Modularizing Error Recovery

Jeeva Paudel[*]
Department of Computer Science
University of Saskatchewan
Canada, S7N 5C9

Christopher Dutchyn[†]
Department of Computer Science
University of Saskatchewan
Canada, S7N 5C9

ABSTRACT

Error recovery is an integral concern in compilers. Improving error recovery requires comprehension of a large and complex code base, in order to locate the places which raise errors and places which handle them. This involves significant amounts of cognitive effort to identify these error-related static locations in the compiler, and the dynamic points in compilation where errors are detected. Essentially, the error recovery concern is a global design issue which is entangled with many other functional concerns, and whose implementation is frequently scattered across different program elements. Current compiler implementations often do not explictly identify error-related control dependencies and fail to separately characterize the actions to take in the event of errors.

In the context of the AspectJ compiler (ajc), we modularize error concerns as join points-and-advice aspects which provide improved modularity by explicitly declaring the loci of error detection, and providing extension points as advice to handle these errors. We apply this modularization to support a diverse set of examples of error recovery. As a result, the compiler writer no longer needs to navigate and understand the entire compiler source in order to replace or extend error recovery actions.

1. INTRODUCTION

Conceptually, compilers consist of multiple phases, each of which can encounter errors. Although errors are commonplace and form a global design issue, very few languages have been designed with error handling in mind[1, 15]. Planning error handling and recovery right from early stages of compiler development can both simplify their structure and improve their response to errors in the program to be compiled. A good error handler should:

- allow compiler designers to quickly identify the source, and associated context of errors,
- be amenable to improvements,
- not signficantly slow down the processing of correct programs, and
- try to recover from simple errors whenever possible.

[*]jep924@mail.usask.ca
[†]cjd032@mail.usask.ca

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

Effective realization of these goals, however, is challenging for two major reasons. First, the error concern is tightly coupled to different phases of a compiler, and is interleaved with the underlying program logic in a complex manner. Take for instance, semantic analysis and intermediate code generation phases. Semantic analysis phase synthesizes and maintains type and value environments as part of type-checking and symbol table loading. Similarly, it also computes the escape values as part of escape analysis. These attributes will be inherited by the intermediate code generation phase for managing static links and generating intermediate representation trees. Such an intricate relationship between these phases engenders arcane control dependencies, which cannot be described by typical responsibility boundaries. Thus, it affects several design choices of compilers in subtle but pervasive ways. Second, error-related control dependencies are not explicitly defined and localized into distinct modular units.

As a result, error handling is further complicated by the fact that it is not implemented in separate dimension. Although it is a global design issue and cuts across different units of functionality, it is still being implemented in traditional hierarchical fashion, which is incapable of modularizing the cross-cutting concerns.

In this paper, we aim to simplify error handling in compilers. In order to meet this goal, we modularize the error handlers discovered in different phases of a compiler into a distinct unit. This facilitates modification and improvement of error reporting and patching processes in the subsequent error recovery tasks.

In order to demonstrate our modularization endeavor, we have chosen ajc. Although it is an aspect-oriented compiler, it does not incorporate aspects to modularize the error handlers within itself. Hence, there are plenty of opportunities for aspectization of error handling concerns. Further, owing to its support for weaving from source code and compiled code, often times the compiler has to vouch for correctness of a single functionality at different levels. Hence, the need and opportunity for handling the same error at different levels in ajc provides a fair level of complexity in modularization of error concerns, over other mainstream compilers.

In light of these problems of non-modular and hidden

structure of error handling in compilers, the research contributions of this paper are:

1. Identification of points in the static program structure and dynamic execution graph of the compiler that are responsible for error handling.
2. Diverse set of examples for modularization of distinct kinds of scattered errors.
3. Explicit extension points for error recovery
4. Assessment of the modified compiler in terms of
   - modularity,
   - performance, and
   - correctness.

To achieve these goals, we leverage the abstraction and modularization capabilities of Aspect Oriented Programming-(AOP)[10]. Particularly, we rely on its *join point model*(JPM) to render more flexible the association between error handling code and normal application code.

We begin by reviewing some important details about error handling. Next, we provide a brief overview of the scattered and tangled nature of existing error handlers in the compiler chosen for this study. Then, we present examples of our modularization approach. This is followed by demonstration of subsequent changes in the modular units to accommodate our evolving needs. Finally, we close with a discussion of several issues that arise during error modularization, and a description of future work.

2. ERROR RECOVERY

Most programming language specifications do not describe how a compiler should respond to errors; the decision is left to compiler designers[1]. As a result, naive compilers are designed in a simple manner to report failure and stop further computation in the event of errors. But, this behavior would be unkind to programmers. A reasonable approach from the user perspective would be a meaningful error report or, even some kind of adjustment so that the compilation could proceed. Most compilers attempt to adjust some inputs or values at the point of error in a way that allows compilation to proceed further, even if not until the end of the program. This process is called *error recovery*.

Error recovery is a four-step process, which consists of: detection, diagnosis, reporting and patching[4]. Current implementation techniques demand a significant amount of time and cognitive effort in identifying the source, context and nature of errors for reporting and repairing them. Even worse is the fact that this process has to be repeated every time a new error reporting or patching facility is to be tested or introduced.

Our design goal here is to identify and explicitly define such error handling artifacts – sites, contexts and control flows. By sites, we mean the locations in the program source code where errors need to be detected and handled. By contexts, we mean the calling or executing types, values used or synthesized at error sites, and formals and return types of the methods raising errors. Control flow here refers to the execution path of the program.

