Comprehension by Derivation

Douglas R. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304 USA
smith@kestrel.edu

Abstract

We argue that to comprehend a software system is to have a handle on its requirements, specifications, and design decisions. These kinds of information support the reuse of system code for a variety of purposes and support its ongoing extension, migration, and evolution. Our work at Kestrel Institute has focused on ways to mechanize the development and evolution of software from formal specifications. By-products of such a process include formal records of design decisions and proofs, as well as executable code. In this approach, reuse can take place at non-code levels, including domain theories, specifications, and design knowledge. Evolution takes place by modifying requirements, specifications, or design decisions, and then reusing previous design structures. When restricted to particular application domains, the generation of correct-by-construction code from specifications can be completely automatic.

1 Overview

A software system can be viewed as a composition of information from a variety of sources, including

- the concepts of the application domain,
- the requirements on the system's behavior,
- design knowledge about system architectures and interfaces, protocols, algorithms, data structures, code optimization techniques, and
- the run-time hardware/software/physical environment in which the software will execute.

In current programming practice most, or all, of this information is lost, becoming unavailable to support later maintenance, migration, and evolution activities. Belated attempts to reconstruct this information are likely to be expensive and approximate. Our thesis is that a more rational design process that is centered around requirement specifications, and proceeds to code by mechanized application of design refinements will improve not only the initial design of a system, but also leave behind exactly the information needed to rationally understand, migrate, and evolve the code. Evolution, in our view, should take place at the specification level, not the code level. If requirements change, then a modification is made to the specification and then new code is generated. If a design change is desired (e.g. a new algorithm or a new target platform), then a change to the design refinements is made and new code is generated.

This paper reviews a mechanizable framework for representing these various sources of information, and for composing them in the context of a refinement process. The framework is founded on a category of specifications. Morphisms are used to structure and parameterize specifications, and to refine them. Colimits are used to compose specifications. Diagrams are used to express the structure of large specifications, the refinement of specifications to code, and the application of design knowledge to a specification.

The framework features a collection of techniques for constructing refinements based on formal representations of programming knowledge. Abstract algorithmic concepts, datatype refinements, program optimization rules, software architectures, abstract user interfaces, policies, and so on, are represented as diagrams of specifications and morphisms. We arrange these diagrams into taxonomies, which allow incremental access to and construction of refinements for particular requirement specifications. For example, a user may specify a scheduling problem and select a theory of global search algorithms from an algorithm library. The global search theory is used to construct a refinement of the scheduling problem specification into a specification containing a global search algorithm for the particular scheduling problem.

We particularly emphasize more recent work on automating the enforcement of user-specified policies on a system design. A policy can be thought of as a modular cross-cutting constraint that helps determine what to do at decision points. Technically, enforcing a policy reduces the nondeterminism in a system, thereby creating a refinement. Policy enforcement can be thought of as a generalization of aspect-oriented programming, where the policy is a kind of aspect to be "woven" into the code. We describe two classes of policies and the means for automating their enforcement: (1) AspectJ-aspects treated as logical invariants, and (2) error-handling policies. In other work we have explored the enforcement of timing and resource constraints, and access control policies.

The framework has been partially implemented in the Specware [4] and Designware [8] systems. Specware provides basic support for composing specifications and refinements, and generating code. Code generation in Specware is supported by inter-logic morphisms that translate between the specification language/logic and the logic of a particular programming language (e.g. CommonLisp or C++). Specware is intended to be general-purpose and has found use in industrial settings. Designware extends Specware with taxonomies of software design theories and support for constructing refinements from them. Recently Specware has been extended to support the specification and refinement of behavior and the generation of imperative code. Various domain-specific generators have been built on top of Specware, including the domains of high-performance scheduling algorithms, MatLab/StateFlow compilation, and JavaCard applet generation.

