Toward Automated Enforcement of Error-Handling Policies

Douglas R. Smith and Klaus Havelund
Kestrel Technology LLC
Palo Alto, CA 94304

August 8, 2005

Abstract

Modern systems are prone to failure due to poor handling of errors that might arise. We report on a design for a tool called HandlErr that allows system developers (1) to state error-handling policies in a modular class-like notation, and (2) to automatically enforce those policies throughout the system code. The enforcement mechanisms are based on recently developed scalable and conservative static analysis algorithms. When static analysis cannot provide enough information about the applicability of a policy at a program point, then runtime monitoring code is inserted and the error-handling policy is applied based on runtime information.

1 Introduction

System developers have a natural tendency to focus on nominal behavior during design, often deferring an exhaustive analysis and treatment of possible abnormal situations. This strategy makes some sense since error-handling code is voluminous and obscures the nominal flow of control. Empirical measures of the amount of code devoted to error-handling in fielded systems vary from a few percent up to two-thirds [2, 15], with the amount increasing with code size and age. However, post-mortem analysis of a wide range of system failures often points to poor handling of errors. The more that a system is embedded and dependent on proper data and interaction with other systems, the more defensive it needs to be in order to prevent failures. Better techniques for supporting the development of robust system code would be a general value.

This study addresses the problem of creating a scalable technology for developing robust software systems. Although we focus on Java, the technical approach applies to most programming languages. We describe the design for a tool, called HandlErr , that takes as input a large non-robust Java program together with modular specifications of error handling policies, and robustifies the program by automatically inserting error detection and handling codes according to the policies. The error handling policies are expressed in a Java-like notation for compatibility with current programming practice. The modular policies support a more productive programming practice that produces higher quality code. HandlErr could be used both during system development and to robustify legacy code, without affecting functionality.

```
class AddNumbersFromFile {

  static void doIt(String fileName) throws IOException {
    DataInputStream source = null;
    if(fileName!=null)
       source = new DataInputStream(new FileInputStream(fileName));
    int count = source.readInt();
    int sum = addEm(source,count);
    System.out.println("Sum is " + sum);
  }

  static int addEm(DataInputStream s, int c) throws IOException {
    int sum = 0;
    for (int i = 0; i < c; i++)
      sum += s.readInt();
    if(s.available()==0)s.close();
    return sum;
  }
}
```

Figure 1: **Nonrobust Java Program**

The key idea of HandlErr is to separate error-handling policies from nominal case code. This separation of concerns is closely related to aspect-oriented programming, which our techniques extend. HandlErr takes a collection of user-specified error-handling policies and system code and composes them by enforcing the policies in the code. The effect is to insert error-handling code at all code locations where the policies apply. By using sound static analysis algorithms to detect where the policies apply, HandlErr ensures that no potential faults are overlooked. By allowing the user to express modular error-handling policies, HandlErr supports more uniform error-handling, more productive focus on error-handling content, and significantly reduced effort to change the policies. HandlErr will uniformly and exhaustively apply the policies throughout the code, automatically.

Although the examples in this paper do not illustrate it, it is possible that the static analysis to apply a policy cannot determine which policy to apply at a certain code location. In that case, runtime test code is inserted that makes the determination at runtime. Thus HandlErr will support a spectrum of analysis from purely static weaving of policies to runtime monitoring and application.

2 Motivating Example

Figure 1 shows a simple Java program that reads numbers from a file and sums them. We will use this as a running example since it exhibits several of the most common sources of errors and poor error-handling: (1) I/O operations, (2) the handling of ill-formed data, and

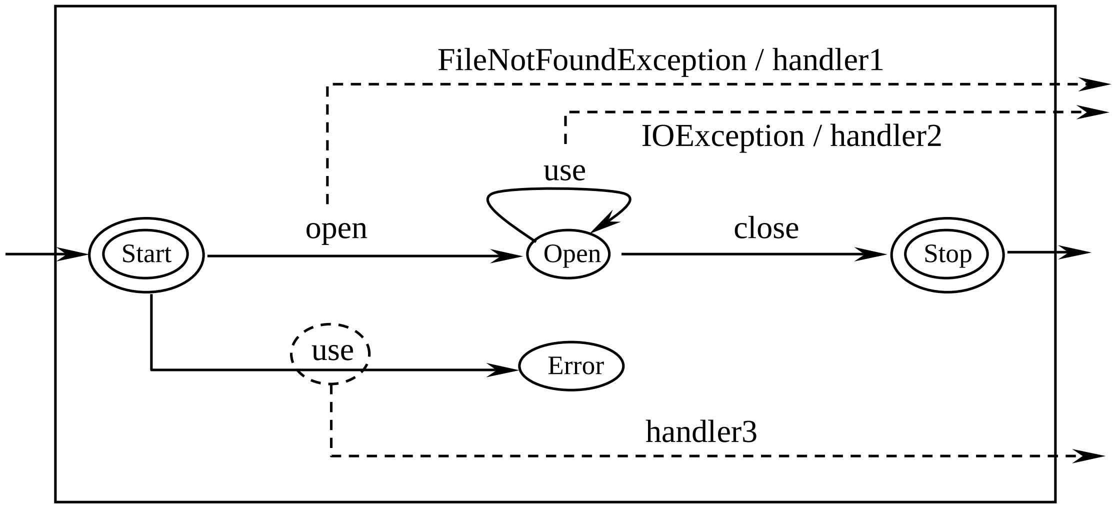

Figure 2: **Simplified Generic File Management Policy**

(3) obligations to observe proper and complete sequencing of operations in the acquisition and release of resources. The example Java program relies entirely on Java's builtin exception handlers, but when fully robustified (to give pertinent messages, to close files upon termination, and to handle minor errors), the resulting code is mostly error handling code.

This paper reports on the the two key conceptual bases for HandlErr : (1) error-handling policies and (2) policy enforcement mechanisms.