Modularization of such artifacts increases the value of the system in terms of available options for better comprehension, re-usability, extension points for error recovery and assesement of new error recovery strategies.

```
1   public ResolvedMember[] getDeclaredFields() {
      raiseCantFindType(WeaverMessages.
3         CANT_FIND_TYPE_FIELDS);
      return NO_MEMBERS;
5   }

7   public ResolvedMember[] getDeclaredMethods() {
      raiseCantFindType(WeaverMessages.
9         CANT_FIND_TYPE_METHODS);
      return NO_MEMBERS;
11  }
     .............
13   .............
     .............
15  }

17  public int getModifiers() {
      raiseCantFindType(WeaverMessages.
19        CANT_FIND_TYPE_MODIFIERS);
      return 0;}
```

Listing 1: Error Reporting related to MissingTypes

2.1 Error Recovery in *ajc*

AspectJ is an aspect-oriented extension to Java. In addition to normal front-end and back-end components of a compiler, it also contains a matcher (for name patterns) and a weaver (both for intertype declarations and for advice). This study mostly targets the weaver. The weaver alone generates more than 150 different error messages. References to these errors, and their handlers are scattered across different units of code such as: methods, classes, and packages. Examples of errors at the weaver end include those related to parsing, argument binding, type resolution, missing types, compilation environment, and weaver states.

Consider Listing 1, which shows a part of code in MissingResolvedTypeWithKnownSignatures class. While attempting to resolve a required type in the World[1], if the weaver fails to find any dependent type, it returns an instance of this class. This class defers the production of the "*can not find type*" error until some code requires such information which cannot be determined from the type Signature alone. This enables the weaver to be more tolerant to missing types and thus, delay the compilation failure to a certain extent.

This class has facility to report errors upon attempts to access various missing types- fields, methods, interfaces, pointcuts, superclasses and modifiers. Although, they are neatly modularized in a single class, this implementation has four major problems:

- First, there is clear tangling of two different concerns here: *functional requirement* and *error handling*. The functional requirement here is that these methods should return some default values of appropriate types. The error concern however is that, an error should be reported upon any attempt to access non-existent attributes of MissingResolvedTypeWithKnown-Signatures type. Such tangling inhibits possibilities of creating and re-using abstractions of error handling and functional operation. Consider cases, where we would like only error reporting and cases where we would like these methods to return some default values without reporting errors. As the implementation stands now,

[1] ajc collects all members that have an invasive effect outside their own compilation unit into a World before any weaving can take place.

```
public abstract privileged aspect MissingAndIncompatibleTypes {
  protected abstract pointcut currentType(MissingResolvedTypeWithKnownSignature aType);
  protected abstract pointcut contextForMissingTypes();
  protected abstract pointcut contextForIncorrectTypeAssignability(ResolvedType otherType);
  pointcut missingResolvedTypes(MissingResolvedTypeWithKnownSignature aType):
    currentType(aType)
    && contextForMissingTypes();
  pointcut incompatibleTypeAssignability(MissingResolvedTypeWithKnownSignature aType,
      ResolvedType otherType):
    currentType(aType)
    && contextForIncorrectTypeAssignability(otherType);}
```

Listing 2: Abstract Aspect for reporting Missing Resolved Types

```
privileged aspect MissingResolvedTypeErrorReporter extends MissingAndIncompatibleTypes {
  protected  pointcut contextForMissingTypes():
    execution(public ResolvedMember[] *.getDeclaredFields())
        ||execution(public ResolvedMember[] *.getDeclaredMethods())
        ||execution(public ResolvedType[] *.getDeclaredInterfaces())
        ||execution(public ResolvedMember[] *.getDeclaredPointcuts())
        ||execution(public ResolvedType *.getSuperclass())
        ||execution(public int *.getModifiers())
        ||execution(public boolean *.hasAnnotation(UnresolvedType));

  protected  pointcut contextForIncorrectTypeAssignability(ResolvedType otherType);

  protected pointcut currentType(MissingResolvedTypeWithKnownSignature aType):
    this(aType);
}
```

Listing 3: Concrete Aspect to report Missing Resolved Types

this is not possible.

- Second, it lacks clarity of program execution path in which the errors are reported. A careful examination of the above implementation reveals that the error reports are generated before these methods return values. However, this is not visibile outright from the current implementation. This would be more cumbersome when error handlers are interspersed across different methods in different classes. This because the loci of such handlers are not explicitly identified and defined in the program source.

- Third, it lacks a principled way to remember the location of these error reporters[2]. Next time, if a developer wants to extend existing error reporting, he would need to inspect the code again to find them. Thus, it lacks a facility to properly define and localize the context in which these reports should be generated. Further, by looking at this implementation, it is difficult to convince if these are the only methods that generate raiseCantFindType(..) error, or there are other methods too.

- Fourth, this implementation is brittle to changes. For error diagnosis or handling, we might to need to carry out data-oriented or conrol-oriented changes or both[11]. Design patterns [7] such as *Subject-Observer* and *Visitor* will suffice for a decently modular implementation, if any one of these changes is required. In cases where both of them are required, it is quite difficult to do so without code repetition and tangling using traditional programming paradigms. Among different proposals such as DemeterJ[11] and AOP proposed to solve this problem, we will use AOP in this paper.

