Interprocedural Exception Analysis for C++

Prakash Prabhu[1,2], Naoto Maeda[1,3], Gogul Balakrishnan[1], Franjo Ivančić[1], and Aarti Gupta[1]

[1] NEC Laboratories America, 4 Independence Way, Suite 200, Princeton, NJ 08540
[2] Princeton University, Department of Computer Science, Princeton, NJ 08540
[3] NEC Corporation, Kanagawa 211-8666, Japan

Abstract. C++ Exceptions provide a useful way for dealing with abnormal program behavior, but often lead to irregular interprocedural control flow that complicates compiler optimizations and static analysis. In this paper, we present an interprocedural exception analysis and transformation framework for C++ that captures the control-flow induced by exceptions and transforms it into an exception-free program that is amenable for precise static analysis. Control-flow induced by exceptions is captured in a modular interprocedural exception control-flow graph (IECFG). The IECFG is further refined using a novel interprocedural dataflow analysis algorithm based on a compact representation for a set of types called the Signed-TypeSet domain. The results of the interprocedural analysis are used by a lowering transformation to generate an exception-free C++ program. The lowering transformations do not affect the precision and accuracy of any subsequent program analysis. Our framework handles all the features of synchronous C++ exception handling and all exception sub-typing rules from the C++0x standard. We demonstrate two applications of our framework: (a) automatic inference of exception specifications for C++ functions for documentation, and (b) checking the "no-throw" and "no-leak" exception-safety properties.

1 Introduction

Exceptions are an important error handling aspect of many programming languages, especially object-oriented languages such as C++ and Java. Exceptions are often used to indicate unusual error conditions during the execution of an application (resource exhaustion, for instance) and provide a way to transfer control to special-purpose exception handling code. The exception handling code deals with the unusual circumstance and either terminates the program or returns control to the non-exceptional part of the program, if possible. Therefore, exceptions introduce additional, and often complex, interprocedural control flow into the program, in addition to the standard non-exceptional control flow.

The interprocedural control flow introduced by exceptions necessitate global reasoning over whole program scope, which naturally increases the potential for bugs. Stroustrup developed the notion of exception safety guarantees for components [18]. Informally, exception safety means that a component exhibits *reasonable behavior* when an exception is raised. The term "reasonable" includes all the usual expectations for error-handling: resources should not be leaked, and that the program should remain in a well-defined state so that execution can continue. Stroustrup introduced various degrees of exception safety guarantees that can be expected from components:

- **No leak guarantee**: If an exception is raised, no resources such as memory are leaked.
- **Basic guarantee**: In addition to the no leak guarantee, the basic invariants of components (for example, properties that preserve data structure integrity) are maintained.
- **Strong guarantee**: In addition to the basic guarantee, this requires that an operation either succeeds or has no effect, if an exception is raised.
- **No throw guarantee**: In addition to the basic guarantee, this requires that an operation is guaranteed not to raise an exception.

However, it is very difficult to ensure such exception-safety properties, because developers may overlook exceptional control-flow hidden behind multiple levels of abstraction. For instance, in a code block containing local objects as well as exceptions, programmers have to reason about non-local returns induced by exceptions, and at the same time understand the effects of the implicit calls to the destructors of local objects along the exception path correctly. Unlike Java, all C++ exceptions are unchecked, and library developers are not required to annotate interfaces with exception specifications. Furthermore, dynamic exception specifications (anything other than `noexcept` specification) are deprecated in the latest C++0x draft standard [17]. Consequently, developers increasingly rely on documentation to discern throwable exceptions from a function interface (more so in the absence of source code), which makes it hard to reason about programs that use library functions that throw exceptions. Therefore, a tool that automatically models the behavior of exceptions precisely would be useful.

Existing Approaches to C++ Exceptions. Program analysis techniques, both static and dynamic, are often applied in the context of program optimization, automatic parallelization, program verification, and bug finding. These techniques rely heavily on both intraprocedural and interprocedural control flow graph information, which are utilized to compute relevant information as needed (e.g., dependence analysis or program slicing). However, existing compiler frameworks for C++ (for example g++, clang/LLVM [10]) do not build precise models for exceptions. Specifically, they only analyze exceptional control flow within a locally declared *trycatch* statement, and do not perform either an intraprocedural or interprocedural analysis. Therefore, they make conservative assumptions about interprocedural control flow, which causes their models to include paths between *throw* statements and *catch* blocks that are infeasible at runtime. An alternative approach is to use such frameworks to generate a semantically equivalent C program from the given C++ program and use the lowered C code for further analysis. However, the code generated by these tools use custom data structures and involves calls into opaque runtimes, which need to be modeled conservatively in static analysis algorithms, resulting in further loss in precision.

Our Approach. In this paper, we present an interprocedural exception analysis and transformation framework for C++ that (1) captures the control-flow induced by exceptions precisely, and (2) transforms the given C++ into an exception-free program that is amenable for precise static analysis. We summarize our contributions below:

– We propose a *modular* abstraction for capturing the interprocedural control flow induced by exceptions in C++, called the interprocedural exception control flow graph (IECFG). The IECFG is constructed through a sequence of steps, with each step refining it. The modular design of IECFG is motivated by the need to model implicit calls to destructors during stack unwinding, when an exception is thrown. The modularity of IECFG is also important in practice, for permitting re-use in presence of separate compilation units.
– We design and implement an interprocedural exception analysis algorithm to model the set of C++ exceptions that reach *catch* statements in the program using the *Signed-TypeSet* domain, which represents a set of program types compactly. Our analysis is formulated in conjunction with the construction of the IECFG. A unique feature of our framework is the capability to safely terminate the IECFG construction at certain well-defined points during interprocedural propagation, thereby, allowing clients, such as optimizing compilers or program analysis, to trade-off speed over precision.
– We present a lowering algorithm that uses the results of our exception analysis to generate an exception-free C++ program. Unlike standard compilers, our algorithm does not use non-local jumps or calls into any opaque C++ runtime systems. Absence of an external runtime and non-local jumps enables existing static analyses and verification tools to work soundly over C++ programs with exceptions, without needing to model them explicitly within their framework. While the IECFG construction is modular in the sense of allowing separate compilation units, the lowering algorithm to generate exception-free code is not modular. It requires a global view of all source code under analysis so that all known possible targets of virtual function calls can be determined.
– We present the results of using our interprocedural exception analysis and transformation framework on a set of C++ programs. We compute the exception specifications for functions and check the related "no throw" guarantee. We also check the "no leak" exception-safety property.