We observe that errors are the flip-side to normal behavior, so it seems reasonable to *specify error-handling policies in the context of normal-case behavior*. Consequently, our approach has been to express policies as state machines that represent both the normal behavior of some aspect of a system together with the abnormalities that may arise. We call these state machines *error-handling policies*, or *policies* for short.

For example, a simplified language-neutral policy for file management is depicted in Figure 2 (a more complete textual notation is presented in the next section). The outer box indicates the temporal scope of the policy. Control-flow arrows are labeled with guarded actions. Solid arrows correspond to normal program control flow. Dotted arrows correspond to abnormal or exceptional control flow, expressing the action taken when an operation throws an exception. For example, if a `FileNotFoundException` is thrown during an `open` operation then `handler1` should be executed. The handlers associated with each exception are presented as code templates to be instantiated at design-time by the policy enforcement mechanism. The arrows exiting the policy correspond to the possible outcomes of the behavior, both normal-case passing of control and the throwing of exceptions. Certain states are *safe*, written with a double circle, indicating a global obligation with respect to this policy. In this example, the enforcement algorithm is obliged to ensure that whenever the program is about to terminate while in an unsafe state, it must close all open files. The figure also represents erroneous actions, particularly, using a file before opening it. The policy specifies that the `use` action in the `Start` state should be replaced with the throw of an exception.

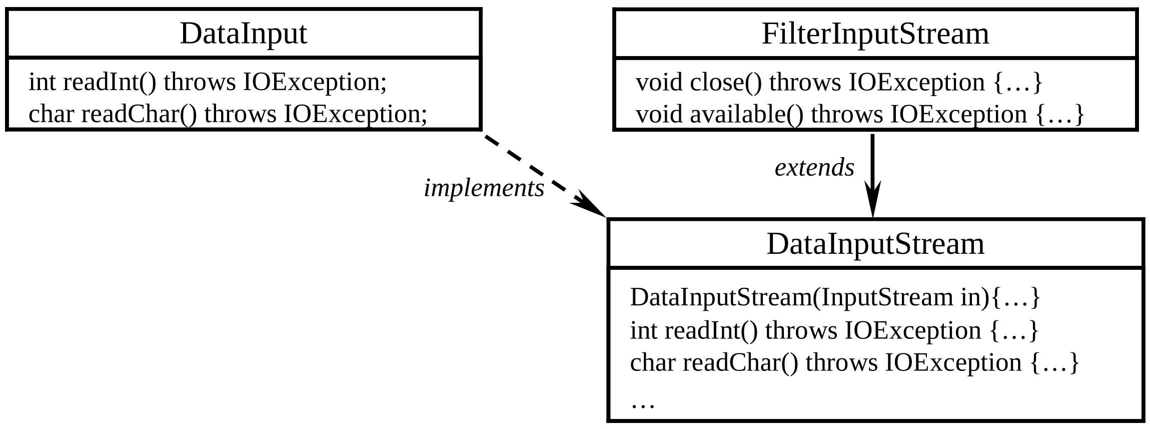
DataInput
int readInt() throws IOException;
char readChar() throws IOException;
FilterInputStream
void close() throws IOException {...}
void available() throws IOException {...}
implements
extends
DataInputStream
DataInputStream(InputStream in){...}
int readInt() throws IOException {...}
char readChar() throws IOException {...}
...

Figure 3: **Partial Interface to java.io.DataInputStream**

3 Policy Language

HandlErr treats policies in a manner similar to classes in Java, allowing extension/inheritance and instantiation. Moreover, policies are expressed using an extension of Java syntax with pattern notations.

3.1 Policy Syntax

Our running example uses the DataInputStream class from java.io whose interface and dependencies are sketched in Figure 3. DataInputStream provides methods for opening a stream (via its constructor), reading various types, checking availability of data, closing a stream, and others. A class policy that constrains the behavior of any instance of DataInputStream is shown in Figure 4. Generally, a policy has the following form:

```
policy policyName extends policy-list {
  instance-variable*
  transition*
  }
```

where (1) policy-list is a comma-separated list of zero or more policies, (2) * is used to denote zero or more occurrences of, (3) each instance variable is declared using Java syntax (separated by semicolons), and (4) transitions have the form

```
policy DataInputStreamPolicy {
  string filename;
  DataInputStream in;

  Start: { DataInputStream(FileInputStream(filename)) returns in } -> Open

  Start: { in.read*() } -> Error
           replace {throw new Error("Attempt to read from an unopen File"); }

  Start: { in.available() } -> Start
           replace {throw new Error("Attempt to invoke available on an unopen File");}

  Start: { in.close() } -> Start
           replace {print("Attempt to close an unopen File"); }

  Open: { in.read*() } -> Open
          catch (EOFException e)
                {throw new Error("EOF: insufficient data in file " + filename); }
          catch (IOException e)
                {throw new Error("Cannot read from File " + filename); }

  Open: { in.available() } -> Open
          catch (IOException e)
                {throw new Error("Unable to determine whether file "
                                 + filename + " contains more data"); }

  Open: { in.close() } -> Closed
          precondition {in.available() == 0}
                       {System.out.println("Closed file " + filename
                                           + " when it contained extra data"); }

  Open : { exit } -> Closed
           preaction
           { System.out.println("Performing a missing close on file " + filename);
             in.close();
           }

  Closed: { in.read*() } -> Closed
            replace {throw new Error ("File " + filename + "already closed"); }

  Closed: { in.available() } -> Closed
            replace {throw new Error ("Attempt to invoke available on a closed file: "
                                      + filename); }

  Closed: { in.close() } -> Closed
            replace {throw new Error ("File " + filename + "already closed"); }

}
```

Figure 4: **Policy on DataInputStreams**

```
source-state: {transition pattern} -> target-state
    precondition  {precondition pattern}  {precondition handler pattern}
    postcondition {postcondition pattern} {postcondition handler pattern}
    invariant     {invariant pattern}     {invariant handler pattern}
    preaction                             {preaction handler pattern}
    postaction                            {postaction handler pattern}
    replace                               {replacement code pattern}
    catch (Exception e)                   {exception handler pattern}
```

where zero or more of the optional clauses (precondition, postcondition, invariant, preaction, postaction, replace, catch) may occur.