[2]We are ignoring the facility of axnnotations, because that too requires inspection of a large body of code.

```
public ResolvedMember[] getDeclaredFields() {
  return NO_MEMBERS;
  }
public ResolvedMember[] getDeclaredMethods() {
  return NO_MEMBERS;
  }
       .............
       .............
       .............

public int getModifiers() {
  return 0;
  }
```

Listing 4: MissingTypeWithKnownSignature class after extraction of error concern

These are the problems we are trying to address by modularization of error concern.

3. DESIGN OF MODULAR ERROR HANDLERS

Before further discussions, we provide a brief overview of the AOP [8] concepts: *JPM, inter-type declaration, and aspect*. The JPM defines the structure of dynamic cross-cutting[3] concerns and underlies the concepts of:

1. **join points** – these are points in program execution or static locations in the source code, including access to the control flow contexts (cflow).

[3]By cross-cutting, we mean the concepts that are scattered across traditional units of modularization, such as: methods, classes, modules and packages

```
1  before(MissingResolvedTypWithKnownSignature aType):
       missingResolvedTypes(aType){
3
       String typeName = thisJoinPointStaticPart.
                    getSignature().getName();
5
       String weaverMsg = WeaverMessages.
                    CANT_FIND_TYPE;
7
       if (typeName.endsWith("Methods"))
9        weaverMsg = WeaverMessages.
                    CANT_FIND_TYPE_METHODS;
11     else if (typeName.endsWith("Fields"))
         weaverMsg = WeaverMessages.
13                  CANT_FIND_TYPE_FIELDS;
       ................
15     ................
       ................
17
       else if (typeName.endsWith("Modifiers"))
19       weaverMsg = WeaverMessages.
                    CANT_FIND_TYPE_MODIFIERS;
21
       aType.raiseCantFindType(weaverMsg);}
```

Listing 5: Improved advice for more informative reporting of Missing Types

2. **pointcuts** – they provide a means of identifying join points.

3. **advice** – it is a means of affecting the semantics at the join points, by changing behavior before, after, or around the join points.

The other concept adopted by AOP is inter-type declarations. They provide a way of instrumenting the static behavior of classes by supporting the notion of open classes. Aspect is a module encapsulating these AOP constructs.

We now have sufficient information to describe the general design for modularizing error recovery to support better comprehension, decreased redundancy, and increased re-use opportunities for recovery. In short, we identify the points in program execution where error checks should happen, and explicitly move them into separate modular units, along with the actions to be taken in case of such errors. Here, we do not propose any efficient error recovery schemes, but only try to localize and ecapsulate such error-related concerns into clean modular units. Hence, existing error recovery in ajc[4] is not changed, save for the example implementations depicting the usability of our design.

To locate declarations and references of error handlers, we mainly followed two approaches to code inspection. We used development aspects to identify *the error print streams, loggers, exception throwing points* and *handlers*. To identify other customized error reporters, we had to manually inspect the code. A major finding of this was that the compiler is riddled with error handlers, and it involved significant effort to locate them. So, here we try to reduce the economic burden associated with identifying such join points, by encapsulating them in dedicated responsibility boundaries.

We have implemented a set of aspects in ajc to modularize error handlers pertaining to:

- parsing of aspectj constructs such as type patterns, pointcuts, advices and inter-type declarations
- creation of initialization and pre-initialization shadows for type mungers
- type resolution and
- enforcement of type munging rules.

Now, we will look at three of these examples, and their subsequent re-use and improvement for enhanced reporting and recovery. Two of these are semantic errors and the remaining one is syntactic error.

3.1 Modularizing MissingType Error Handler

This example is about missing types, first introduced in section 2.1.

Lets begin by extracting the error handlers related to missing types from Listing 1. Using AspectJ, we extract and modularize handlers related to missing type error into aspects as shown in Listing 2 and 3. After separation of error handlers, the refactored class now looks as shown in Listing 4. Here, we will assess the benefits of such modularization.

The advice shown in Listing 5 is part of an abstract aspect MissingAndIncompatibleTypes, and defines the action to take in the event of missingResolvedTypes. An abstract aspect helps us define a set of events that is left undefined, but that can be advised. Then in the concrete aspect MissingResolvedTypeErrorReporter, shown in Listing 3, we bind the methods to the specific events to which the handler should be hooked. This way, we are separating location (points in static program and dynamic execution graph) where errors might occur from the actions that should be taken upon their occurrence. Further, by looking at the context associated with the advices, we can infer end-to-end data flow related to error handlers. In our examples, it is easy to spot that "*missing type*" errors are raised only by instances of MissingResolvedTypeWithKnownSignature types. Yet another benefit of this modularization is flexibility in accommodating design decisions. For instance, we decide to change the design decision to report errors only after returning default values in response to the getters in Listing 1. We can easily model this change by changing the kind of advice to after.

At present, our error reporter is very naive in that it says nothing other than *"missing type"* error. If we wish to inform precisely the kind of missing type, this change needs to occur only in the advice body as shown in Listing 5. Note that by decoupling error concern from the functional code, we can make changes to each of them in relative isolation.

3.2 Modularizing Parsing Error Handler

Our second example is about errors while parsing. Specifically, here we will look at errors raised as a result of ill-formed type patterns and pointcut expressions.