2 Basic Concepts

A specification is a finite presentation of a theory. The signature of a specification provides the vocabulary for describing objects, operations, and properties in some domain of interest, and the axioms constrain the meaning of the symbols. The theory of the domain is the closure of the axioms under the rules of inference.

Example: Here is a specification for partial orders, using notation adapted from Specware. It introduces a sort E and an infix binary predicate on E, called le, which is constrained by the usual axioms. Although Specware allows higher-order specifications, first-order formulations are sufficient in this paper.

spec $Partial\text{-}Order$ is
  sort E
  op $\_le\_ : E, E \rightarrow Boolean$
  axiom *reflexivity* is $x\ le\ x$
  axiom *transitivity* is $x\ le\ y\ \wedge\ y\ le\ z\ \Longrightarrow\ x\ le\ z$
  axiom *antisymmetry* is $x\ le\ y\ \wedge\ y\ le\ x\ \Longrightarrow\ x = y$
end-spec

The generic term *expression* will be used to refer to a term, formula, or sentence.

A model of a specification is a structure of sets and total functions that satisfy the axioms. However, for software development purposes we have a less well-defined notion of semantics in mind: each specification denotes a set of possible implementations in some computational model.

A specification morphism translates the language of one specification into the language of another specification while preserving the property of provability, so that any theorem in the source specification remains a theorem under translation.

A *specification morphism* $m : T \rightarrow T'$ is given by a map from the sort and operator symbols of the *domain* spec T to the symbols of the *codomain* spec T'. To be a specification morphism it is also required that every axiom of T translates to a theorem of T'. It then follows that a specification morphism translates theorems of the domain specification to theorems of the codomain. An *interpretation* (between theories) is a slightly generalized morphism that translates symbols to expressions.

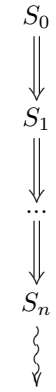
Example: A specification morphism from *Partial-Order* to *Integer* can be expressed in the form
  morphism $Partial\text{-}Order\text{-}to\text{-}Integer$ is
  $\{E \mapsto Integer,\ le \mapsto \leq\}$

Translation of an expression by a morphism is by straightforward application of the symbol map, so, for example, the $Partial\text{-}Order$ axiom $x\ le\ x$ translates to the Integer formula $x \leq x$. The three axioms of a partial order remain provable in *Integer* theory after translation.

Morphisms and interpretations are used in Specware both to express part-whole relationships in a structured specification and to express refinement between specifications.

When a morphism is used as a refinement, the intended effect is to reduce the number of possible implementations when passing from the domain spec to the codomain. In this sense, a refinement can be viewed as embodying a particular design decision or property that corresponds to the subset of possible implementations of the domain spec which are also possible implementations of the codomain.

3 Software Development by Refinement



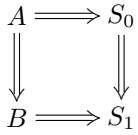
The development of correct-by-construction code via a formal refinement process is shown to the left. The refinement process starts with a specification S_0 of the requirements on a desired software artifact. Each $S_i,\ i = 0, 1, ..., n$ represents a structured specification and the arrows $\Downarrow$ are refinements. The refinement from S_i to S_{i+1} embodies a design decision which cuts down the number of possible implementations. Finally an inter-logic morphism translates a low-level specification S_n to code in a programming language. Semantically the effect is to narrow down the set of possible implementations of S_n to just one, so specification refinement can be viewed as a constructive process for proving the existence of an implementation of specification S_0 (and proving its consistency).

Clearly, two key issues in supporting software development by refinement are: (1) how to construct specifications, and (2) how to construct refinements.

A specification-based development environment supplies tools for creating new specifications and morphisms, for structuring specs into diagrams, and for composing specifications via importation, parameterization, and colimit. In addition, a software development environment needs to support a large library of reusable specifications, typically including specs for (1) common datatypes, such as integer, sequences, finite sets, etc. and (2) common mathematical structures, such as partial orders, monoids, vector spaces, etc. In addition to these generic operations and libraries, the system may support specialized construction tools and libraries of domain-specific theories, such as resource theories, or generic theories about domains such as satellite control or transportation.