The transition pattern is a mixture of pattern expressions (defined below) and Java that is intended to match Java source code. Upon normal completion of the transition code (i.e the source code that matches the transition pattern), a transition is made to the target-state.

If a precondition clause is specified, then an instance of the precondition pattern is tested before the transition code - if it fails, then an instance of the precondition handler pattern will be executed. Similarly, the postcondition is tested after the transition code and the invariant is tested both before and after the transition.

A preaction clause indicates that the appropriate instance of the preaction handler pattern should be executed before the transition. Similarly, a postaction clause indicates that the appropriate instance of the postaction handler pattern should be executed after the transition. A replace clause is used to replace the code that matches the transition pattern with an instance of the replacement code pattern. It is intended to be used when a certain operation or event is illegal in the source-state (e.g. it corresponds to a security violation, or abuse of a resource).

A catch clause specifies a class of exceptions that the transition code may throw. The intention is to wrap the corresponding code in a try-catch clause with an appropriate instance of the exception handler pattern. The handler patterns are arbitrary Java code that can refer to any variable referenced in the transition pattern. A handler pattern may or may not contain a throw, as appropriate. A handler pattern without a throw may be used to express code for fixing the current state when an error occurs, or simply to emit a warning message.

Pattern language for code templates

The patterns between curly braces in transitions and clauses are Java templates - an extension of Java syntax with pattern notations. Here are some of the pattern notations needed in our examples:

- *String patterns* – { read*() } matches any method invocation whose name begins “read”. More generally, it seems reasonable to utilize AspectJ-like pattern constructs for describing expression and method-call patterns.
- *Declared Variables* – when `in` is a declared variable, the first occurrence of pattern { in.read*() } binds `in` to the object on which `read` is invoked. Subsequent occurrences of `in` must match that binding.

- *Implicit Naming* – { expr returns x } matches any expression (e.g. method call) that matches { expr }, but the matching process binds the result to `x`.
- *Context patterns* – { .. in.read*() .. } matches the smallest context that includes a method call that matches {in.read*()}.

As an example, when variable `f` has been declared, the pattern

```
{ sum = ..  f.read*() ..; }
```

matches the statement

```
sum = sum + in.readInt();
```

with binding {`f` ↦ `in`}.

A *policy instance* is defined by values created in the source code that correspond to the policy variables declared in the policy. The map from policy variables to source code expressions that create the values of a policy instance is called the *instance binding.* For example, an instance of policy DataInputStreamPolicy (Figure 4) that is applicable to the AddNumbersFromFile example (Figure 1) is given by the instance binding

```
{fn ↦ fileName,
 in ↦ source}
```

The *scope* of a policy instance is the innermost block that encloses the value flow of the instance bindings.

One special idiom arises in order to express and handle obligations:

```
source-state: {exit} -> safe-final-state
              preaction {preaction code pattern}
```

Here the transition pattern is the keyword `exit`, which matches any normal or abnormal exit from the scope of a policy instance. This rule defines an obligation to transition the system to a safe state before exiting. In words, the obligation states that when the system attempts to exit from the policy scope and it is in the unsafe `source-state`, then execute the preaction code pattern before exit.

```
policy AddNumbersPolicy extends DataInputStreamPolicy {

  Open0: { count = in.read*() } -> Open1
         postcondition (0 <= count && count <= 1000)
             {System.out.println("count received an illegal value: "
                                 + count
                                 + "\nsetting count to 0");
              count = 0;}
         catch (EOFException e)
              {throw new Error("File " + in.filename
                                 + " contains no data!"); }

  Open1: { in.read*() } -> Open1

}
```

Figure 5: **Policy for AddNumbersFromFile**

3.2 Policy Extension and Inheritance

Just as Java allows building new classes by inheriting members, thereby achieving reuse and specialization, HandlErr also supports the development of policies by inheritance. The extending policy must use the same states as its parent, but with (optional) ordinal suffixes added. For example, the policy in Figure 4 is extended to obtain an application-specific policy shown in Figure 5 which takes into account that program AddNumbersFromFile has two stages of reading from the file: first read in a count of the numbers in the rest of the file, then read in the remaining numbers and sum them. The policy overrides some transitions of `DataInputStreamPolicy` so that they give more meaningful messages. In `AddNumbersFilePolicy`, the `Open` policy state from DataInputStreamPolicy is replaced by `Open0` and `Open1`. This numbering scheme ensures that HandlErr knows how the extending state machine is a refinement of the parent state machine (more detail on state machine refinement can be found in [12]). If a transition has several catch clauses, they are ordered (topologically) by specificity. When there is a conflict, the extending policy overrides the parent policy. [1] [2]

HandlErr is envisioned to provide a basic hierarchy of policies that mirrors the Java libraries, and thus providing generic application-independent error-handling support. By using policy extension, the developer can create application-specific policies. Note the handling of a violated postcondition in the first transition in `AddNumbersFilePolicy`: here the user indicates that if the read-in value of `count` is out of bounds, then for this application it is acceptable to set the value to zero and continue (after also warning the user of the problem). This is an example of an application-specific remedial action in response to an error.

[1]examples of potential conflicts?

[2]Potential criticism of application-specific policies – worst case there is one transition per source statement, then the separation of policy and code is disruptive rather than simplifying an helpful; e.g. as with Open0 and Open 1 – counter-criticism is that the policy context and structure (refinement of the generic policy) is useful.

3.3 Policy Enforcement

HandlErr 's policy enforcement strategy depends on static program analysis to find where the policies apply in the system design. The analysis must be *conservative* – whenever there exists an instance of policy `E` in program `P`, then HandlErr must find it.

Static program analyses are further classified in terms of *context sensitivity* – a context-insensitive analysis produces for each method a formula characterizing the effect of calling it, whereas a context-sensitive analysis produces for each method and each calling context, a formula characterizing the effect of calling the method.