In order to modularize error handlers related to parsing, we first identify the join points that lead to ParserExceptions, and define them as pointcuts. Then, as part of advice implementation, we surround computation under these join points with try-catch blocks. The catch block is engineered to perform appropriate error reporting whenever ParserException is raised. First consider handling errors when parsing a given type pattern. Listing 6 shows pointcut specification for capturing join points which might raise ParserException when parsing type patterns. Likewise Listing 7 depicts pointcuts for capturing join points that might raise ParserException when parsing Pointcuts. From these pointcut definitions, it

[4]Although the design decisions were conceived with AspectJ in mind, they are equally relevant to and applicable to other aspect oriented compilers with support for the join point model, inter-type declarations, and aspects.

```
/* Capture Invalid Type Pattern */
pointcut captureInvalidTypePattern(String patternString, AjAttributeStruct location):
  cflow(execution(private static TypePattern parseTypePattern(String, AjAttributeStruct))
    &&args(patternString, location))
  && (call(public TypePattern PatternParser.parseTypePattern())
    ||call(public void *.setLocation(ISourceContext, int, int)))
  && withincode(private static TypePattern parseTypePattern(String, AjAttributeStruct));
```

Listing 6: Point to capture invalid type pattern while parsing AspectJ attributes

```
/* Capture Invalid Pointcut */
pointcut captureInvalidPointcut(String pointcutString, AjAttributeStruct location):
  cflow(execution(private static Pointcut parsePointcut(String, AjAttributeStruct, boolean))
&&args(pointcutString, location, *))
  && (call(public Pointcut PatternParser.parsePointcut())
    ||call(public void *.setLocation(ISourceContext, int, int)))
  && withincode(private static Pointcut parsePointcut(String, AjAttributeStruct, boolean));
```

Listing 7: Point to capture invalid pointcut while parsing AspectJ attributes

is clear that *parsing* error occurs occur in the control flow of top-level or recursive excecution of parseTypePattern or parsePointcut methods. Since the behavior to instrument at these join points is quite similar, we can apply a single advice to report errors at these sites. The advice is shown in Listing 8.

Besides these, the compiler has to deal with several other errors that occur during parsing, such as: when parsing perclauses, annotation pointcuts, annotation aspects and other aspectj constructs. Such closely related pointcuts are good candidates to be localized into a single aspect that handles parsing errors across different aspectj attributes. Further, composing them provides an opportunity to handle errors through a single advice.

3.3 Modularizing InCorrectReturnType Error Handler

This example provides recovery from errors that occur when new types violate type munging rules. Specifically, when an existing type hiearchy is changed or new one is introduced, the weaver has to ensure that these types do not violate the pre-defined type-munging rules. Here, we will look at error handling related to method overriding in sub-types.

Any attempt to weaken the post condition of an overriding method in a derived class by broadening the range of return types violates *Liskov Substitution Principle*[13]. Accordingly, ajc raises an error under such a circumstance. Here, we examine the error handler that provides this behavior resulting from inter-type declarations. Since AspectJ supports weaving from source code and pre-compiled code, the aforementioned error has to be dealt with at both of these levels. For brevity, we do not present the current implementation of this error handler here.

We have identified and localized the artifacts related to this error in aspects. The pointcuts and associated advices are shown in Listing 9 and 10. From these pointcuts, one could easily find that InCorrectReturn errors are raised in two situations. First, in the control flow of addition of inter-type mungers as indicated by cflow pointcut cflow(execution(public void ResolvedType.addInterTypeMunger-(ConcreteMunger)) in Listing 10. The exact point where this happens is while checking the type of overriding method (resulting from new type munger to be added) as indicated by execution(public boolean Resolved.checkLegalOverride(...)). Further, we also have access to the context of this error through args(..) pointcut. Second situation in which InCorrectReturn is handled is while enforcing type compatibility rules as part of type munging rules for declare parents as indicated by execution (private boolean BcelMunger.enforceDecpRule4_compatibleReturn(..)).

These pointcuts give us a clear idea of the control flow of the program, where these errors should be handled. We can improve error handling capability of the compiler by trying to recover from this error. A simple recovery scheme is to change the return type of the overriding method so that it same as that of the overridden method. In doing so, we need to keep all other properties of the overriding method intact. We have already modularized the related control dependencies, identified the associated context and action to trigger in the event of occurrence of this error. Hence, we can easily reuse this information while implementing the error recovery action. This concept of re-usability of pointcuts is more visible in Figure 1. $ERAspect_n$ are aspects where pointcuts are defined as part of error-handler modularization. Essentially, they intercept methods raising errors (such as red-colored node with label **C** in the figure) to generate error reports and make a safe exit from the program. If it is an exception condition, these aspects propagate back along the path which led to this method execution and do proper exception handling, as indicated by red-dashed arrows from nodes **C** to **B** to **A** and then to **Error**.