Example. Consider the C++ program shown in Fig. 1. The program has three functions, of which `get()` allocates a `File` object and attempts to read a line from `File` by calling `readLine()`. If the file does not exist, `readLine()` throws an `IOException` that is handled in the `get()` function. Otherwise, a call is made to `read()` which throws an `EOFException` if the end of the file is reached, which is handled in `readLine()`. An exception modeling framework has to abstract the interprocedural control flow due to exceptions correctly, and also take into account the implicit calls made to destructors during stack unwinding, when an exception propagates out of a function (e.g., destruction of `str` in `read()` when `EOFException` is thrown).

```
string File::read() {
 string str(__line);
 if (EOF)
   throw EOFException();
 return str;
}
```

```
class EOFException { ... };
class IOException { ... };
```

```
void get () {
 string s;
 try {
  File *file = new File("1.txt");
  file->readLine();
  delete file;
 }
 catch (IOException& ie) {
  cout<< "IO-Failure";
 }
 return;
}
```

```
string File::readLine() {
 string s;
 try {
  if (invalidFile)
   throw IOException();
  s = read();
  return s;
 }
 catch (EOFException& e) {
  return string("");
 }
}
```

Fig. 1. Running Example

There are two bugs worth noting in this example, both of which have to do with exceptions: (1) violation of "no leak" guarantee, the `file` object gets leaked along the exception path from `readLine()` to the `catch` block in `get()`, and (2) violation of "no throw" guarantee, a potential `std::bad_alloc` exception thrown by `new` is not caught in `get()`. Our exception analysis and transformation framework enables checking these properties easily. ■

Comparison with Java Exception Analysis. Several analysis approaches for modeling Java exceptions have been proposed in the recent past. Most approaches [3, 8, 9] compute an interprocedural exception control flow graph as we do. There have been some attempts to analyze the "no leak" exception-safety guarantee for Java programs also [11]. However, there are a number of major differences between exception handling in Java and C++, which require different design decisions in comparison to Java-based exception analysis techniques:

1. In C++, when an exception propagates out of a function, destructors are invoked on all stack-allocated objects between the occurrence of the exception and the catch handler in a process called *stack unwinding.* Stack unwinding in C++ is a major difference compared to Java, and raises various performance issues, along with complicating the modeling of exceptional control-flow.
2. C++ destructors can call functions which throw exceptions internally, leading to a scenario where multiple exceptions are live during stack unwinding. Unlike implicitly invoked destructors, Java provides "finalizers", that are invoked non-deterministically by the garbage collector. Although the use of "finally" blocks in Java can result in multiple live exceptions, these blocks are created and controlled explicitly by the programmer, and therefore, multiple live exceptions in Java are apparent from the code itself.
3. The exception subtyping rules for Java are limited to only parent-child relationships within the class hierarchy. Besides, all exception classes trace their

lineage to a single ancestor, the `Exception` class. In contrast, C++ exception subtyping rules are richer and include those concerning multiple inheritance, reference types, pointers, and few other explicit type-conversion rules among functions as well as arrays.

4. The exception specification and checking mechanism in Java is much stronger than in C++. In particular, Java has a "checked exception" category of exceptions, which explicitly requires programmers to either catch exceptions thrown within a function, or declare them as part of the interface. C++ has no concept of checked exceptions, and the dynamic exception specifications are deprecated in the latest C++0x standards draft. Exception specifications in C++ may not even be accurate, which results in a call to `std::unexpected()` function, which may be redefined by an application.
5. C++ provides an exception probing API while Java does not. It provides a means to conditionally execute code depending on whether there is an outstanding live exception by calling `std::uncaught_exception()`. This can be used to decide whether or not to throw exceptions out of a destructor. C++ also allows users to specify abnormal exception termination behavior by providing custom handlers for `std::terminate()` or `std::unexpected()`.
6. Java exceptions are handled based on runtime types, whereas in C++ static type information is used to decide which catch handler is invoked. Therefore, pointer analysis is required in Java to improve the accuracy of matching throw statements with catch blocks. For C++, we can avoid a heavy duty pointer analysis for exceptions. (However, call graph construction in C++ can be improved with the results of a pointer analysis on function pointers and virtual function calls.)

2 Preliminaries

We first describe the abstract syntax of a simplified intermediate language (IL) for C++ used within our framework. The language is based on CIL [14], with additional constructs for object-oriented features. Fig. 2 shows the subset of the actual IL that is relevant for the exception analysis and transformation framework. Types within our IL include the primitive ones (*int, float, void*) as well as user defined classes (*cl*), derived types (pointer, reference and array), function types, and can additionally be qualified (*const, volatile, restrict*). Each class type can inherit from a set of classes, and has a set of fields and member functions, some of which may be *virtual*. Visibility of the class members and the inherited classes is controlled by an access specifier.

A program is a set of globals. A global is either a type or a function. A function has a signature and a body, which is a block of statements. Statements include instructions, regular control flow statements (*loop*, *if*, *trycatch*), irregular control flow statements causing either local (*goto*, *break*, *continue*) or global (*throw*, *return*) alterations. An instruction is one of the following: an assignment, an allocation operator, a deallocation operator, a global function call, or a member function call. Expressions could involve binary operators, unary operators, pointer dereferences or indirections, reference indirections, and cast operations.

Constant	c	$\in$	$\mathit{Constant}$
Identifiers	id	$\in$	$\mathit{Identifier}$
Labels	l	$\in$	Label
Access	a	$::=$	$\mathit{private} \mid \mathit{protected} \mid \mathit{public}$
Qualifier	cv	$::=$	$\mathit{const} \mid \mathit{volatile} \mid \mathit{restrict}$
Class	cl	$::=$	$\mathit{class}\ id\ :\ \overline{a\ t}\ \{\overline{a\ t\ fi};\overline{a\ \mathit{virtual}?\ m}\}$
Type	t	$::=$	$id \mid t* \mid t\ \& \mid t\,[e] \mid t \rightarrow t \mid \mathit{void} \mid \mathit{int} \mid \mathit{float} \mid \overline{cv}\ t$
Variable	v	$::=$	id
Lvalue	lv	$::=$	$lh\ e$
Lhost	lh	$::=$	$v \mid *\ e$
Program	p	$::=$	$\overline{g}$
Global	g	$::=$	$t\ id \mid f$
Function	f	$::=$	$t\ id\ (\overline{t\ id}) = b$
Block	b	$::=$	$\{\overline{s}\}$
Statement	s	$::=$	$i \mid \mathit{return}\ e? \mid \mathit{goto}\ l \mid \mathit{break} \mid \mathit{continue} \mid$
			$\mathit{if}\ e\ b_1\ b_2 \mid \mathit{loop}\ b \mid \mathit{throw}\ e? \mid \mathit{trycatch}\ b\ \overline{h}$
Handler	h	$::=$	$(t\ v)\ b \mid (...)\ b$
Instruction	i	$::=$	$\mathit{call}\ id\ e_f\ \overline{e} \mid \mathit{mbrcall}\ id\ e_{this}\ e_f\ \overline{e} \mid e\ :=\ e \mid v\ :=\ \mathit{new}\ e \mid \mathit{delete}\ e$
Cast	$cast$	$::=$	$\mathit{staticcast} \mid \mathit{dyncast} \mid \mathit{constcast} \mid \mathit{reintcast}$
Expression	e	$::=$	$c \mid lv \mid \mathit{unop}\ e \mid e\ \mathit{binop}\ e \mid \mathit{cast}\ t\ e \mid \&lv \mid lv\&$

Fig. 2. Abstract Syntax of the Simplified IL for C++

C++ Exceptions. C++ exceptions are *synchronous*. Asynchronous exceptions, which in Java are raised due to internal errors in the virtual machine, are categorized as program errors in C++ and are not handled by the exception constructs of C++. Synchronous exceptions, in contrast, are expected to be handled by the programmer and are only thrown by certain statements in the program, such as (a) *throw* statement, which throws a fresh exception or rethrows a caught exception, (b) function call, which transitively throws exceptions uncaught within its body or its callees, (c) *new* operator, which can throw a `std::bad_alloc` exception, and (d) *dynamic_cast*, which can throw a `std::bad_cast` exception.