For example, a flow-insensitive points-to analysis of `AddNumbersFromFile` would find that the value created by the constructor `DataInputStream` in `doIt`, say `v`, may flow to variables `source` and `s`. Restated, the analysis asserts that at every program point `source` = `v` $\vee$ `s` = `v`. Similarly, a flow-sensitive dataflow analysis of `AddNumbersFromFile` would assert that the policy state before the statement `count = source.readInt();` is either `Start` or `Open`.

To gain scalability in enforcing policies, HandlErr may produce incomplete analyses and therefore some false positives. For our purposes however, false positives will result in extra error-handling code that will never be executed. Such dead code costs no extra runtime, and we believe, negligible extra code size. However, false positives can only arise at program points with ambiguous policy states (more than one possible policy state). In this case, we may instrument the code with runtime monitors in order to decide exactly which error handlers apply. This runtime tracking occurs a small overhead. In other words, where we cannot get exact analysis statically, we regain it dynamically, paying a small price of runtime overhead to achieve exact enforcement of policies.

The enforcement strategy proceeds in stages, as presented in the following subsections.

3.3.1 Value-Flow Analysis

The first stage is interprocedural value-flow analysis that links program points to value creation, and computes the value flow across statements and method calls. The goals are (1) to identify policy instance creation sites by identifying program points where the values are created for the instance variables, (2) to compute alias sets, and (3) to determine the scope of the value flow for the policy. This information is used to determine the scope of a policy instance – the innermost block that encloses the value flow of the instance bindings and to support policy simulation in the next stage.

Several recent projects have presented points-to algorithms that scale well. Das' flow-insensitive pointer analysis runs in near-linear time and was used to analyze Word97 (approx 2.1 MLOC) in two minutes on a dated PC [1, 3]. Whalen and Lam [16] have developed a context-sensitive pointer alias analysis for Java using OBDDs (Ordered Binary Decision Diagrams) to compactly represent and reason about enormous numbers of calling contexts.

In our example, value-flow analysis finds an instance of policy `AddNumbersFilePolicy` (Figure 5) determined by the creation of a stream by the constructor call in `doIt`. The scope of the instance is determined by the flow of the stream value to `source`, which can flow to `s` in `addEm`,

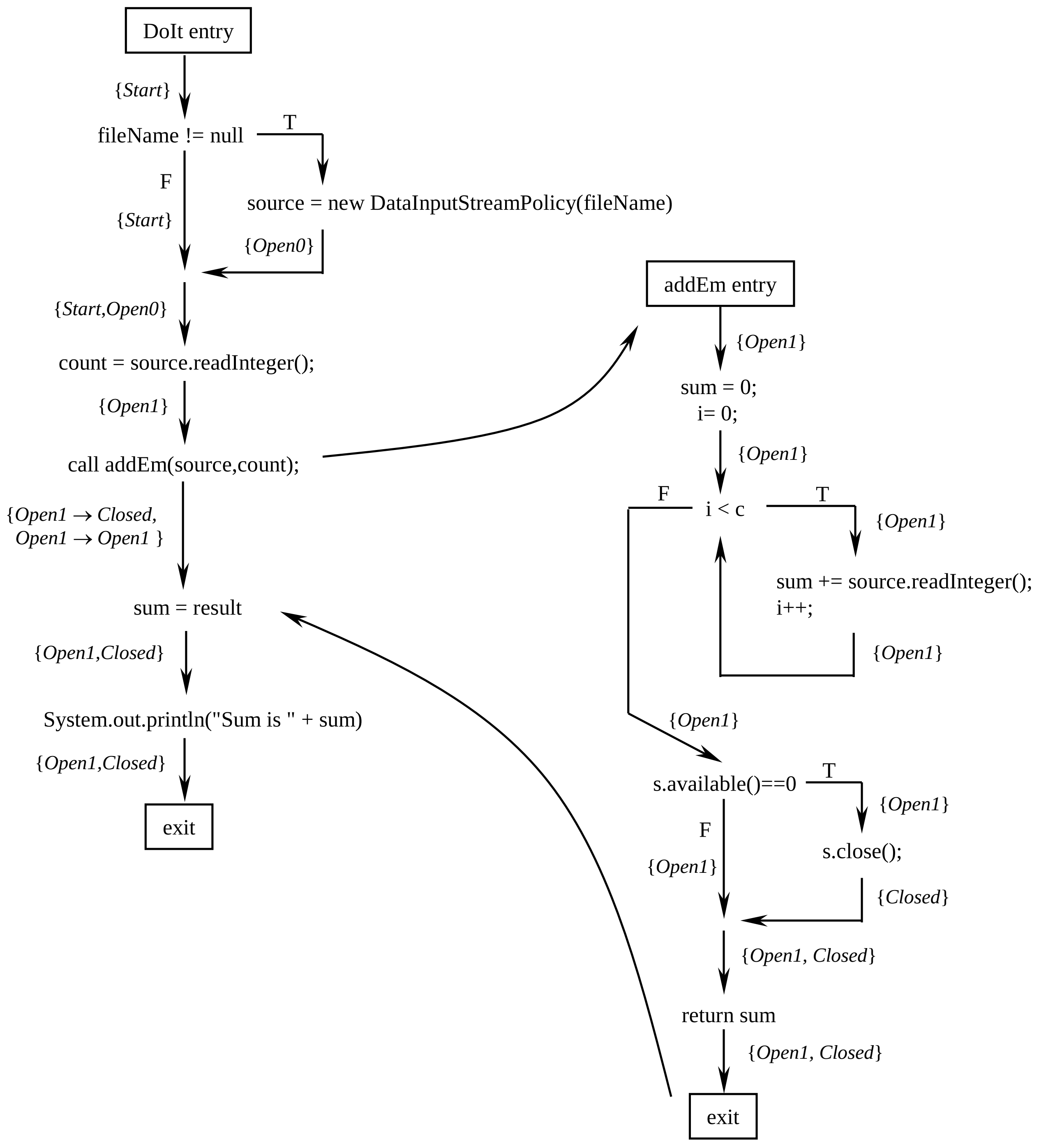

Figure 6: Policy Simulation on RobustAddNumbersFromFileClass

and which is deallocated upon exit from `doIt`. Thus the scope of the instance is the `doIt` method.

