eclipse.org/aspectj/`

[3] This is actually a problem we are trying to solve with MAKAO (Makefile Architecture Kernel for Aspect Orientation), a re(verse)-engineering framework for build systems. More info on `http://users.ugent.be/~badams/makao/`.

[4] In Aspicere2, properties, return values and procedure actuals are accessible from within the advice by pointers instead of by their real type. This is why e.g. the `error_var`-property's type is `int*` instead of `int`.

[5] You can easily memorise this, as the first *before*-advice runs *before* all others, while the first *after*-advice executes *after* all others. Analogous rules hold for *around*-advice. This precedence system differs at some points from AspectJ's.

```
1  int f(void){
     int tmp=OK;
3    ...
     /*@manual()*/
5    tmp=g();

7    if(tmp==EASY_TO_FIX_ERROR){
       /* recover manually */
9    }else if(tmp==INITIAL_CLEANUP_ERROR){
       ...
11     rethrow(tmp);
     }
13
     ...
15 }

17 int rethrow(int a){
     return a;
19 }
```

Figure 8. Small example illustrating developer overriding of exception handling.

```
1  /*necessary imports*/

3  int around range_check(int* R,int Arg,
     int LowerBound,int UpperBound) on Jp:
5    idiomatic_proc(Jp)
     && property(Jp,error_var,R)
7    && range(Jp,Arg,LowerBound,UpperBound){
       if((Arg < LowerBound)||(Arg > UpperBound)){
9        *R=PARAM_ERROR;
         LOG(*R,OK);
11       return *R;
       }else{
13       return proceed();
       }
15 }
```

Figure 9. Parameter range checking aspect.

```
1  range(Jp,Arg,LowBound,UpBound):-
     annotation(Jp,range,[ArgName,LowBound,UpBound]),
3    nth_arg(Jp,Arg,ArgName)
     .
```

Figure 10. Accompanying Prolog metadata of aspect in Fig. 9.

```
   /*necessary imports*/
2
   int* intro memory_op_success() on Jp:
4    memory_allocation(Jp,_,_);

6  Type intro memory_op_alloc(TYPE Type) on Jp:
     memory_allocation(Jp,_,Type);
8
   void after successful_alloc(int* SuccessVar,
10  TYPE Type,Type AllocVar,Type Actual)
    returning (int* Code) on Jp:
12   memory_allocation(Jp,Actual,Type)
     && property(Jp,memory_op_success,SuccessVar)
14   && property(Jp,memory_op_alloc,AllocVar){
       if(*Code==OK){
16       *SuccessVar=1;
         *AllocVar=*Actual;
18     }
   }
20
   void after cleaning_up(int* R,int* SuccessVar,
22  TYPE Type,Type AllocVar) on Jp:
     idiomatic_proc(Jp)
24   && enclosingMethod(JpCall,Jp)
     && memory_allocation(JpCall,_,Type)
26   && property(Jp,error_var,R)
     && property(JpCall,memory_op_success,SuccessVar)
28   && property(JpCall,memory_op_alloc,AllocVar){
       if((*R!=OK)&&(*SuccessVar==1)){
30       mem_free(AllocVar);
       }
32 }
```

Figure 11. Memory handling aspect.

proper value he expects and recover from it the way he/she wants. Fig. 8 illustrates this on lines 7–8. So, control flow transfer happens by default, while logging should be explicitly asked for.

What if the developer just wants to do some initial recovery manually, followed by the default exception handling behaviour? This is actually easy to do (lines 9–12): use the `@manual`-annotation and just call the identity procedure (aptly called `rethrow`, see lines 17–19) after the local recovery. This will again trigger the default exception handling on the original error.

Although not necessarily a part of the return-code idiom, we can extract the argument range checking concern as well, which gives us the aspect of Fig. 9 and its accompanying Prolog file (Fig. 10). It only involves looking for `@range`-annotations on procedures (see line 1 of Fig. 3) containing the relevant argument name and the two bounds. The advice body is similar to Fig. 2's dark F-shape. Fig. 10's predicate allows easy access to both annotation info (line 2) as well as the checked argument `Arg` (line 3).