As part of error recovery, aspects $EHAspect_n$ will re-use this pointcuts defined in $ERAspect_n$. At this point, if there is need for additional contexts, besides those captured with args() pointcut, we always have reflective access to them through thisJoinPoint. These aspects will also instrument the methods leading to error, and try to change the context at the point of error so that compilation could proceed further. This is shown by green dashed-arrows in Figure 1. For example: in case of binary weaving, the only change that error recovery involves is replacement of error reporter from the advice with a call to proceed. Essentially, we continue current computation with corrected subMethod, with all other formals remaining the same. So, the only additional work in this case is creating a corrected subMethod

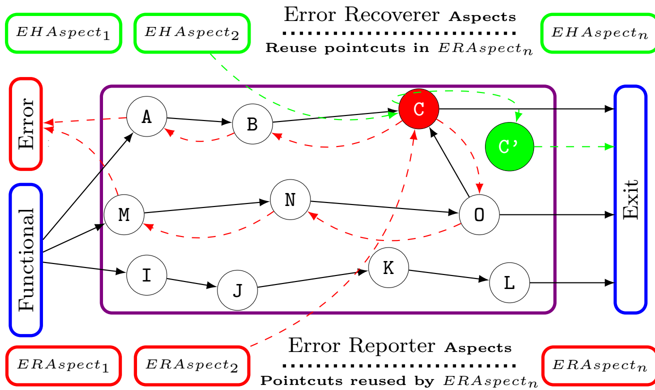

Figure 1: Separation of Concerns and Reusability of Code: Artifacts of Modular Error Handling

from the old one by using supMethod. The advice will now look like one shown in Listing 11.

Similar is error recovery implementation in case of join points matching checkMungerTobeAdded(..) pointcut. Here, the advice incorporates execution of addInterTypeMunger(ConcreteMunger munger) with corrected munger instead of the old faulty munger. A point to note here is that: creating new munger with corrected return type is an expensive operation. It is because we must discard some intermediate states obtained after execution of addMunger(..) method and create them all over again. This involves creating new data structures, traversals over existing ones, and resolving types in the world again.

The recovery algorithm implementation for the remaining pointcut is also similar to this. Hence, we skip discussion of the same. If any new errors stem from recovery attempts, we issue a warning message so as to apprise users of the attempted error recovery.

4. EVALUATION

As we contemplate modular error recovery, the first question that arises is the cost and benefit of the idea. In compilers, this means that we have to vouch for correctness and peformance guarantees of our modularization. We begin this section with a description of how this is done. Next, we evaluate our idea from modularity perspective to examine its re-usability, potential benefits, and any issues that might occur in the process of modularization.

4.1 Correctness Assessment

A fundamental property of compilers is correctness. In order to verify this property of our candidate compiler, we made sure that it passed all the existing JUnit tests, and the new ones that tested for correctness of error recovery.

4.2 Performance Assessment

For performance assessment, we compared elapsed time to complete the JUnit tests in three different platforms. First, is the Mac OS 10.5.6 running on an Intel core 2 Duo MacBook 2.1 with 3 GB memory, 4MB L2 cache, 2.16 GHz processor speed and 667MHz bus speed. The second is Windows XP operating system running on Sony Intel Pentium dual core T3200 Laptop with 2GB memory, 1MB L2 cache, 2GHz processor speed and 667 MHz system bus speed. The third is the Mandriva Linux OS running on an Intel Core 2 CPU with 2.4 GB memory and 4 MB cache and 2.4 GHz Processor Speed. Table 4.2 shows the mean results of ten different readings, along with standard deviations of the results, for both the original and the restructured ajc. We found that modularization of error handlers with aspects resulted in 0.5% increase in compilation time of the JUnit test. This small differences between elapsed time for completion of JUnit tests in the original compiler and error-modularized compiler demonstrate that time overhead of aspectization of error handlers is minimal, and can be safely ignored. Note that the major source of this performance overhead is attributable to expensive operations involved in recovering from the *"incorrect return type"* error from an inter-type declared method.

Platform	Compiler	Time (sec)	σ
Mac OS X	Original	966.4	9.23
	Restructured	971.6	9.56
Ubuntu	Original	959.9	8.11
	Restructured	964.2	8.05
Windows XP	Original	973.2	9.77
	Restructured	977.9	10.34

Table 1: Performance Assessment

Package org.aspectj	LOC in Implementation *OO*	*AO*		Concerns modularized
weaver	13548	13159	195 in aspects	Error Detection & Reporting
bcel	16005	15702	677 in aspects	Detection, Repo--rting & Recovery
Total	29553	29733		

Table 2: Assessing impacts of Modularization of Error Reporting and Recovery using LOC metric

4.3 Modularity Assessment

Figure 2 shows the nature of error handling at the weaver end of ajc. In the figure, the lines in red represent code relating to error handling and are proportionate to the size of the classes represented by white blocks. Before modularization, the error handlers were scattered across 37 different classes, and also methods within them, as indicated by dispersed red-lines within a single white block.

After our modularization endeavor, we were able to bring

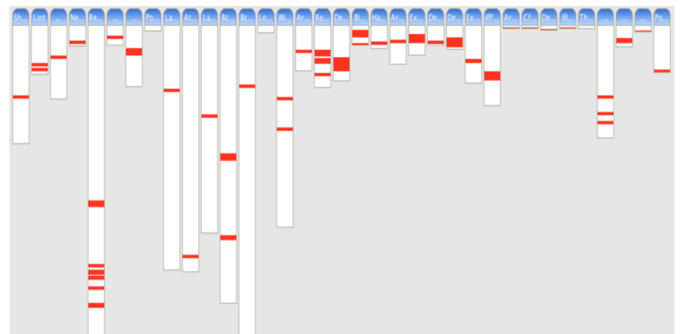

Figure 2: Scattering of error handlers before modularization

```
/* Report error on capturing invalid Type Pattern or Pointcut */
Object around(String patternOrPointcutString, AjAttributeStruct location):
        /* Pointcuts where similar behavior has to be implemented are composed together */
    captureInvalidTypePattern(patternOrPointcutString, location)
  ||captureInvalidPointcut(patternOrPointcutString, location) {
  try{
    return proceed(patternOrPointcutString, location);
  } catch (ParserException e){
    AtAjAttributes.reportError("Invalid "
        + thisJoinPointStaticPart.getSignature().getName().replace("", "parse")
        + patternOrPointcutString + "' : " + e.toString()
        + (e.getLocation() == null ? "" : " at position "
        + e.getLocation().getStart()), location);
    return null;
  }
}
```