3.3.2 Policy Simulation

The second stage is a flow-sensitive interprocedural dataflow analysis that simulates the policy automata over the Control Flow Graph (CFG) of the application code. The result of policy simulation includes (1) a map from source code program points to sets of policy states, (2) a map from source code expressions and statements to sets of policy transitions, (3) a map from program points and policy variables to source code expressions (used for pattern instantiation), and (4) a summary of the state changes effected by method calls. The value-flow analysis from the previous step is used to eliminate unnecessary work in this stage by restricting the policy simulation to just those value flows that may occur in each method call context. The analysis is *ambiguous* if any program point has more than one policy state associated with it, and it is *unambiguous* otherwise.

The policy simulation needed by HandlErr builds on a long tradition of dataflow analysis algorithms going back at least to Kildall [11] who defined the general problem of computing the meet-over-all-paths problem over a finite semilattice. The formal structure of this class of problems admits a low-order polynomial-time flow-sensitive exact analysis algorithm scheme (essentially a Tarski fixpoint iteration algorithm applied to program structure). Recently, Kildall's algorithm has been extended from intra-procedural to inter-procedural analysis in the RHS algorithm by Reps, Horowitz, and Sagiv [13], still preserving low-order polynomial-time complexity. Engler et al. [7] and Das et al. [4] adapted and extended the RHS algorithm to safety properties represented as state machines. The ESP algorithm of Das et al. adds in path sensitivity [4] and value flow analysis [1], supporting correlation between program properties and policy states and more accurate matching in the presence of aliasing. We have developed a variant of the above algorithms and used hand-simulation to test it. These algorithms gives us confidence that we can implement scalable and conservative policy simulation code.

For example, Figure 6 shows the result of performing policy simulation on the CFGs of AddNumbersFromFile. The control flow arcs are labeled with exact sets of policy states. As in the RHS algorithm, code blocks for procedure call are separated from procedure return, and the arc between them is labeled with a summary of the effect of the procedure call. Note that the analysis is ambiguous, particularly due to the conditionals in `doIt` and `addEm`.

3.3.3 Policy Enforcement

HandlErr will support two forms of enforcement, depending on whether the policy simulation produced an ambiguous or unambiguous analysis for a given policy.

Ambiguous Analysis Distinguishing ambiguous analyses is important for several reasons. First, in general it will be difficult (or impossible in the case of concurrency) to determine statically which transition clauses to apply. Consequently, ambiguous analyses lead to the need to perform some runtime state tracking to provide the information missing from the static analysis. Second, if the code is ambiguous with respect to a policy, then it is likely to be poorly organized, hard to understand, and even incorrect. Consequently there should be value in informing the programmer about the sources of ambiguity and attempting to restructure the code to remove ambiguities.

A policy `CP` whose variables are typed over classes $\texttt{C}_1, ..., \texttt{C}_n$ can be translated into extensions of those classes. Constants are introduced for each policy state, as well as a `currentState` field. The policy fields are also declared in order to provide information on the policy bindings. Each method `m` in `C` is overridden by code that performs a case analysis on the value of `currentState`, performing error-handling or invocation of `super.m` as appropriate. If there is more than one class involved, then an update to the policy state in one class instance must be communicated at the same time to the other class instances to which it is coupled. Any obligations are handled by defining a finalize clause. The effect is (1) to instrument instances of the class with code to track the runtime policy state of the object, (2) to expose that state and related information to the runtime system, and (3) to provide context-specific error handling.

For our example, AddNumbersPolicy has a DataInputStream field and HandlErr automatically translates this policy into an extension of DataInputStream shown in Figures 7 and 8. HandlErr then automatically replaces DataInputStream with DataInputStreamForAddNumbers in the user's application code, as shown in Figure 9. At the same time, HandlErr will warn the user that the two conditionals in `doIt` and `addEm` give rise to ambiguous policy state.

```
public class DataInputStreamForAddNumbers extends DataInputStream {
  public static final int Start = 1;
  public static final int Open0  = 2;
  public static final int Open1  = 3;
  public static final int Closed  = 4;
  int currentState = Start;
  public String filename;

  public DataInputStreamForAddNumbers(String filename) throws FileNotFoundException {
      super(new FileInputStream(filename)); // field in stores the file handle
      this.filename = filename;
      this.currentState = Open0;
  }

  public boolean inState(int state){
      return this.currentState == state;
  }

  // renamed because readInt() is final
  public int readInteger() throws IOException {
      int x = 0;
      switch(currentState){
        case Start:
            throw new Error("Attempt to read from an unopen File");
        case Open0:
            try{
                x = super.readInt();
            } catch (EOFException e){
                throw new EOFException("File" + filename + "contains no data!");
            } catch (IOException e){
                throw new IOException("Cannot read from file " + filename);
            }
            if(!(0 <= x && x <= 1000)) {
                System.out.println("count received an illegal value: "
                                   + Integer.toString(x)
                                   + "\nsetting count to 0");
                x = 0;
            }
            currentState = Open1;
            break;
        case Open1:
            try{
                x = super.readInt();
            } catch (EOFException e){
                throw new EOFException("EOF: insufficient data in file " + filename);
            } catch (IOException e){
                throw new IOException("Cannot read from file " + filename);
            }
            break;
        case Closed:
            throw new Error("File " + filename + "already closed");
      }
      return x;
    }
```