Exception Handling and Subtyping rules for C++. Exceptions in C++ are caught using exception handlers, defined as part of the *trycatch* statement. Each *trycatch* statement has a single *try* block followed by a sequence of exception (*catch*) handlers. An exception object thrown from within the *try* block is caught by the first handler whose declared exception type matches the thrown exception type according to the C++ exception subtyping rules. If no match is found for a thrown object amongst the handlers, control flows either to an enclosing *trycatch* statement or out of the function to the caller.

The exception subtyping rules for C++ as defined in the final C++0x draft standard [17] are shown in Fig. 3. The type of a thrown exception is given by t_T and the type declared in the exception handler is given by t_C in each rule. A handler is a match for an exception, if any of the following conditions hold:

– The handler's declared type is the same as the type of the exception, even when ignoring the const-volatile qualifiers (Rules **EQ** and **CVQUAL**).
– The handler's declared type is an unambiguous public base class of the exception type. Arrays are treated as pointers and functions returning a type are treated as pointers to function returning the same type (Rules **SUBCL**, **ARR**, **FPTR** and **CVQUAL**).
– The handler's declared type is a reference to the exception type (Rule **REF**).

$$\frac{t_T = t \quad t_C = t}{t_T \leq t_C}\ [\textbf{EQ}]$$

$$\frac{t_T = cv\ t \quad t_C = t}{t_T \leq t_C}\ [\textbf{CVQUAL}]$$

$$\frac{t_T = t_1 \quad t_C = t_2 \quad t_1 \in sub(t_2)}{t_T \leq t_C}\ [\textbf{SUBCL}]$$

$$\frac{t_T = t_1 \quad t_C = t_2[] \quad t_1 \leq t_2*}{t_T \leq t_C}\ [\textbf{ARR}]$$

$$\frac{t_T = t_1 \quad t_C = (\_ \rightarrow t_2) \quad t_1 \leq (\_ \rightarrow t_2)*}{t_T \leq t_C}\ [\textbf{FPTR}]$$

$$\frac{t_T = t \quad t_C = t\&}{t_T \leq t_C}\ [\textbf{REF}]$$

$$\frac{t_T = t_1 * \quad t_C = t_2 * \quad t_1* \leq_{conv} t_2*}{t_T \leq t_C}\ [\textbf{PTR}]$$

$$\frac{t_T = std::nullptr\_t \quad t_C = t*}{t_T \leq t_C}\ [\textbf{NULLPTR}]$$

$$\frac{t_T = t_1 * \quad t_C = void*}{t_T \leq t_C}\ [\textbf{VOID}]$$

Fig. 3. Exception Subtyping Rules

- The handler's declared type is a pointer into which the exception type, which also is a pointer, can be converted using C++ pointer conversion rules (Rule **PTR**).
- The two remaining rules concern generic pointers modeled by `void` $*$ and `std::nullptr_t` [17] (Rules **NULLPTR** and **VOID**).

3 Signed-TypeSet Domain

In this section, we present a novel abstract domain for compactly representing a set of program types, which we call the *Signed-TypeSet* domain.

Definition 1. *The Signed-TypeSet domain Γ is defined as: $\Gamma = \{(s, T_{prog}) \mid s \in \{+,-\}, T_{prog} \subseteq \{t \mid t \text{ is a program exception type}\}\}$*

The semantics of a positive set of exception types is the standard one, while a negative set of exception types represents "*every exception type other than those in the set*". For instance $(+, \{IOException\})$ represents the *IOException* program type alone, while $(-, \{IOException, EOFException\})$ represents any exception type other than *IOException* and *EOFException*. Exceptions thrown by unknown library calls are modeled concisely as $(-, \{\})$. For external library calls, a special unknown exception type $t_{unknown}$ is introduced explicitly only at the point when the lowering transformation is to be done.

We use a signed domain in our framework, rather than a domain of only positive set of program types to make the IECFG computation modular. Its use is especially beneficial in the presence of unknown library calls and deprecated use of dynamic exception specifications in C++. A negated set of types succinctly captures the unknown exceptions that could potentially be thrown by opaque library calls that are not caught. We would also like to incrementally integrate the results of exception analysis from separately compiled functions whenever available, while at the same time maintaining a safely analyzable exception result at all intermediate points. Therefore, our exception dataflow analysis begins with an over-approximation of the set of all exception types that could be raised by a throwable statement, and refines the set via interprocedural propagation. This

```
Algorithm 1: union ($\cup_\Gamma$)
Input: $\tau_a \in \Gamma, \tau_b \in \Gamma$
Output: $\tau_c \in \Gamma$
case $\tau_a = (-, T_a) \wedge \tau_b = (-, T_b)$
    $\tau_c = (-, T_c)$ where $T_c = \{t \mid t \in T_a \wedge t \in T_b\}$;
case $\tau_a = (-, T_a) \wedge \tau_b = (+, T_b)$
    $\tau_c = (-, T_c)$ where $T_c = \{t \mid t \in T_a \wedge t \notin T_b\}$;
case $\tau_a = (+, T_a) \wedge \tau_b = (-, T_b)$
    $\tau_c = (-, T_c)$ where $T_c = \{t \mid t \in T_b \wedge t \notin T_a\}$;
case $\tau_a = (+, T_a) \wedge \tau_b = (+, T_b)$
    $\tau_c = (+, T_c)$ where $T_c = \{t \mid t \in T_a \vee t \in T_b\}$;
```

```
Algorithm 2: intersection ($\cap_\Gamma$)
Input: $\tau_a \in \Gamma, \tau_b \in \Gamma$
Output: $\tau_c \in \Gamma$
case $\tau_a = (-, T_a) \wedge \tau_b = (-, T_b)$
    $\tau_c = (-, T_c)$ where $T_c = \{t \mid t \in T_a \vee t \in T_b\}$;
case $\tau_a = (-, T_a) \wedge \tau_b = (+, T_b)$
    $\tau_c = (+, T_c)$ where $T_c = \{t \mid t \in T_b \wedge t \notin T_a\}$;
case $\tau_a = (+, T_a) \wedge \tau_b = (-, T_b)$
    $\tau_c = (+, T_c)$ where $T_c = \{t \mid t \in T_a \wedge t \notin T_b\}$;
case $\tau_a = (+, T_a) \wedge \tau_b = (+, T_b)$
    $\tau_c = (+, T_c)$ where $T_c = \{t \mid t \in T_b \wedge t \in T_a\}$;
```

```
Algorithm 3: set difference ($-_\Gamma$)
Input: $\tau_a \in \Gamma, \tau_b \in \Gamma$
Output: $\tau_c \in \Gamma$
case $\tau_a = (-, T_a) \wedge \tau_b = (-, T_b)$
    $\tau_c = (+, T_c)$ where $T_c = \{t \mid t \in T_b \wedge t \notin T_a\}$;
case $\tau_a = (-, T_a) \wedge \tau_b = (+, T_b)$
    $\tau_c = (-, T_c)$ where $T_c = \{t \mid t \in T_a \vee t \in T_b\}$;
case $\tau_a = (+, T_a) \wedge \tau_b = (-, T_b)$
    $\tau_c = (+, T_c)$ where $T_c = \{t \mid t \in T_b \wedge t \in T_a\}$;
case $\tau_a = (+, T_a) \wedge \tau_b = (+, T_b)$
    $\tau_c = (+, T_c)$ where $T_c = \{t \mid t \in T_a \wedge t \notin T_b\}$;
```

```
Algorithm 4: equals ($=_\Gamma$)
Input: $\tau_a \in \Gamma, \tau_b \in \Gamma$
Output: bool
case $(\tau_a = (-, T_a) \wedge \tau_b = (-, T_b)) \vee (\tau_a = (+, T_a) \wedge \tau_b = (+, T_b))$
    if $(T_a \subseteq T_b) \wedge (T_b \subseteq T_a)$ then
        true
    end
    else
        false
    end
case default
    false
```