Listing 8: Advice to report error on finding invalid type pattern and pointcut while parsing AspectJ attributes

```
/* Capture incorrect return type error during binary weaving of declare parents */
pointcut checkCompatibilityOfReturnTypes(BcelClassWeaver weaver, ResolvedMember superMethod,
    LazyMethodGen subMethod):
    execution(private boolean BcelTypeMunger.enforceDecpRule4_compatibleReturnTypes(BcelClassWeaver,
    ResolvedMember, LazyMethodGen))
    && args(weaver, superMethod, subMethod);
Object around(BcelClassWeaver weaver, ResolvedMember superMethod, LazyMethodGen subMethod):
  checkCompatibilityOfReturnTypes(weaver, superMethod, subMethod){
  if (!superMethod.getGenericReturnType().getSignature().replace('.', '/').equals(
      subMethod.getGenericReturnTypeSignature().replace('.', '/'))) {
    ResolvedType subType = weaver.getWorld().resolve(subMethod.getReturnType());
    ResolvedType superType = weaver.getWorld().resolve(superMethod.getReturnType());

    if (!superType.isAssignableFrom(subType)) {
      weaver.getWorld().getMessageHandler().handleMessage(
          MessageUtil.error("The return type is incompatible with "
              + superMethod.getDeclaringType() + "." + superMethod.getName()
              + superMethod.getParameterSignature(), subMethod
              .getSourceLocation()));
    }
  }
  return proceed(weaver, superMethod, subMethod);
}
```

Listing 9: Pointcut to capture overriding methods with incorrect return types, during binary weaving

```
/* Capture incorrect return type error while weaving from source */
pointcut checkMungerTobeAdded(ResolvedType resType, ResolvedMember parent, ResolvedMember child):
          cflow(execution(public void ResolvedType.addInterTypeMunger(ConcreteTypeMunger))
            && this(resType))
          && (execution(public boolean ResolvedType.checkLegalOverride(ResolvedMember, ResolvedMember))
              && args(parent, child))
          && if(!Modifier.isFinal(parent.getModifiers()));

pointcut checkConflictWithExistingTypes():
    cflow(execution(private boolean ResolvedType.compareToExistingMembers(ConcreteTypeMunger,Iterator))
          && this(resType))
          && execution(public boolean ResolvedType.checkLegalOverride(ResolvedMember, ResolvedMember));

Object around(ResolvedType resType, ResolvedMember parent, ResolvedMember child):
checkMungerTobeAdded(resType, parent, child){
  if (!(resType.world.isInJava5Mode() && parent.getKind() == Member.METHOD)) {
    if (!parent.getReturnType().equals(child.getReturnType())) {
      resType.world.showMessage(IMessage.ERROR, WeaverMessages.format(
          WeaverMessages.ITD_RETURN_TYPE_MISMATCH, parent, child), child
          .getSourceLocation(), parent.getSourceLocation());
    }
  }
  return proceed(resType, parent, child);
}
```

Listing 10: Pointcuts to capture inconsistent method overriding and advice to report this error while weaving from source

```
Object around(ResolvedType resType, ResolvedMember parent, ResolvedMember child):
  checkMungerTobeAdded(resType, parent, child){
    if (!(resType.world.isInJava5Mode() && parent.getKind() == Member.METHOD)) {
      if (!parent.getReturnType().equals(child.getReturnType())) {
        ResolvedMember newChild = correctChild(parent, child);
        return proceed(resType, parent, newChild);
      }
    }
    else return proceed(resType, parent, child);
  }
```

Listing 11: Advice to recover from inconsistent method overriding, at binary weaving level

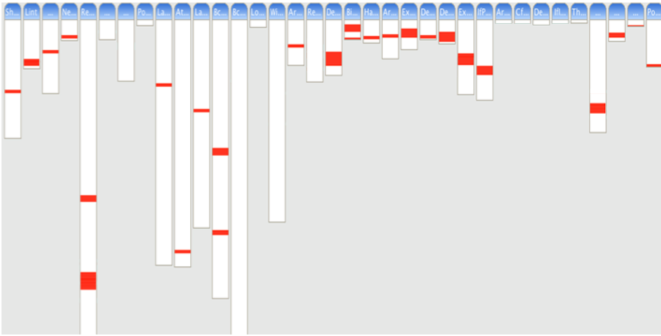

Figure 3: Error handling situation after modularization

down the scattering of error handlers down to 26 classes. Further, we were able to decrease scattering among different methods within a class. This is visible from more condensed red-lines within classes, as shown in Figure 3.

At this point, it was difficult to extract error handler from all classes into aspects. This was primarily because the error handlers were too much dependent upon the local context within methods and classes, and it was not possible to modularize them with aspects alone.