Figure 7: **Implementation of Policy AddNumbersPolicy (part 1/2)**

```
  public int available() throws IOException {
      int avail = 0;
      switch(currentState){
        case Start:
           throw new Error("Attempt to invoke available on an unopen File");
        case Open0: case Open1:
           try{
               avail = super.available();
           } catch (IOException e){
                   throw new Error("Unable to determine whether file"
                                    + filename + " contains extra data");
           }
           break;
        case Closed:
           throw new Error("Attempt to invoke available on a closed file: "
                          + filename);
      }
      return avail;
  }

  public void close() throws IOException {
    switch(currentState){
        case Start:
            throw new Error("Attempt to close an unopen File");
        case Open0: case Open1:
            currentState = Closed;
            if(super.available() != 0)
                System.out.println("Closed file " + filename
                                   + " when it contained extra data");
            super.close();
            break;
        case Closed:
            throw new Error("File " + filename + " already closed");
    }
  }

// handle the obligation to close the file, if not already done so.
  protected void finalize() throws Throwable {
      if(!inState(Closed)){
        try {
            System.out.println("Performing a missing close on " + filename);
            super.close();
        } finally {
            super.finalize();
        }
     }
  }
}
```

Figure 8: **Implementation of Policy AddNumbersPolicy (part 2/2)**

```
public class RobustAddNumbersFromFile {

  static void doIt(String fileName) throws IOException {
    // REPLACE DataInputStream WITH DataInputStreamForAddNumbers
    // BEGIN
    DataInputStreamForAddNumbers source = null;
    // END

    if(fileName!=null){
        // REPLACE
        //   DataInputStream(new FileInputStream(fileName));
        // WITH
        //   DataInputStreamForAddNumbers(fileName)
        // BEGIN
        source = new DataInputStreamForAddNumbers(fileName);
        // END
    }

    // REPLACE readInt WITH readInteger
    // BEGIN
    int count = source.readInteger();
    // END

    int sum = addEm(source,count);
    System.out.println("Sum is " + sum);
  }

  static int addEm(DataInputStreamForAddNumbers s, int c) throws IOException {
  int sum = 0;
  for (int i = 0; i < c; i++)
  // REPLACE readInt WITH readInteger
  // BEGIN
      sum += source.readInteger();
  // END
  if(s.available()==0)s.close();
  return sum;
  }
}
```

Figure 9: Policy Applied to AddNumbersFromFile

There are some simple semantics-preserving code transformations that HandlErr could apply to reduce or eliminate ambiguous analyses. For example, an if-then-else in which the two branches produce different analyses:

```
if(test){
  block1;
} else {
  block2;
}
block3;
```

can be transformed to the equivalent

```
if(test){
  block1;
  block3;
} else {
  block2;
  block3;
}
```

which will work to keep differing policy states in distinct cases. For example, the following policy-ambiguous code in `doIt`

```
    if(fileName!=null)
       source = new DataInputStream(new FileInputStream(fileName));
    count = source.readInt();
```

is transformed to

```
    if(fileName!=null){
       source = new DataInputStream(new FileInputStream(fileName));
       count = source.readInt();
    } else {
      count = source.readInt();
    }
```

which gives an unambiguous analysis along each branch. Applying the policy clauses and simplifying the code ultimately results in the unambiguous code

```
    if(fileName==null){
       throw new Error("Attempt to read from an unopen File");
    }
    source = new DataInputStream(new FileInputStream(fileName));
    count = source.readInt();
```

```
class AddNumbersFromFile1 {

  static void doIt(String fileName) throws IOException {
    DataInputStream source = null;
    if(fileName==null){
       throw new Error("Attempt to read from an unopen File");
    }
    source = new DataInputStream(new FileInputStream(fileName));
    count = source.readInt();
    int sum = addEm(source,count);
    System.out.println("Sum is " + sum);
  }

  static int addEm(DataInputStream s, int c) throws IOException {
    int sum = 0;
    for (int i = 0; i < c; i++)
      sum += s.readInt();
    s.close();
    return sum;
  }
}
```

Figure 10: **AddNumbersFromFile with Unambiguous Analysis**

which is closer to what the programmer, arguably, should have written in the first place.

In our example, there is no obvious transformation that would eliminate the ambiguity that results from the conditional in `addEm`. HandlErr can provide a warning about it, but the programmer must decide whether and how to clean up the code.

Unambiguous Analysis If the analysis is unambiguous, then HandlErr checks to see whether the application code has detectors and handlers as specified by the policy. If not, then appropriate instances of policy clauses are added to the source code. As in the ambiguous case, HandlErr generates extensions to the underlying classes of the policy, but in this case the extensions only record the policy bindings and do not provide policy-state tracking. HandlErr treats obligations to drive the code to a safe state by an appropriate instantiation of the policy obligation clause at each program point where an exit from an unsafe state could be made.

Suppose that `AddNumbersFromFile` has been transformed into code that has an unambiguous analysis, as shown in Figure 10. HandlErr then generates the class extension in Figure 11 and the robustified version of `AddNumbersFromFile` shown in Figures 12, 13, and 14.

```
public class DataInputStreamForAddNumbers1 extends DataInputStream {
  public String filename;

  public DataInputStreamForAddNumbers1(String filename) throws FileNotFoundException {
      super(new FileInputStream(filename)); // field in stores the file handle
      this.filename = filename;
  }
}
```

Figure 11: **Class Extension for AddNumbersFromFile1**

```
class RobustlyAddNumbersFromFile1 {

  static void doIt(String fileName) throws IOException{
    // REPLACE DataInputStream WITH DataInputStreamForAddNumbers1
    // BEGIN
    DataInputStreamForAddNumbers1 source = null;
    // END

    if(fileName==null){
        // ROBUSTIFY:
        // count = source.readInt();
        // WITH:
        //   Start: { in.read*() } -> Error
        //      replace {throw new Error("Attempt to read from an unopen File"); }
        // BEGIN
           throw new Error("Attempt to read from an unopen File");
    }
    // REPLACE DataInputStream WITH DataInputStreamForAddNumbers1
    // ROBUSTIFY:
    //   source = DataInputStream(new FileInputStream(fileName));
    // WITH:
    // Start: { new DataInputStream(new FileInputStream(fn)) returns in} -> Open1
    //        catch (FileNotFoundException e)
    //        {throw new Error("File " + fn + " cannot be found"); }
    // BEGIN
    try {
      source = new DataInputStreamForAddNumbers1(fileName);
    } catch (FileNotFoundException e) {
          throw new Error("File " + fileName + " cannot be found");
    }
    // END
```