5. Memory cleanup through join point properties and type parameters

Having tackled the error propagation and logging concerns, we will now focus on the cleanup of resources in case of errors, and more in particular of any dynamically allocated memory. The exact problem involves detecting which variables refer to allocated memory (need to be cleaned up) and which ones are just dangling pointers (no cleanup). Thanks to the return-code idiom, catching the memory allocation procedure's return value and checking it when halting from the continuation does the job. Of course, there should be a way to access the allocated memory. For both these purposes, join point properties of the memory allocation procedure calls are a perfect fit.

Fig. 11 shows the resulting memory cleanup aspect. The two mentioned join point properties are declared on lines 3–4 and 6–7. The first one (initialised by default on `OK`) signals whether the corresponding memory allocation is a success, in which case the second one holds a reference to the allocated memory area. The properties are associated with `memory_allocation`s, i.e. calls to `memory_operation`s (lines 4–9 on Fig. 12) of which some examples are given (lines 1–2). For each call, both the second[6] actual `Actual` (line 8) as well as its type `Type` (line 7) are captured. Yes, it is possible that a float is allocated or some user-defined struct. As C lacks interfaces or something similar, and `void`-pointers are not type-safe (although it would work in this case), we apply the original Aspicere's type parameters [9] here. These can be used in advice bodies like C++-templates do, or as an around-advice's return type. They just need to be declared as `TYPE`s in the advice's signature (lines 6, 10 and 22 of Fig. 11) before they can be used.

[6] Indexing in code starts from zero.

```
memory_operation("mem_alloc").
memory_operation("mem_calloc").

memory_allocation(Jp,Actual,Type):-
  invocation(Jp,FName),
  memory_operation(FName),
  nth_arg_type(Jp,1,Type),
  nth_actual(Jp,1,Actual)
  .
```

Figure 12. Accompanying Prolog metadata of aspect in Fig. 11.

Advice `successful_alloc` (lines 9–19) catches the return error value of allocation procedure calls and stores a reference to the allocated memory area, but only if no error occurred. Advice `cleaning_up` (lines 21–32) frees allocated memory after the continuation has been skipped (`error_var`-property differs from `OK`) for each successfully allocated variable (line 29).

Notice on Fig. 7 that memory allocation and cleanup won't take place when a range violation occurs. This is semantically correct, as careful investigation of Fig. 2 learns. On line 13, the `mem_free`-call will always be run to cleanup the allocation of line 3 (which has succeeded when a range violation happens). As this is a redundant transaction (allocate, detect range error and deallocate), our aspect just ignores memory allocation and cleanup in this case. An alternative which adheres better to the idiom would be to advise the last `memory_allocation`'s local continuation join point, provided that all such allocations occur at the beginning of procedures.

What if the call to `mem_free` on line 30 of Fig. 11 goes wrong? Aspicere2 does *not* allow advice on advice, which would let this error go unnoticed. Fortunately, replacing calls to `mem_free` by invocations of a wrapper around it solves this if the wrapper is part of the base code. Indeed, our aspects will target either the wrapper or the `mem_free`-call it contains, providing default exception handling. If some more specialised recovery actions would be needed in this case, one could put them into an additional aspect or just add it directly to the wrapper.

6. Discussion and Future Work

6.1 Benefits

Looking back at all aspects and Prolog files we presented, one could argue whether the original code of Fig. 1 is not much shorter. At first sight, it is. However, keeping in mind that the aspects have to be written only once and will be superimposed throughout the system, they form in fact an initial investment that will pay itself back once a whole module is refactored. All developers need to do is to annotate their code or (in some cases) override default exception handling behaviour. In [1] the code reduction achieved on a representative module of 20 kLOC has been measured. Exception handling accounted for 9% of this (1716 LOC), whereas our aspects (together with their Prolog files) account for 122 LOC. For each logged linked error, there will be a `@log`-annotation. If developers want to recover manually, there will be another extra line of code, i.e. the `@manual`-annotation. More precise numbers can only be obtained when applying our aspects in practice, but all in all our approach seems to allow for a substantial large code reduction. This enhances base code readability and facilitates software evolution.