4.4 Benefits of Modular Error Handling

Based upon our modularization endeavor in ajc, we provide a list of benefits of separating error handling from the program and implementing it as a separate pluggable construct. They are:

1. **Quickened and simplified error handler detection**: The join point model serves to define the locations and control flows associated with error handlers. This makes it easy to identify error recognition loci, because they are now well defined in their dedicated place: the pointcuts.

2. **Increased comprehensibility of error handlers**: In addition to control dependencies, pointcuts also expose the data dependencies associated with the error handlers. From the list of formals and return values in the pointcut definition, one can easily infer the data dependencies of error handlers. This serves to improve their comprehensibility.

3. **Increased re-usability**: From examples presented in section 4, it is noticeable that the error recoverer aspects re-use the pointcuts that were used for error reporting. Correct error recovery schemes need to know the precise sites of error occurrence, handling, the type of exception being handled and the control flow of its occurrence. All such information is encapsulated by error reporting aspects ($ERAspect_n$). Later, the error recovery aspects $EHAspect_n$) re-use these pointcuts to garner such information. Then, they replace the erroneous node in the control flow path with a corrected one. Other approaches to error recovery might involve going back in the control graph and continuing execution with changed contexts. This will then allow computation to proceed further, as indicated by green-dashed arrows in the Figure 1.

4. **Reduced Dependencies**: In its present state, there is tight coupling between several classes in ajc because of dependencies relating to error handlers. This is shown by means of dependency structure matrix (DSM) in Figure 4. Here, we consider the dependencies arising only due to error concern. With the modularization we have envisioned, the dependencies among classes (represented by $C_{1...n}$) resulting from error concern will now look like one shown in Figure 5. After aspect-oriented modularization of error handlers, the dependency picture changes dramatically. Dependencies among classes owing to error concern is now completely removed. That means the classes will no more depend on the error concerns in other classes. Further, the dependency direction is now changed such that aspects (represented by $A_{1...n}$) responsible for error handling will now depend upon the existence of these classes to collect required contexts. This new dependency is visible in the lower part of the DSM in Figure 5.

5. **Cleaner separation of concerns**: Separation of error behavior of classes from normal functional behavior makes it easier to understand what those two behaviors really are, because their implementations do not collide in the same text. Additionally, as shown in Figure 1, the control flows related to error handling are separated from other functional control flows. With traditional OO implementations, one would need to propagate back along the control flow path in which the error was raised in order to notify the enclosing types about the error. In Figure 1, if the red-node **C** is a method that raises an error, then as indicated by red dashed-arrows, the compiler would have to propagate back along the path of its control flow to apprise the enclosing type about the error. This again leads to contaminating the functional control with error control flow. However, with aspect oriented mod-

Figure 4: Dependency among classes because of error handlers

Figure 5: Dependency among classes after modularization of error handlers with aspects

ular implementations of error handlers, this would not be necessary. As the control in which the error occured is already precisely captured by means of pointcuts, one can use this information for appropriate error handling. So, this prevents tangling of error and functional control flows.

6. **Increased flexibility in design decisions regarding errors**: Design decisions such as when, where and how to do error handling are easily modifiable. This benefit comes from the expressive power of the JPM.

These benefits, however, come at the expense of tiny performance benefits, which are attributable to byte-code weaving and error recovery computations. Overall, modular error handling engenders key benefits at the cost of modest performance overhead.

4.5 Issues in Modularizing Error Recovery

Besides performance and correctness, modularizing error recovery has to contend with a number of serious difficulties which may not be visible outright. Some of them are:

1. Masking of error handling advices by foreign aspects: It is possible that an unbounded foreign aspect completely replaces our desired computation by introducing new advice around adviceexecutions related to error handlers.
2. Error handling at undesired phases: AspectJ does not provide any explicit means of controlling order of application of multiple aspects at the same join point. Thus, any unintended order of aspect composition may affect error handling behavior in ways unanticipated by the developers.
3. Side effects of error handlers: Unbounded pointcuts may result in advice weaving at unintended points. However, strictly bounded pointcuts may prove too restrictive at times [14]. Hence, it is difficult to correctly identify the circumstances in which foreign aspects might need bounded and unbounded pointcuts.

Although we have identified few such issues, there certainly remain other issues that might afffect modularization of errors. Their identification and implementation remains as a part of the future work.

5. RELATED AND FUTURE WORK

Others have attempted to modularize error handling concern. Lippert and Lopes[12] carried out a study to assess the suitability of aspects for separation of exception handling code from normal application code. Likewise, Adrian et al.[3] carried out a similar study to separate exception handlers from middleware. Lippert and Lopes's study was later complemented by Castor et al.[6] by means of aspect-oriented refactoring of error handling in different real-world applications. Their study concluded that although aspectization of exception handlers created increased opportunities for reuse, and decreased interference in the program texts; they are not always sufficient to encapsulate all kinds of exceptions. Further, they concluded that it is difficult to re-use the exception-related artifacts in practice.

Another study that investigates exception handling in an aspect-oriented setting is by Karin et al [9]. This work primarily deals with restoring the safe state of objects, after a method throws an exception and terminates abruptly.

None of these studies have however, identified the potential for artifact and knowledge re-use for error recovery that comes from carefully implemented modular error handlers. This paper extends the findings and conclusions of the aforementioned papers to modularize error recovery.