Figure 12: Robustified AddNumbersFromFile (part 1/3)

```
  int count = 0;
  // ROBUSTIFY:
  // count = source.readInt();
  // WITH:
  // Open0: { count = in.read*() } -> Open1
  //   postcondition (0 <= count && count <= 1000)
  //       {System.out.println("count received an illegal value: "
  //                           + Integer.toString(count)
  //                           + "\nsetting count to 0");
  //        count = 0;}
  //   catch (EOFException e)
  //         {throw new Error("File " + in.filename + " contains no data!"); }
  // BEGIN
  try {
    count = source.readInt();
  } catch(EOFException e){
        source.close();
        throw new Error("File " + source.filename + " contains no data!");
  } catch(IOException e){
        source.close();
        throw new Error("Bad data in file" + source.filename);
  }
  if(!(0 <= count && count <= 1000)){
      System.out.println("Count received an illegal value: "
                            + Integer.toString(count)
                            + "\nsetting Count to 0");
     count = 0;
  }
  // END

  int sum;
  try{
      sum = addEm(source,count);
  } catch (IOException e){
      source.close();
      throw e;
  }
  System.out.println("Sum is " + sum);
}
```

Figure 13: Robustified AddNumbersFromFile (part 2/3)

```
  static int addEm(DataInputStreamForAddNumbers1 s, int c) throws IOException {
      int sum = 0;
      for (int i = 0; i < c; i++) {
        // ROBUSTIFY:
        // sum += source.readInt();
        // WITH:
        // Open: { sum = .. in.read*() ..} -> Open
        //       catch (EOFException e)
        //       {throw new Error("Not enough data in file" + fn); }
        // Open: { in.read*() } -> Open
        //       catch (IOException e)
        //       {throw new Error("Bad data in file" + fn); }
        // BEGIN
        try {
          sum += s.readInt();
        } catch(EOFException e){
            s.close();
            throw new Error("Not enough data in file" + s.filename);
        } catch(IOException e){
            s.close();
            throw new Error("Bad data in file" + s.filename);
        }
        // END
     }
     s.close();
     return sum;
  }

}
```

Figure 14: Robustified AddNumbersFromFile (part 3/3)

4 Usage Levels for HandlErr

We envision that HandlErr has several different levels of usage, depending on user expertise and cost/benefit tradeoffs.

1. *Basic Usage* – The user treats HandlErr as a completely automatic Java preprocessor, requiring no effort or understanding from the user. HandlErr analyzes the Java library classes used in the application and extracts those class policies from its library that are applicable. The user's application is automatically instrumented with internal tracking of policy state and richer error-handling than the Java classes. Since the process of applying class policies is simply a matter of adding some class extensions and making some simple modifications to the source code, we anticipate that the HandlErr preprocessing will take negligible time compared to compilation.

   For further flexibility, HandlErr could allow the user to browse a checklist of policies that HandlErr can apply. The user would modify this list, perhaps adding policies from domain-specific libraries and deleting others.

2. *Advanced Usage* – At this level, class and application policies are treated as a part of the programming process, essentially a kind of aspect-oriented programming specialized to error-handling and robustness. The user creates and evolves policy hierarchies along with code development. HandlErr is treated as an extension of the compiler, and the policies are treated as program modules. The effect is a more modular approach to programming which can be seen as extending Aspect-Oriented Programming to behaviorally-specified aspects.

   The user gets the following feedback from HandlErr :

   - *Compile time* – HandlErr prints messages about program statements that may lead to error states (e.g. reading a file that has been closed). HandlErr takes care to eliminate false positives and to only present information about original flaws, not the cascade of errors that may follow from them.
   - *Run time* – The augmented source code will catch a wide range of runtime errors according to the policies that HandlErr applied. The handlers provide informative feedback on the nature and location of the error. As far as possible, obligations, such as file, lock, and resource release, are performed prior to termination.

3. *Expert Usage* – A more expressive policy language is used to express policies, enabling a richer range of properties and behaviors. User expertise is needed since the cost of applying such policies rises with their semantic richness, so judgment regarding cost/benefits is required.

5 Related Work

Programming Languages and Program Transformation

Error-handling policies could be expressed via the pointcut/advice pairs of AspectJ [10]. From the point of view of AspectJ, our policy enforcement approach makes the following contributions.

First, a policy automaton generalizes the notion of pointcut by providing a behavioral context (versus a simple method call) for advice. Although one could be simulate behavioral context using AspectJ pointcuts, it would be complex and obscure. The state machine provides a clearer way to express loci of advice. Second, the state machines provide the loci for multiple advice code patterns. In AspectJ (and most other forms of AOP) each pointcut is associated with one advice template. Third, the policy automata match arbitrary code statements and is not restricted to method calls.

One must also compare the builtin mechanisms. HandlErr handles obligations such as exit from the scope of a policy instance which, to our knowledge, is not expressible in AspectJ. Conversely, AspectJ provides point-cut notations to provide special forms of behavioral context. The notation `cflow m` constrains event matching to the dynamic context of a call to method `m` (i.e. when a call to `m` is on the stack). To capture this expressivity seems to require extending the policy automata to be hierarchical, as in Statecharts [8].

Schneider's concept of enforceable security policies [14] and mechanized enforcement [6] is similar to the HandlErr approach. Our contributions include the use of the policy automaton to express compactly multiple properties, and the focus of fast static analysis to scale up HandlErr .