Fig. 4. Operations on the Signed-TypeSet domain

deliberate design decision allows clients to terminate the analysis at any point during the analysis and safely use the refined IECFG structure at that point for other analyses.

We define set operations on Γ, that are as efficient as the set operations on normal sets. These operations (shown in Fig. 4) mimic the normal set union, intersection, difference, and equality operations. Given two elements τ_a and τ_b from Γ, these operations result in another element τ_c in Γ. All the operations perform a case analysis on the signs of the two elements, and perform normal set operations on the constituent set of program types. The operations are fairly straightforward, and in their full generality need to take into account the exception subtyping rules described earlier. For the sake of conceptual simplicity, we assume that the set of exception types arising at the catch blocks have already been expanded to contain all possible "exception subtypes" in the program, before doing the specific set operations. However, in our implementation, we perform these operations without doing full expansion and correctly account for exception subtypes on demand.

4 Intraprocedural Exception Control Flow Graph

An intraprocedural exception control flow graph is defined as follows:

Definition 2. *An intraprocedural exception control flow graph (ECFG) for a function f, denoted by G_{intra_f} is a tuple $\langle N, E_{reg}, E_{excep}, E_{exceps}, E_c, n_s, n_e, n_{excepe}\rangle$ with Signed-TypeSet domain Γ, where*

– *N is the set of nodes in the graph, consisting of the following distinct subsets:*
$$N = N_{reg} \cup N_c \cup N_{cret} \cup N_{ecret} \cup N_{throw} \cup N_{catch} \cup \{n_s, n_e, n_{excepe}\}$$
where
  - *N_{reg} is the set of regular nodes.*
  - *N_c is the set of call nodes.*
  - *N_{cret} is the set of call-return nodes.*
  - *N_{ecret} is the set of exceptional-call-return nodes.*
  - *N_{throw} is txhe set of throw, new, or dynamic_cast nodes.*
  - *N_{catch} is the set of header nodes of catch blocks.*
  - *n_s, n_e, n_{excepe} are unique start, exit and exceptional-exit nodes.*

– *E_{reg} is the set of regular control flow edges: $E_{reg} \subseteq (N_{reg} \cup N_{cret} \cup N_{catch}) \times N$*

– *E_{excep} is the set of intraprocedural exception control flow edges:*
$$E_{excep} \subseteq ((N_{ecret} \cup N_{throw}) \times (N_{catch} \cup \{n_{excepe}\}) \times \Gamma)$$

– *E_{exceps} is the set of exception-call-summary edges:*
$$E_{exceps} \subseteq (N_c \times N_{ecret} \times \Gamma)$$

– *E_c is the set of normal call-summary edges:*
$$E_c \subseteq N_c \times N_{cret}$$

Example (ECFG structure). The ECFGs for the functions in our running example are shown in Fig. 5. Consider the ECFG for the `get()` function. In addition to the start (*s-get*) and exit (*e-get*) nodes present in regular CFGs, the ECFG has a new exceptional-exit (*exe-get*) node. Control flows through an exceptional-exit node, every time an exception propagates out of a function. Each call instruction is represented by three nodes: in addition to the call node (*c-readLine*) and call-return node (*cr-readLine*) present in regular CFGs, the ECFG has a new exceptional-call-return node (*ecr-readLine*) through which control flows when the callee terminates with an exception. The ECFG has two additional exception-related nodes: throw node, one for every potential throwing statement, and catch-header node, one for every catch block in the code.

There are three kinds of edges in a ECFG: (a) normal control-flow edge (solid lines) as in any CFG, (b) normal call-summary edges (long-dashed lines) between call and call-return nodes, and (c) exception edges (short-dashed lines) which can either be a summary edge (between call and exceptional-call-return nodes) or a normal exception edge (for intraprocedural exceptions). Every exception edge is annotated with an element from the Signed-TypeSet domain Γ. The exception annotations represent the dataflow facts used in our interprocedural exception analysis, as described in Sect. 5. ■

ECFG Construction. The algorithm to construct the ECFG of a program performs a post-order traversal on the abstract syntax tree (AST) of the C++ code, creating a set of ECFG nodes and edges as it visits each AST node. Each *visit* method returns a triple $\langle N_b, N_e, N_{unres}\rangle$, where N_b and N_e represent the set of nodes corresponding to start and normal exit of the ECFG "region" corresponding to the current AST node. N_{unres} represents the set of nodes in an ECFG region that has some unresolved incoming or outgoing edges, which are resolved by an ancestor's visitor. For instance, a *throw* node that is not enclosed

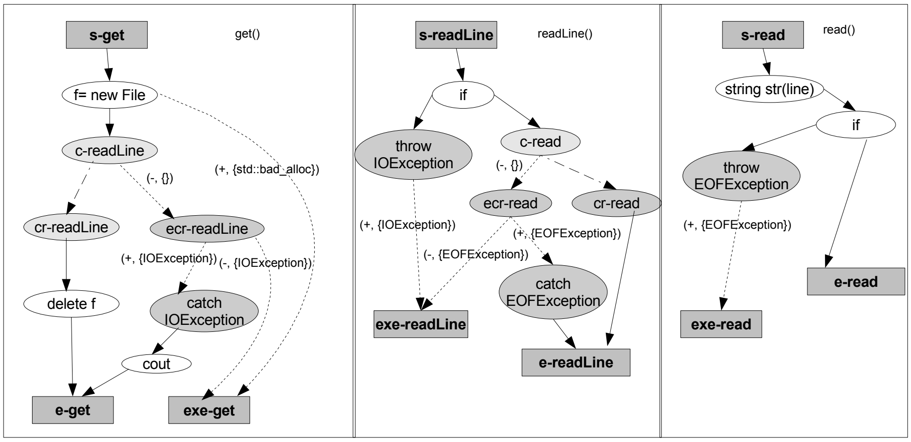

Fig. 5. Intraprocedural Exception Control Flow Graphs for the Program in Fig. 1

within a *trycatch* block is resolved at the root (function declaration), by creating an exception edge to the exceptional-exit node. Alg. 5 shows the visit routine for a *trycatch* statement.