Castor Filho et al.[5] have carried out an interesting study of the interplay between aspects and exceptions. They provide a catalog of exception handling scenarios and use it to guide aspectization of exception handling. This paper does not describe how the implementation strategies and hence, their guidelines would change in situations where developers wanted to capture the context of the exceptions too. A future project might be to create a catalog of exceptions along with the control flow in which they occur, the context they would use and delimit the interface between application code and error handling aspects. This study would then be more useful and complete in its purpose as a guide to desirable and undesirable aspectizations relating to error concern.

Another recent development in this field is *EJFlow*[2]. There, the authors extend the AspectJ language with additional syntax to track the flow of exceptions in a program and provide global views of exception control flows. They propose a construct called "an explicit exception channel" that captures the location of exception raisers, handlers and other intermediate sites, type of exception, and a list of exceptions that must be signaled to the enclosing context. Es-

sentially, this is a bunch of abstract pointcuts composed together, which the users could later concretize to define their exceptions and pertaining contexts. The major limitation of this tool is that: it lacks the power of *cflow* pointcuts. For instance, consider an exception that can occur within a single static location. Depending upon the control flow and context associated with this exception, the way errors are reported and handled could be totally different. This applies to those exceptions as well, which are raised from the same static locations in the program code. Remember the ways in which error reporting and later recovery differed for different levels of weaving in ajc, even if the error was same in both cases and was raised from the same location. EJFlow fails to handle such contexts associated with exceptions, in its present form.

Most of these efforts try to modularize exception handlers for different goals. However, none of them focus on error recovery issues and the ways in which we could leverage modular error handling to realize modular recovery quickly and easily. Further, unlike other papers, we describe how this could be done. However, it should be noted that this paper does not exercise any sophisticated error recovery schemes. It only provides an illustrative implementation of simple error reporting and recovery techniques to demonstrate the successs of our modularization efforts. Implementation of more complex and efficient error recovery techniques remains as future work.

6. CONCLUDING REMARKS

We have shown how error recovery can be modularized by leveraging the expressive and encapsulation powers of AOP. This is the first published description of how to implement modular error recovery in compilers and define the control flow pertaining to error handlers without any language extensions. In contrast to other proposals, this study identifies how modularizing error handling provides opportunities for re-using existing software artifacts and knowledge to create and add new error recovery schemes.

Error handling and recovery often involves comprehending and modifying an unfamiliar and complex code base. Our approach makes it easier to quickly identify the source and location of errors, understand their behavior and test new recovery schemes. To facilitate this process, this work addresses queries such as

- what is the control flow in which this error occurs?
- what is the context at the error site?
- which part of code accommodates a new error recovery flow path?
- how would the system behave in the absence of error recoverers?
- how to capture this error in yet another site?

Answers to such queries provide users with a broader perspective of error concerns in the system - such as structural, relational and behavioral - by the use of static and dynamic information. These sources of information all help the developers make better informed decisions about error recovery.

We experimentally demonstrate that, in exchange for modest runtime overhead, error recovery modularization leads to aforementioned benefits, in addition to a clean separation of error concerns from functional concerns.

Current study provided only a prototype implementation of modular error handling and recovery. Our ongoing work encompasses more extensive error recovery, and an analysis of the re-usability of pointcuts that comes with added recovery schemes. For this, we intend to provide a taxonomy for errors in compilers based on the similarity of their associated contexts, reporting and handling sites, and control flows of occurrences.

7. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 2006.

[2] N. Cacho, F. C. Filho, A. Garcia, and E. Figueiredo. EJFlow: Taming exceptional control flows in aspect-oriented programming. In *AOSD '08*, pages 72–83, New York, NY, USA, 2008. ACM.

[3] A. Colyer, A. Clement, R. Bodkin, and J. Hugunin. Using AspectJ for component integration in middleware. In *OOPSLA '03*, pages 339–344, New York, NY, USA, 2003. ACM.

[4] R. P. Corbett. *Static semantics and compiler error recovery*. PhD thesis, 1985.

[5] Fernando, Castor, A. Filho, C. Garcia, Mary, F, and Rubira. Extracting error handling to aspects: A cookbook. In *ICSM*, pages 134–143. IEEE, 2007.

[6] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranh ao, A. Garcia, and C. M. F. Rubira. Exceptions and aspects: the devil is in the details. In *FSE '06*, pages 152–162, New York, NY, USA, 2006. ACM.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[8] E. Hilsdale and J. Hugunin. Advice weaving in aspectj. In *AOSD '04*, pages 26–35, New York, NY, USA, 2004. ACM.

[9] K. Hogstedt. Automatic detection and masking of nonatomic exception handling. *IEEE Transactions on Software Engineering*, 30(8):547–560, 2004.

[10] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[11] K. J. Lieberherr and D. Orleans. Preventive program maintenance in Demeter/Java. In *ICSE '97*, Boston, MA, 2004. ACM.

[12] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *ICSE '00*, pages 418–427, New York, NY, USA, 2000. ACM.

[13] B. Liskov and J. Wing. A behavioral notion of subtyping. *TOPLAS*, 16(6):1811–1841, Nov. 1994.

[14] N. McEachen and R. T. Alexander. Distributing classes with woven concerns: an exploration of potential fault scenarios. In *AOSD '05*, pages 192–200, New York, NY, USA, 2005. ACM.

[15] M. P. Robillard and G. C. Murphy. Designing robust Java programs with exceptions. In *FSE '00*, pages 2–10, New York, NY, USA, 2000. ACM.