A refinement-based development environment supplies tools for creating new refinements. One of our innovations is showing how a library of abstract refinements can be applied to produce refinements for a given specification. In previous work we developed abstract refinements that embodied design knowledge about (1) algorithm design, (2) datatype refinement, and (3) expression optimization. Other types of design knowledge can be similarly expressed and exploited, including interface design, software architectures, domain-specific requirements capture, and others. In addition to these generic operations and libraries, the system may support specialized construction tools and libraries of domain-specific refinements.

The key concept here is that abstract design knowledge (e.g. about datatype refinement, algorithm design, software architectures, program optimizations, visualization displays, and so on) can be expressed as abstract refinements. The domain of one such refinement represents the abstract structure that is required in a user's specification in order to apply the embodied design knowledge. The refinement itself embodies a design constraint – the effect is a reduction in the set of possible implementations. The codomain of the refinement contains new structures and definitions that are composed with the user's requirement specification.



The figure to the left shows the application of a library refinement $A \Longrightarrow B$ to a given (structured) specification S_0. First the library refinement is selected. The applicability of the refinement to S_0 is shown by constructing a *classification arrow* from A to S_0 which classifies S_0 as having A-structure by making explicit how S_0 has at least the structure of A. Finally the refinement is applied by computing the pushout in the category of diagrams. The creative work lies in constructing the classification arrow [6, 7].

4 Toward System Design by Refinement

The results described above were originally applied to the development of functional programs. To support the specification and development of complex systems, we felt the need to specify behaviors via some notion of state machine [5]. The resulting formalism, called *evolving specifications* (or simply *especs*), is a category of behavioral specifications that supports explicitly modeling the logical structure and behavior of systems. The framework supports precise, automatable operations for the composition of especs and their refinement. The espec framework is partially implemented in the Epoxi system.

As described in the previous section, library refinements of especs and refinement generators can be applied to refine a system specification to code. In a current project, we are generating idiomatic C and Java code from especs.

A key new issue emerges with the design of complex systems in which cross-cutting constraints play a more prominent role than in small programs. A concern is *cross-cutting*

if its manifestation cuts across the dominant hierarchical structure of a program. Aspect-Oriented Software Development (AOSD) offers new insights and tools for the modular development of systems with cross-cutting features [1, 3]. A simple example is an error logging policy – the requirement to log all errors in a system in a standard format. Error logging necessitates the addition of code that is distributed throughout the system code, even though the concept is easy to state in itself. Cross-cutting concerns explain a significant fraction of the code volume and interdependencies of a system. The interdependencies complicate the understanding, development, and evolution of the system.

Current tool support for AOSD is provided mainly in the form of code-level constructs. We briefly present some ideas on how cross-cutting constraints can be treated in a mechanized specs-to-code refinement process (for more detail, see [9]). Our approach is (1) to express the essential intention of an aspect by a logical invariant, and (2) to generate code to maintain the invariant throughout the system. The generated maintenance code corresponds to statically woven advice in AspectJ, and could be expressed either directly as AspectJ aspects, or by direct generation and insertion of maintenance code into the system. To state invariants that express cross-cutting features often entails the need to reify certain extra-computational values such as history or the runtime call stack.

A simple example serves to introduce the technique: maintaining an error log for a system. More precisely, whenever an exception handler is invoked, we require that an entry be made in an error log. To express this requirement, we first ask: what does the error log mean as a data structure? Informally, the idea is that at any point in time *t*, the error log records a list of all exceptions that have been raised by the program up to time *t*. In order to formalize this we need some way to discuss the history of the program at any point in time.