The *VisitTryCatchStmt* routine for *trycatch* statements works as follows: it first constructs the ECFG nodes and edges for the *try* block and all the catch handlers, by visiting them recursively (Lines 1-4). It then divides the exception-related ECFG nodes in the *try* block into two sets: (a) the throw nodes[1], and (b) the exceptional-call-return nodes (Lines 5-7). For the throw nodes, a match is sought in the sequence of handlers by applying the C++ exception subtyping rules described earlier. As soon as the first match is found, an exception edge is created from the *throw* node to the *catch* header node, annotated with the appropriate exception information from the Signed-TypeSet domain (Lines 8-18). For the throw nodes, this information is always a positive set with a singleton exception type, which is the type of the throw expression.

For an exceptional-call-return node n_{ecret}, a map ECR_{Γ} is used during the search for an exception-type match amongst the handlers. ECR_{Γ} maps an exceptional-call-return node to an element from the Signed-TypeSet domain. Initially, ECR_{Γ} holds "all exception types" $((-, \{\}))$, which represents the most conservative assumption as far as possible exceptions thrown from a call are concerned, for all exceptional-call-return nodes. Every time a catch block is encountered and a possible match occurs, $ECR_{\Gamma}(n_{ecret})$ is incrementally updated to hold the "remaining exception types" that could be thrown from this call (using the difference operator $-_{\Gamma}$). The final value of $ECR_{\Gamma}(n_{ecret})$ is used to annotate the edge between the n_{ecret} and the exceptional-exit node of the function. At every match with a catch header node, an exception edge is created from

[1] In C++ programs, `new` and `dynamic_cast` operators may also throw `bad_alloc` and `bad_cast` exceptions, respectively. For sake of clarity, Alg. 5 only considers `throw`. Our implementation deals with `new` and `dynamic_cast` operators properly.

```
Algorithm 5: VisitTryCatchStmt
Input: $s$: A trycatch statement where $s = b_1\ (h_1, h_2, ..., h_k)$
Output: $N_b \times N_e \times N_{unres}$
let $(N_{b_1}, N_{e_1}, N_{unres_1}) = VisitBlock(b_1)$; // Creates ECFG region for a block
foreach $h_i$ do
    let $(N_{b_{h_i}}, N_{e_{h_i}}, N_{unres_{h_i}}) = VisitHandler(h_i)$ // Creates ECFG region for a handler
end
let $S_{throw} = \{n \mid n \in N_{unres_l} \wedge ir(n) = throw\}$;
let $S_{ecret} = \{n \mid n \in N_{unres_l} \wedge ir(n) = call \wedge n \in N_{ecret}\}$;
let $S_{rest} = N_{unres_1} - (S_{throw} \cup S_{ecret})$;
foreach $n_{th} \in S_{throw}$ do
    for $i$=1 to $k$ do
        $e_{T_{catch}} = (+, \{t\})\ where\ h_i = (t\ v)\ b$;
        $t_{catch} = t\ where\ h_i = (t\ v)\ b$;
        $n_{catch_i} = n\ where\ (n \in N_{b_{h_i}}) \wedge (ir(n) = catch)$;
        $\tau_{throw} = (+, \{t'\})\ where\ (ir(n_{th}) = throw\ e) \wedge (T_{prog}(e) = t')$;
        if $t' \leq t_{catch}$ then
            $E_{excep} = E_{excep} \cup \{(n_{th}, n_{catch_i}, \tau_{throw})\}$;
            $N_{done} = N_{done} \cup \{n_{th}\}$;
            continue: foreach outer loop
        end
    end
end
foreach $n_{ecret} \in S_{ecret}$ do
    for $i$=1 to $k$ do
        $\tau_{catch} = (+, \{t\})\ where\ h_i = (t\ v)\ b$;
        $\tau_{ecret} = ECR_{\Gamma}(n_{ecret})$;
        $\tau_{intersect} = \tau_{ecret} \bigcap_{\Gamma} \tau_{catch}$;
        $\tau_{remain} = \tau_{ecret} -_{\Gamma} \tau_{catch}$;
        $ECR_{\Gamma}(n_{ecret}) = \tau_{remain}$;
        $E_{excep} = E_{excep} \cup \{(n_{ecret}, n_{catch_i}, \tau_{intersect})\}$;
    end
end
$N_{reg} = N_{reg} \cup \{n_{join}\}$;
$E_{reg} = E_{reg} \cup \{(n, n_{join}) \mid n \in N_{e_1}\} \cup \{(n, n_{join}) \mid \exists\ i\ such\ that\ n \in N_{e_{h_i}}\}$;
$N_b = N_{b_1}$; $N_e = \{n_{join}\}$; $N_{unres} = (\cup_{1 \leq i \leq k} N_{unres_{h_i}} \cup N_{unres_1}) - N_{done}$;
```

Fig. 6. *VisitTryCatch* routine for ECFG construction

the exceptional-call-return node to the catch header, annotated with appropriate exception information (Lines 20-28). The *VisitTryCatchStmt* routine creates a header node (n_{try}) and a join (n_{join}) node, in addition to those created by its children (Lines 30-32).

The handling of the exceptional-call-return node illustrates one distinguishing feature of our approach, as compared to other exception analysis algorithms proposed for Java: the dataflow facts are initialized with an over-approximation, which can be refined using the exception information from the catch-header node. Our choice of the Signed-TypeSet domain, which allows a negative set of exception types, not only permits modeling of unknown library calls within the same framework, but also permits a safe termination of our analysis at any point after the construction of the intraprocedural exception flow graphs.