Of course, before this can happen, the current base code will need a thorough conversion. First, the actual main concern should be recovered from the tangled representation it is held captive in (cf. Fig. 2). Also, the relevant error values should be looked up. Finally, all error handling code should vanish and annotations should be inserted to guide the aspects. This is indeed a major effort, requiring extensive test suites to validate the migration results. Nevertheless, this situation comes as no surprise when migrating from a code base that is dictated by a couple of idioms, especially when they are as invasive as the return-code idiom. The automated approach for concern verification presented in [1] could be applied here. Automatic pointcut and advice construction, however, is unnecessary. Indeed, our major pointcuts are based on "semantic" concepts like:

```
int f(int a, int** b){
  int r = OK;
  bool allocated = FALSE;
  r = mem_alloc(10, (int**) b);
  allocated = (r == OK);
  if((a < 0) || (a > 10))
    ROOT_LOG(PARAM_ERROR,r);
  LINK_LOG(g(a),LINKED_ERROR,r);
  NO_LOG(h(b), r);
  if((r != OK) && allocated)
    mem_free(b);
  return r;
}
```

Figure 13. Macro solution for Fig. 1.

- returning an integer;
- local continuation join points;
- annotations;
- join point properties.

At the same time, exception handling advice is actually very generic by itself, which is enforced by the use of context variables for representing types, annotation attributes, etc. This robustness property can only be proven by applying the aspects on various modules, possibly having differences in idiom interpretation.

The resulting code of Fig. 3 could in fact be the starting point for any exception handling strategy, so the cleanup effort is no waste of time. An aspect-based solution has another benefit: changing exception handling strategies is now a matter of writing and using another aspect. As return values were used to indicate an error status in the original implementation, they are now pretty much useless in the base code. Instead of turning them into `void`, keeping them adds extra possibilities for aspect implementations. This is an extra benefit of section 2.2's restrictions.

6.2 Alternatives

Fig. 13 shows a macro-based solution presented in [2] for Fig. 1's example procedure. Macros `ROOT_LOG`, `LINK_LOG` and `NO_LOG` hide assignments to the error value as well as any needed `LOG`-calls. Cleanup code was not elaborated on in [2] as it was considered as a separate concern. The macros already are a significant improvement as checks are hidden behind them. They still need to be called manually, which is a likely smaller source of problems than the current situation, and there is still an explicit error variable. A practical case is required to see if our aspect approach proves to be feasible and less error prone.

Instead of having to advise local continuation join points, it could also be possible to add a new keyword to the language: `break(some_value)`. Its semantics are analogous to the `break`-keyword used to jump out of loops, but now the current executing base procedure is stopped. More work is needed to really assess this keyword's viability and general applicability.

6.3 Implementation notes

Having learnt a lot from our experiments with the original Aspicere [9], we redesigned the language a bit and reimplemented

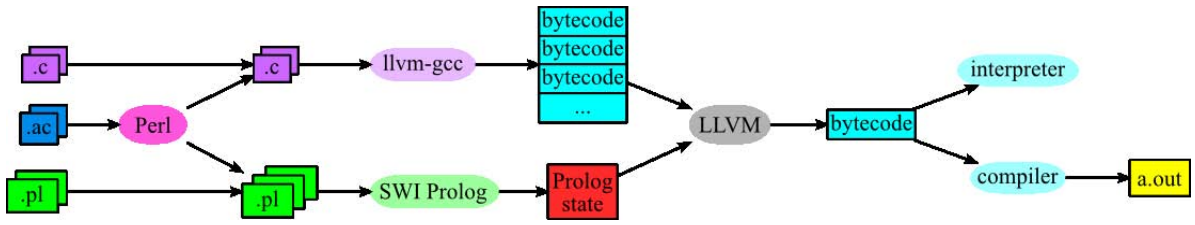

Figure 14. Aspicere2's weaver architecture.

its weaver. Instead of our own XML framework and extended C parser, we opted for LLVM (Low-Level Virtual Machine)[7]. This provides an SSA-form intermediate representation called LLVM bytecode, which is generated from C or C++ by a GCC-based frontend. Weaving boils down to bytecode transformation at link-time, at which time all modules' bytecode has been glued together. This is illustrated on Fig. 14. A reification pass then traverses the link-time module and asserts all interesting program entities as Prolog facts. Next, all pointcuts (transformed into Prolog rules) are queried to find any matching join points. Armed with the resulting set of matches, a weaving pass transforms the IR representation of the full program. Afterwards, optimisations reduce the woven code by inlining, dead code elimination, etc. The end result is a bytecode representation of the whole, woven program that can be interpreted or linked with all required libraries into a normal executable.