Maintaining a history variable

The execution history of the program can be reified into the state by means of a *virtual* variable (also called a shadow or ghost variable). That is, imagine that with each action taken by the program there is a concurrent action to update a variable called *hist* that records the history up until the current state

$$s_0 \xrightarrow[hist := hist::\langle s_0, act_0, s_1\rangle]{act_0} s_1 \xrightarrow[hist := hist::\langle s_1, act_1, s_2\rangle]{act_1} s_2 \cdots$$

where state transitions are represented as triples of pre-state, action, post-state. Obviously this would be an expensive variable, but it is only needed for specification purposes, and usually only a residue of it will appear in the executable code.

Invariant

Given the history variable, the expression $map(action, hist)$ represents the sequence of actions so far in the execution history, where $action$ selects the action part of a state transition. To express the invariant, we need a test for whether an action represents an error; i.e. whether it represents the invocation of an exception handler. Let $error?(act)$ be true when act is an exception, so $filter(error?, map(action, hist))$ is the sequence of error actions so far in the execution history.

We can now represent the semantics of the error log as the invariant:

$$\Box\ errlog \ = \ filter(error?, map(action, hist))$$

i.e. in any state, the value of the variable $errlog$ is the sequence of error actions that have occurred previously.

The idea is that the programmer asserts this formula as a requirement on the code. It is a cross-cutting requirement since exceptions can be raised anywhere in the code, regardless of its structure.

Disruptive Code and Static Analysis

In order to enforce the invariance of the asserted formula, we must find all actions in the code that could possibly disrupt the invariant, and then generate new code for maintaining the invariant in parallel with the disruptive action. The set of all code points that could disrupt the invariant corresponds to the AspectJ concept of code points that satisfy a pointcut. The maintenance code that we generate for each such disruptive code point corresponds to a point-specific instance of the advice of an aspect.

Generally, the characterization of disruptive code is based on the Liebniz or substitutivity rule:

$$x = x' \implies I(x) = I(x')$$

where x is the vector of state variables and $I(x)$ is the invariant. The disruptive actions are necessarily those actions in the code that might change the dependent variables of the invariant. A static analyzer would be used to (1) find all actions in the source code that could possibly change the dependent variables of the invariant, and (2) when possible, run inexpensive tests to determine if the invariant is actually violated by the action. For each potentially disruptive action that the static analyzer finds, action-specific maintenance code needs to be generated.

In our example, the dependent variable of the invariant is *hist*, which is changed by every program action. The $error?$ predicate serves as an inexpensive test that an action might violate the invariant. A static analyzer would scan the code (i.e. the abstract syntax representation of the code) looking for all actions that satisfy $error?$.

Specification and Derivation of Maintenance Code

Suppose that act is an action such that $error?(act)$. In order to preserve the invariant, we need to perform a maintenance action that satisfies

assume: $errlog = filter(error?, map(action, hist))$
achieve: $errlog' = filter(error?, map(action, hist'))$

The postcondition can be simplified as follows:

$$errlog' = filter(error?, map(action, hist'))$$

$$\iff \{\text{using the definition of } \textit{hist}\}$$

$$errlog' = filter(error?, map(action, (hist :: \langle \_, act, \_ \rangle)))$$

$$\iff \{\text{distributing } map \text{ over :: }\}$$

$$errlog' = filter(error?, map(action, hist) :: act)$$

$$\iff \{\text{distributing } filter \text{ over ::, using assumption that } error?(act)\ \}$$

$$errlog' = filter(error?, map(action, hist)) :: act$$

$$\iff \{\text{using the precondition/invariant inductively }\}$$

$$errlog' = errlog :: act$$

which is easily satisfied by the simple update

$$errlog := errlog :: act.$$

This maintenance action is to be performed in parallel with act. Again, note that this generated maintenance code corresponds to an instance of an aspect's advice that is applicable where act occurs in the source code.