Figs. 7, 8, and 9 show the *Visit* routines for the remaining IL constructs. The *VisitBlock* routine is straightforward, recursively visiting each child statement and then creating edges between corresponding nodes. The *VisitStmt* routine creates ECFG nodes and edges differently based on the type of AST node. For *if* and *loop* statements, it creates ECFG regions with appropriate join and header

Algorithm 6: *VisitBlock*

```
   Input: $b$: Block where $b = s_1, s_2, ..., s_n$
   Output: $N_b \times N_e \times N_{unres}$
1  for $i = 1$ to $n$ do
2      let $(N_{b_i}, N_{e_i}, N_{unres_i}) = VisitStmt(s_i)$;
3  end
4  $E_{reg} = E_{reg} \cup \{(n_1, n_2) \mid \forall i \cdot 1 \leq i < n \cdot (n_1 \in N_{e_i} \wedge n_2 \in N_{b_{i+1}})\}$;
5  $N_b = N_{b_1}; N_e = N_{e_n}; N_{unres} = \cup_{1 \leq i \leq n} N_{unres_i}$;
```

Algorithm 7: *VisitStmt*

```
    Input: $s$: Statement
    Output: $N_b \times N_e \times N_{unres}$
1   switch typeOf(s) do
2       case $instr \in \{call, mbrcall\}$
3           $VisitCallInstr(i)$;
4       case $instr \notin \{call, mbrcall\}$
5           $N_{reg} = N_{reg} \cup \{n_i\}; N_b = N_e = \{n_i\}; N_{unres} = \{\}$
6       case $break \mid continue \mid goto \mid return$
7           $N_{reg} = N_{reg} \cup \{n_s\}; N_b = N_{unres} = \{n_s\}; N_e = \{\}$;
8       case $if\ e\ b_1\ b_2$
9           let $(N_{b_1}, N_{e_1}, N_{unres_1}) = VisitBlock(b_1)$;
10          let $(N_{b_2}, N_{e_2}, N_{unres_2}) = VisitBlock(b_2)$;
11          $N_{reg} = N_{reg} \cup \{n_{if_e}, n_{join}\}$;
12          $E_{reg} = E_{reg} \cup \{(n_{if_e}, n_{b_1}) \mid n_{b_1} \in N_{b_1}\} \cup \{(n_{if_e}, n_{b_2}) \mid n_{b_2} \in N_{b_2}\}$
13                  $\cup \{(n_{e_1}, n_{join}) \mid n_{e_1} \in N_{e_1}\} \cup \{(n_{e_2}, n_{join}) \mid n_{e_2} \in N_{e_2}\}$;
14          $N_b = \{n_{if_e}\}; N_e = \{n_{join}\}; N_{unres} = N_{unres_1} \cup N_{unres_2}$;
15      case $loop\ b_l$
16          let $(N_{b_l}, N_{e_l}, N_{unres_l}) = VisitBlock(b_l)$;
17          let $S_{continue} = \{n \mid n \in N_{unres_l} \wedge ir(n) = continue\}$;
18          let $S_{break} = \{n \mid n \in N_{unres_l} \wedge ir(n) = break\}$;
19          $E_{reg} = E_{reg} \cup \{(n_c, n_{l_{head}}) \mid n_c \in S_{continue}\} \cup \{(n_b, n_{l_{exit}}) \mid n_b \in S_{break}\}$ ;
20          $N_{reg} = N_{reg} \cup \{n_{l_{head}}, n_{l_{exit}}\}$;
21          $N_b = \{n_{l_{head}}\}; N_e = \{n_{l_{exit}}\}; N_{unres} = N_{s_1} - (S_{continue} \cup S_{break})$;
22      case $throw \mid new \mid dynamic\_cast$
23          $VisitThrowingStmt(s)$;
24      case $trycatch$
25          $VisitTryCatchStmt(s)$;
26
27  endsw
```

Fig. 7. *VisitBlock* and *VisitStmt* routines for ECFG Construction

nodes (Lines 8-21), while for some of the non-exception unstructured control flow statements like *break*, *goto*, it creates unresolved nodes (Lines 6-7), which are patched later on by their parents.

The *VisitCallInstr* routine creates three nodes and two edges for a call instruction. The three nodes are a call node n_{c_i}, a call-return node (n_{ecret_i}) and an exceptional-call-return node (n_{ecret_i}). The two edges are a summary edge connecting n_{c_i} with n_{ecret_i}, and an exceptional-summary edge connecting n_{c_i} with n_{ecret_i}. The exceptional-summary edge is annotated with the most approximate element $(-, \{\})$, which represents any exception type. The exception return node n_{ecret_i} is unresolved since its targets are determined by the enclosing *trycatch* statements or by the exceptional-exit-node at function scope.

The *VisitThrowingStmt* routine creates an unresolved node for the *throw* statement, while the *VisitHandler* routine creates a header node corresponding to the *catch* block, and connects it to the nodes created for the statements within the block. The *VisitFunction* routine is the main driver for creating the

```
Algorithm 8: VisitCallInstr
Input: $i$: A $call/mbrcall$ Instruction
Output: $N_b \times N_e \times N_{unres}$
1 $N_c = N_c \cup \{n_{c_i}\}$ ; $N_{cret} = N_{cret} \cup \{n_{cret_i}\}$ ; $N_{ecret} = N_{ecret} \cup \{n_{ecret_i}\}$ ;
2 let $\tau_e = (-, \{\})$ ;
3 $E_c = E_c \cup \{(n_{c_i}, n_{cret_i})\}$ ; $E_{excep} = E_{excep} \cup \{(n_{c_i}, n_{ecret_i}, \tau_e)\}$;
4 $N_b = \{n_{c_i}\}; N_e = \{n_{cret_i}\}; N_{unres} = \{n_{ecret_i}\}$ ;
```

```
Algorithm 9: VisitThrowingStmt
Input: $s$: A $throw/new/dynamic\_cast$ statement
Output: $N_b \times N_e \times N_{unres}$
1 $N_{throw} = N_{throw} \cup \{n_s\}$;
2 $N_b = N_{unres} = \{n_s\}; N_e = \{\}$ ;
```

Fig. 8. *VisitCallInstr* and *VisitThrowingStmt* routines for ECFG construction

ECFG for a function. Once the ECFG region for the function body is created, this routine connects the *return* nodes to the normal exit nodes and *throw* nodes to the exceptional-exit node. It finally resolves the unmatched exceptional-call-return nodes by connecting them to the exceptional-exit node, annotated with appropriate exception type annotation.

Example. In Fig. 5, the ECFG for `get()` has an exceptional-call-return node for `readLine()`. The algorithm creates an exception edge from this node to a catch header that handles `IOException` exceptions and annotates the edge with $(+, \{IOException\})$. The algorithm then creates an exception edge to the exceptional-exit node of `get()` annotated with $(-, \{IOException\})$, which is meant to read "If `readLine()` throws any exception other than `IOException`, control is transferred to the exceptional exit node of `get()`". ■

5 Interprocedural Exception Analysis

Once the intraprocedural graphs have been constructed, they are connected together to form an interprocedural exception control flow graph, which is defined as follows:

Definition 3 *An interprocedural exception graph (IECFG) is defined by the tuple* $I_G = \langle N_s, N_e, N_{excepe}, G_{inter} \rangle$*, where* N_s, N_e, N_{excepe} *are the set of start, exit and exceptional-exit nodes of the constituent ECFGs, respectively, and* G_{inter} *is the union of set of intraprocedural graphs of the functions in the program.*