We have not applied our aspects in practice yet, except for [2]'s running example where they behave as explained in this paper. Nevertheless, we can already make some qualitative efficiency observations. At build time, compilation and linking is replaced by the process outlined in Fig. 14. At run-time, the aspects also add some extra complexity:

- advice has been transformed into procedures to which various calls are issued;
- the cleanup aspect adds extra variables to idiomatic procedures and contains extra calls to the `succesful_alloc`-advice.

On the other hand:

- join point properties can be mapped onto local variables;
- advice `error_code_passing` can be inlined efficiently, resulting in the same code size as the original base code;
- Fig. 9's aspect actually optimises the current implementation as it avoids unnecessary memory (de-)allocation;
- the bytecode optimisation passes can optimise away lots of redundant code.

Experiments will have to indicate which arguments will prevail.

7. Conclusion

We have shown how a combination of carefully crafted aspects relieves the developers from the return-code idiom administration, unless they really want to do manual recovery themselves. At the heart of our approach lies the concept of local continuation join points, giving aspect developers the power to skip the remainder of a procedure at any (join) point during its execution. Join point properties are used to decouple advices from each other, while annotations allow developers to override the default exception handling scheme. The latter, together with type parameters and the fact that all procedures keep returning an integer error value, results in fairly robust pointcuts and advice. We argued that the aspects considerably improve code readability, understandability and evolvability, by extracting all exception control flow and (in general) reducing code size. Compared to the original code, the woven application should have similar run-time efficiency. Our claims still need to be backed by an actual application to a real-world case.

[7] `http://llvm.org/`

Acknowledgments

The authors want to thank Tom Tourwé for his support and comments on details of the actual problem, and both Serge Demeyer and Yanic Inghelbrecht for feedback on an early draft of the paper.

References

[1] M. Bruntink, A. van Deursen, and T. Tourwé. An initial experiment in reverse engineering aspects. In *WCRE '04: Proceedings of Working Conference on Reverse Engineering*, pages 306–307. IEEE Computer Society, 2004.

[2] M. Bruntink, A. van Deursen, and T. Tourwé. Discovering faults in idiom-based exception handling. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 242–251, New York, NY, USA, 2006. ACM Press.

[3] S. Ducasse, A. Lienhard, and L. Renggli. Seaside — a multiple control flow web application framework. In *ISC '04: Proceedings of 12th International Smalltalk Conference*, pages 231–257, Sept. 2004.

[4] Y. Endoh, H. Masuhara, and A. Yonezawa. Continuation join points. In *FOAL '06: Proceedings of the Foundations of Aspect-Oriented Languages Workshop at AOSD 2006*, pages 1–10, March 2006.

[5] J. Guy Lewis Steele. *RABBIT: A Compiler for SCHEME*. PhD thesis, May 1978.

[6] A. Kelley and I. Pohl. *A book on C (Fourth Edition)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1998.

[7] 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*, pages 418–427, New York, NY, USA, 2000. ACM Press.

[8] I. Nagy, L. Bergmans, and M. Aksit. Composing aspects at shared join points. In A. P. Robert Hirschfeld, Ryszard Kowalczyk and M. Weske, editors, *NODe2005: Proceedings of International Conference Net-ObjectDays*, volume P-69 of *Lecture Notes in Informatics*, Erfurt, Germany, Sep 2005. Gesellschaft für Informatik (GI).

[9] A. Zaidman, S. Demeyer, B. Adams, K. D. Schutter, G. Hoffman, and B. D. Ruyck. Regaining lost knowledge through dynamic analysis and aspect orientation. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 91–102, Washington, DC, USA, 2006. IEEE Computer Society.