More generally, suppose that static analysis has identified an action act as potentially disruptive of invariant $I(x)$. If act satisfies the specification

assume : $P(x)$
achieve : $Q(x, x')$

then the maintenance code $maint$ can be formally specified as

assume : $P(x) \wedge I(x)$
achieve : $Q(x, x') \wedge I(x')$

Code for $maint$ often takes the form of a parallel composition

$$act || update$$

of the actions act and $update$. Implicit in this specification is the need to preserve the effect of act while additionally reestablishing the invariant I. If it is inconsistent to achieve both, then the specification is unrealizable.

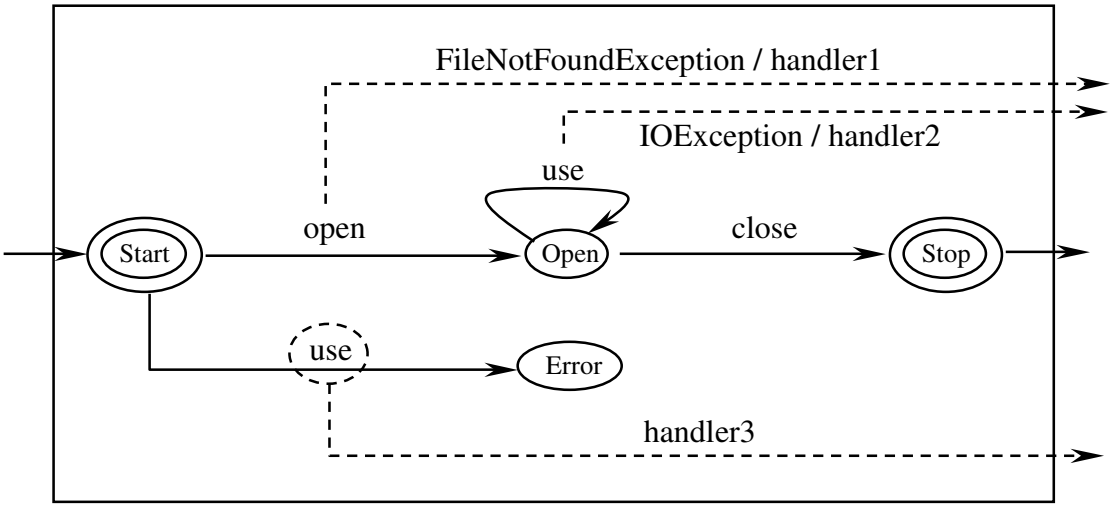
As stated earlier, a policy can be thought of as a modular cross-cutting constraint on a system that helps determine what to do at decision points. Technically, enforcing a policy reduces the nondeterminism in a system, thereby creating a refinement.

The work described in the previous subsection relies on a simple form of policy that can be stated as a logical invariant – a state property that must be maintained. More generally, policies often have a more dynamic character and are better specified by means of automata or temporal logic formulas. To illustrate behavioral policies and their enforcement we briefly present some ideas on error-handling policies. Empirical measures of the amount of code devoted to error-handling in fielded systems vary from a few percent up to two-thirds [2, 11], with the amount increasing with code size and age. 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.

First, what is needed to specify an error-handling policy? Errors are the flip-side to normal behavior, so it seems reasonable to specify error-handling policies in the context of normal-case behavior. 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* (EHPs), or *policies* for short (for more detail, see [10]).

For example, a simplified language-neutral EHP for file management is depicted in Figure 1. 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 exceptions are presented as code templates to be instantiated at design-time by the policy enforcement mechanism. The arrows exiting the EHP 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 EHP. 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. A Java-like syntax for expressing error-handling policies is presented in [10].

The mechanism for enforcing policies has several stages.



Simplified Generic File Management Policy

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.

The second stage is a sound 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.