IECFG Construction. Alg. 12 (*BuildInterECFG*) shows the algorithm for constructing the interprocedural graph from the intraprocedural graphs. Initially, the interprocedural graph consists of the union of all the intraprocedural graphs, constructed independently, as described in Sect. 4. In the next step, the call graph is consulted to determine the call targets for each call site. At each call site, three edges are added: (a) a call edge from call node to start node of the target's intraprocedural graph, (b) a call-return edge from the normal exit

```
Algorithm 10: VisitHandler
Input: $h$: A handler where $h = (t\ v)\ b_1$
Output: $N_b \times N_e \times N_{unres}$
1  let $(N_{b_1}, N_{e_1}, N_{unres_1}) = VisitBlock(b_1)$;
2  $N_{catch} = \{n_h\}$; $E_{reg} = E_{reg} \cup \{(n_h, n_s) \mid n_s \in N_{b_1}\}$;
3  $N_b = \{n_h\}$; $N_e = N_{e_1}$; $N_{unres} = N_{unres_1}$;
```

```
Algorithm 11: VisitFunction
Input: $f$: Function where $f = t\ id\ (\overline{t\ id})$
Output: $G_{intra_f} = <n_s, n_e, n_{excepe}, N, E>$
1  let $(N_{b_1}, N_{e_1}, N_{unres_1}) = VisitBlock(b_1)$;
2  $E_{reg} = E_{reg} \cup \{(n_{s_f}, n) \mid n \in N_{b_1}\} \cup \{(n, n_{e_f}) \mid n \in N_{e_1}\}$ ;
3  foreach $n \in S_{exit}$ where $S_{exit} = \{n \mid n \in N_{unres_1} \wedge ir(n) = return\}$ do
4      $E_{reg} = E_{reg} \cup \{(n, n_{e_f})\}$;
5  end
6  foreach $n \in S_{throw}$ where $S_{throw} = \{n \mid n \in N_{unres_1} \wedge ir(n) = throw\}$ do
7      $E_{excep} = E_{excep} \cup \{(n, n_{excepe_f}, T_{prog}(ir(n)))\}$;
8  end
9  foreach $n \in S_{goto}$ where $S_{goto} = \{n \mid n \in N_{unres_1} \wedge ir(n) = goto\}$ do
10     $E_{reg} = E_{reg} \cup \{(n, n_{tgt}) \mid n_{tgt} \in Node(Label(ir(n)))\}$;
11 end
12 for $n \in N_{ecret}$ do
13     $\tau_{ecret} = ECR_\Gamma(n_{ecret})$;
14     $E_{excep} = E_{excep} \cup \{(n, n_{excepe_f}, \tau_{ecret})\}$
15 end
16 return $<n_{b_f}, n_{e_f}, n_{excepe_f}, N, E>$
```

Fig. 9. *VisitHandler* and *VisitFunction* routines for ECFG construction

node of the target's intraprocedural graph to the call-return node of the function call, and (c) an exception edge from the exceptional-exit node of the target's intraprocedural graph to the exceptional-call-return node in the graph. The exception edge is annotated with the union ($\bigcup_\Gamma$) of the exception information on incoming exception edges of the exceptional-exit node, which serves as the initial dataflow fact for the interprocedural exception analysis. Finally, the summary edges connecting the call node with the call-return and exceptional-call-return nodes are removed.

Interprocedural Exception Analysis. Given that the IECFG construction algorithm initially gives a safe overapproximation of the interprocedural exception flow, the goal of the interprocedural analysis is to refine the dataflow facts on the exception edges as precisely as possible. Alg. 13 shows the interprocedural exception analysis algorithm. A single top-down propagation pass on the call graph will not model exceptions precisely in the presence of recursive functions. Therefore, we need to perform a dataflow analysis. Our analysis operates only on the exceptional-exit and exceptional-call-return nodes, and their incoming and outgoing edges. The abstract domain is the Signed-TypeSet domain as defined in Sect. 3. The analysis is implemented using a worklist W_{list}, which initially has the set of exceptional-exit and exceptional-call-return nodes in reverse topological order on the CG_{SCC}, the directed acyclic graph of strong connected components formed from the call graph. Each iteration of the algorithm removes a node from W_{list}, applies a transfer function, updates its outgoing exception

Algorithm 12. *BuildInterECFG*

```
   Input: p: Program
   Output: $I_G = <N_s, N_e, N_{excepe}, G_{inter}>$
1  $G_{inter} = \cup_{f \in p} G_{intra_f}$;
2  foreach calltriple $(n_{call}, n_{cret}, n_{cexcepret})$ do
3      $F = CallTargets(ir(n_{call}))$;
4      for $f$ in $F$ do
5          let $G_{intra_f} = <n_{s_f}, n_{e_f}, n_{exe_f}, N_f, E_f>$;
6          let $E_c = E_c \cup \{(n_{call}, n_{s_f}), (n_{e_f}, n_{cret})\}$;
7          let $\tau_{exit} = \bigcup_{\Gamma_{e_p \in prede(n_{exe_f})}} excepE_\Gamma(e_p)$;
8          $E_{excep} = E_{excep} \cup \{(n_{exe_f}, n_{cexcepret}, \tau_{exit})\}$;
9      end
10     $E_c = E_c - \{(n_{call}, n_{cret})\}$;
11     $E_{excep} = E_{excep} - \{(n_{call}, n_{excepcret})\}$;
12 end
```

edges with the new dataflow facts and adds the successors nodes to W_{list}, if the data flow information has changed. The algorithm is continued until the W_{list} is empty, at which point the algorithm terminates with a fixed point. Termination of the algorithm is guaranteed due to the fact that the set of exceptions is finite.

A map $excepN_\Gamma$ defines the most recent dataflow information, an element from the Signed-TypeSet domain, corresponding to each exceptional-exit or exceptional-call-return node. Another map $excepE_\Gamma$ is used to hold the exception annotation on each exception edge. It is initialized to the union ($\bigcup_\Gamma$) of the exception information on the incoming edges for each node, and is updated every time the result of $\bigcup_\Gamma$ changes. The transfer functions for the exceptional-exit and exceptional-call-return nodes differ in how they update the exception annotation on the outgoing edges, once $\bigcup_\Gamma$ is computed:

- For an *exceptional-exit node*, each outgoing edge's exception annotation is replaced by the newly computed information at the exit node. This operation reflects the refined set of all of possible (uncaught) exception types that could be thrown from a function, represented in the Signed-TypeSet domain (Lines 8-12 in Alg. 13).
- For an *exceptional-call-return node*, each outgoing edge's old exception annotation, is replaced by an intersection ($\bigcap_\Gamma$) of the old exception annotation with the new exception information available at the node. The intersection operation serves to narrow the set of exception types that was previously assumed for a function call, and hence, iteratively increases the precision of the interprocedural exception flow graph (Lines 13-19 in Alg. 13).