Program Verification

A state machine policy can be viewed as a temporal logic constraint that we intend for the target system to satisfy. Our approach is constructive in that enforcement makes the system satisfy the constraint by means of conservative static analysis and program transformations. Model checking is the well-known technique for checking whether a system satisfies a temporal logic constraint [**?**]. The difficulty of scaling-up software model-checking has motivated recent work on runtime verification of properties [5, 9] where temporal properties are compiled into runtime monitoring code. Several projects are using automata to express safety properties and to check them using static analysis. To make the checking tractable, several projects argue that fast imprecise analysis combined with filters for false positives can still provide useful debugging information [1, 4, 7, 15].

6 Implementation Notes

- use CD-simplify to eliminate as far as possible the runtime cost of evaluating pre/post-condition tests.
- policy hierarchy – HandlErr provides a basic hierarchy of policies that mirrors the Java libraries, and thus provides generic application-independent error-handling support. By using policy inheritance, the developer can create application-specific policies.

7 Summary

The HandlErr approach to robustifying large Java programs makes use of a state-machine-based policy language that expresses abnormal/error conditions in the context of constraints on normal behavior. Implementing class policies as class extensions provides a simple solution to the problem of tracking policy state under conditions of aliasing, concurrency, and value-flow over procedural boundaries. Application policies capture application semantics and they provide a framework for systematically analyzing the kinds of abnormal situations that might arise and how to handle them. Static analysis to determine policy scope enables HandlErr to discharge obligations in a timely manner, allowing for graceful shutdown upon normal or abnormal termination.

Error-handling policies must treat two orthogonal concerns: how to detect error conditions, and what to do about them when they arise. We believe that policy automata provide a precise yet intuitive formalism for expressing error situations. The other main concern, how to treat an error that arises, is mainly an application-specific problem. The simplest approach is to print an informative error message when an error occurs. However, HandlErr supports the expression of arbitrary error-handling code to restore a system to a workable state by means of Java code templates (as when `count` is given an illegal value in `RobustlyAddNumbersFromFile`).

The advantages of the HandlErr approach include

- The state machine notation provides a clear trace-based semantics to policies and we believe that users will find the notation convenient. HandlErr has been designed to reflect Java syntax and pragmatics as much as possible.
- Expressing error-handling policies as separate modules helps to focus users on what can go wrong, leading to a more complete understanding of the system and its environment, as well as supporting the construction of more robust code.
- Improved modularity leads to improved understandability and ease of evolution. Modular error-handling policies with automatic enforcement leads to uniform treatment of error-handling leading to more robust code, less time wasted in tracking down bugs, and reduced development time. In this regard, HandlErr provides a generalized form of Aspect-Oriented Programming where join-points are specified by behavioral contexts, not just statement patterns.
- The library policies and enforcement machinery encodes best-practice programming techniques; e.g. idiomatic safe coding practice that ordinary programmers may not be familiar with. An example is the formulation of the finalization code to handle obligations, including invocation of the finalization code of the object's superclass.
- Static analysis can identify program points that (with high likelihood) have errors. The effect is to aid the programmer in producing more robust code during development, and also to simplify the analysis, allowing the runtime error-handling to be more precise and effective.
- Although one might expect that robust error-handling causes a slight performance degradation, Weimer and Necula report a surprising performance improvement (17%) in file management programs, when resources are correctly deallocated in the presence of exceptions [15]. It may be that correct error-handling of resources more than pays for the overall cost of runtime tracking and error-handling.

Acknowledgments: Thanks to Alessandro Coglio for the name HANDLERR and the motivating example of this paper.

References

[1] ADAMS, S., BALL, T., DAS, M., LERNER, S., RAJAMANI, S. K., SEIGLE, M., AND WEIMER, W. Speeding up dataflow analysis using flow-insensitive pointer analysis. In *The 9th International Static Analysis Symposium (SAS'02)* (2002).

[2] CRISTIAN, F. Exception handling. In *Dependability of Resilient Computers*. BSP Professional Books, Blackwell Scientific Publications, 1989, pp. 68–97.

[3] DAS, M. Unification-based pointer analysis with directional assignments. *ACM SIGPLAN Notices 35*, 5 (2000), 35–46.

[4] DAS, M., LERNER, S., AND SEIGLE, M. ESP: Path-sensitive program verification in polynomial time. In *SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)* (2002).

[5] DRUSINSKY, D. The temporal rover and the atg rover. In *Proceedings of the SPIN 2000 Workshop* (2000), Springer-Verlag LNCS 1885, pp. 323–329.

[6] ERLINGSSON, U., AND SCHNEIDER, F. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop* (Ontario, Canada, September 1999).

[7] HALLEM, S., CHELF, B., XIE, Y., AND ENGLER, D. A system and language for building system-specific, static analyses. In *SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)* (2002).

[8] HAREL, D. Statecharts: A visual approach to complex systems. *Science of Computer Programming 8*, 3 (June 1987), 231–274.

[9] HAVELUND, K., AND ROSU, G. Monitoring Java programs with Java PathExplorer. In *Electronic Notes in Theoretical Computer Science* (2001), K. Havelund and G. Rosu, Eds., vol. 55, Elsevier.

[10] KICZALES, G., AND ET AL. An Overview of AspectJ. In *Proc. ECOOP, LNCS 2072, Springer-Verlag* (2001), pp. 327–353.

[11] KILDALL, G. A unified approach to global program optimization. In *First ACM Symposium on Principle of Programming Languages (POPL)* (1973), pp. 194–206.

[12] PAVLOVIC, D., AND SMITH, D. R. Composition and refinement of behavioral specifications. In *Proceedings of Automated Software Engineering Conference* (2001), IEEE Computer Society Press, pp. 157–165.

[13] REPS, T., HORWITZ, S., AND SAGIV, M. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages* (1995), ACM, pp. 49–61.

[14] Schneider, F. Enforceable security policies. *ACM Transactions on Information and System Security 3*, 1 (February 2000), 30–50.

[15] Weimer, W., and Necula, G. C. Finding and preventing run-time error handling mistakes. In *19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)* (Oct. 2004).

[16] Whalen, J., and Lam, M. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04)* (June, 2004).