In the third stage, the policy is actually enforced by inserting code fragments into the system source code at relevant points. If the analysis from the previous stage is ambiguous, then code transformations are run to try to remove the ambiguity. If the result is unambiguous, then error-handlers from the policy can be inserted directly. For example, if an `read` command is issued at a code point that is labeled `open`, then enforcement will ensure that both `EOFExceptions` and `IOExceptions` are caught. If, on the other hand, the analysis is ambiguous (e.g. multithreaded programs will almost always be ambiguous wrt policy state), then the policy will be enforced by runtime tracking of state - if static analysis cannot reveal the policy state, then dynamic analysis must be used. We perform runtime tracking by extending the relevant classes with a state variable that is updated with each method call. Also, method overriding is used to wrap error-handling code around the parent method. For example, when applying the policy in Figure 1 to `DataInputStreams` in Java, we generate the extended class that is stated elliptically in Figure 2. Then every occurrence of class `DataInputStream` is replaced with `DataInputStreamTracking` in the system source.

5 Summary

This paper has aimed to convey a sense of the possibility of requirements-driven mechanized development of code by refinement. We discussed how common kinds of programming can be represented abstractly and formally, and then mechanically applied to an intermediate specification to produce refinement steps. We described in more detail how two kinds of cross-cutting requirement constraints can be formally specified and automatically enforced, again generating refinements. Both techniques rely on scalable static analysis to find all places in the system code that they apply. The correctness of enforcement depends on the soundness of the analysis; i.e. the absence of false negatives, or missed instances.

The incremental refinement process produces code that

```
public class DataInputStreamTracking
     extends DataInputStream {
  public static final int Start = 1;
  public static final int Open  = 2;
  public static final int Closed  = 3;
  int currentState = Start;
  public String filename;

  public DataInputStreamTracking
          (String filename)
         throws FileNotFoundException {
      super(new FileInputStream(filename));
      this.filename = filename;
      this.currentState = Open;
  }

  public int readInteger()
             throws IOException {
      int x = 0;
      switch(currentState){
        case Start:
            throw new Error(
                   "Attempt to read
                   from an unopen File");
        case Open:
            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);
            }
            break;
        case Closed:
            throw new Error(
                    "File "
                    + filename
                    + "already closed");
      }
      return x;
    }
  ...
}
```

Extended Class with Runtime Tracking

is provably consistent with the initial specification. Moreover, a by-product of the development process is a detailed design record along with the code, proofs, and top-level specification. This is crucial information to support further evolution and migration steps.

Acknowledgments: The research reported here has been supported by the Office of Naval Research, the US Air Force Research Lab, the Defense Advanced Research Projects Agency, and the US Department of Defense.

References

[1] *Aspect-Oriented Software Development*, 2003. http://www.aosd.net/.

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

[3] T. Elrad, R. Filman, and A. Bader, editors. *Communications of the ACM– Special Issue on Aspect-Oriented Programming*, volume 44(10). October 2001.

[4] Kestrel Institute. *Specware System and documentation*, 2003. http://www.specware.org/.

[5] D. Pavlovic and D. R. Smith. Composition and refinement of behavioral specifications. In *Proceedings of Sixteenth International Conference on Automated Software Engineering*, pages 157–165. IEEE Computer Society Press, 2001.

[6] D. R. Smith. Constructing specification morphisms. *Journal of Symbolic Computation, Special Issue on Automatic Programming*, 15(5-6):571–606, May-June 1993.

[7] D. R. Smith. Toward a classification approach to design. In *Proceedings of Algebraic Methodology and Software Technology (AMAST)*, volume LNCS 1101, pages 62–84. Springer-Verlag, 1996.

[8] D. R. Smith. Mechanizing the development of software. In M. Broy and R. Steinbrueggen, editors, *Calculational System Design, Proceedings of the NATO Advanced Study Institute*, pages 251–292. IOS Press, Amsterdam, 1999.

[9] D. R. Smith. A generative approach to aspect-oriented programming. In *Proceedings of the Third International Conference on Generative Programming and Component Engineering*, pages 39–54. Springer-Verlag LNCS 3286, 2004.

[10] D. R. Smith and K. Havelund. Automatic enforcement of error-handling policies. Technical report, Kestrel Technology, September 2004.

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