Uncaught Exceptions. At the end of the analysis, some of the exception edges in the IECFG will have empty exception annotation, which can be eliminated. Empty exception annotations are identified in two phases. The first phase can be done immediately after the analysis, in which those that use a positive sign $((+, \{\}))$ can be removed. The second phase removes an empty exception annotation that uses a negative sign, and requires conversion of the exception information from the Signed-TypeSet domain to the domain of positive set of types. Alg. 14 shows the conversion algorithm, which is done only once, after the

Algorithm 13. *InterProceduralExceptionAnalysis*

```
Input: $I_G = <N_s, N_e, N_{excepe}, G_{inter}>$
Input: $I_{G'} = <N_s, N_e, N_{excepe}, G_{inter'}>$
$W_{list} = N_{excepe} \cup \bigcup_{f \in M} N_{ecret_f}$;
while $W_{list}$ is not empty do
    $n \leftarrow removeNode(W_{list})$;
    $\tau_{n_{old}} = excepN_{\Gamma}(n)$ ;
    $\tau_{n_{new}} = \bigcup_{\Gamma_{ep \in prede(n)}} excepE_{\Gamma}(e_p)$;
    if $\tau_{n_{old}} \neq_{\Gamma} \tau_{n_{new}}$ then
        switch typeOf(n) do
            case ExcepExit
                foreach $e_s \in succe(n)$ do
                    $excepE_{\Gamma}(e_s) = \tau_{n_{new}}$;
                    $W_{list} = W_{list} \cup \{dst(e_s)\}$;
                end
            case ExcepCallReturn
                foreach $e_s \in succe(n)$ do
                    $excepE_{\Gamma}(e_s) = \tau_{n_{new}} \bigcap_{\Gamma} excepE_{\Gamma}(e_s)$;
                    if $typeOf(dst(e_s)) = ExcepExit$ then
                        $W_{list} = W_{list} \cup \{dst(e_s)\}$;
                    end
                end

        end
        $excepN_{\Gamma}(n) = \tau_{n_{new}}$;
    end
end
```

analysis is performed and also serves as a checker for the "no throw" guarantee. The algorithm walks the CG_{SCC} in reverse topological order, and at each step, uses the set of all exception types (positive) that could potentially be thrown by the transitive callees of a function, to serve as the universal set, from which to subtract the negated set of exceptions corresponding to the current function. Whenever a SCC of mutually recursive functions is encountered, the union of the set of uncaught exception types of each constituent function in the SCC is used as a sound overapproximation for the subtrahend. The algorithm, produces a map $excep_F$ that gives for each function the set of potentially uncaught exception types.

Example. Fig. 10 shows the final interprocedural exception control flow graph for our example. The exception edges: *ecr-readLine* → *exe-get* and *ecr-read* → *exe-readLine* are notably missing from the graph. The analysis is able to infer this after performing the intersection operation, between exception information on incoming and outgoing edges of *ecr-readLine* and *ecr-read*:

$$(+, \{IOException\}) \bigcap{}_{\Gamma} (-, \{IOException\}) = (+, \{\}) \text{ and}$$

$$(+, \{EOFException\}) \bigcap{}_{\Gamma} (-, \{EOFException\}) = (+, \{\}).$$

However, we see that the program may potentially fail with the uncaught exception `std::bad_alloc` thrown in `get()` by the *new* operator. ■

Algorithm 14. *ComputeUncaughtExceptions*

```
   Input: $I_G = <N_s, N_e, N_{excepe}, G_{inter}>$
   Output: $excep_F : F \mapsto T_{prog}$
 1 let $T_{excep} = \{\}$;
 2 for $n_t \in N_{throw}$ do
 3     $excep_T(n_t) = T_{prog}(ir(n_t))$ ;
 4     $T_{excep} = T_{excep} \bigcup T_{prog}(ir(n_t))$
 5 end
 6 for $external\ function\ f$ do
 7     $excep_F(f) = \{t_{unknown}\}$
 8 end
 9 $F_l = ReverseTopoOrder(CG_{SCC})$;
10 while $F_l$ is not empty do
11     $F_{scc} = RemoveFront(F_l)$;
12     for $f \in F_{scc}$ do
13         $excep_F(f) = \bigcup excep_T(n_t)$ where $(n_t \in N_{throw}) \wedge (\exists g \cdot g \in F_{scc} \wedge ir_n(n_t) \in g)$
14     end
15     for $f \in F_{scc}$ do
16         switch $excepN_\Gamma(n_{excepe})$ where $n_{excepe} = N_{excepe}(f)$ do
17             case $(+, T_E)$
18                 $excep_F(f) = T_E$
19             case $(-, T_E)$
20                 foreach $g \in TransitiveCallees(f)$ do
21                     $excep_F(f) = excep_F(f) \bigcup excep_F(g)$;
22                 end
23                 $excep_F(f) = excep_F(f) - T_E$;
24
25         end
26     end
27 end
```

6 Generating an Exception-Free Program

In this section, we describe our lowering algorithm that translates a given C++ program into a semantically equivalent program without exception-related constructs such as throw, catch, etc. The lowering algorithm uses the IECFG to eliminate exception-related constructs. There are two main distinguishing features of our lowering algorithm compared to existing C++ compilers:

– Our approach uses a combination of stack storage and reference parameters to simulate exceptions without generating additional runtime calls whose semantics have to be taught to existing static analysis tools.
– Our approach uses the exception target information available in the IECFG, and therefore, when compared to existing C++ lowering techniques, generates fewer infeasible edges between throw statements and catch blocks that are not present in the original program. It also handles insertion of destructors correctly. The modular design of the IECFG makes it easy to insert the destructor calls in a single pass.

The main steps of the lowering algorithm are as follows:

1. **Creation of Local Exception-Objects and Formal Parameters:** Each function's (say *f*) local variable list is extended with: (1) a "type-id" variable, and (2) a local exception-object variable for every exception type that can potentially be thrown within *f*. The "type-id" variable holds the type of the

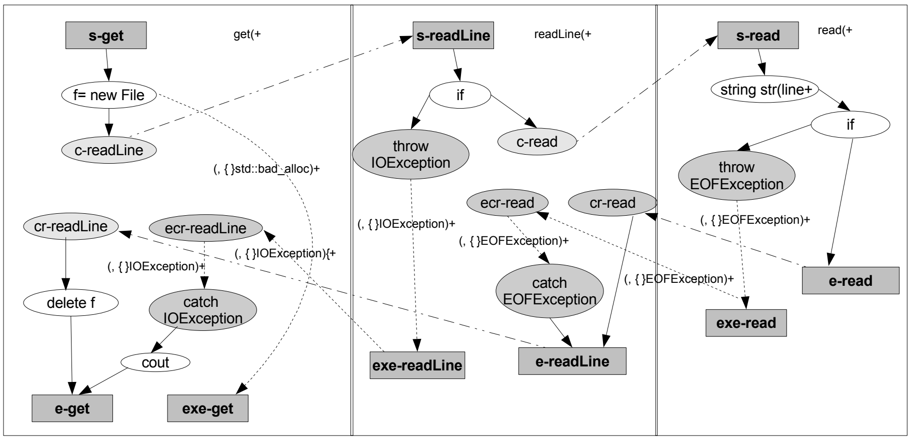

Fig. 10. Interprocedural Exception Control Flow Graph for the program in Fig. 1

thrown exception, and the local exception-object variable holds the thrown exception object and acts as storage for interprocedural exception handling. Additional parameters are added to *f*'s signature: (1) a reference parameter for every uncaught exception type that propagates out of the function, and (2) a reference parameter to hold the "type-id". These reference parameters propagate information about uncaught exceptions to a caller. At each call-site of *f*, appropriate local exception-objects and the caller's local type-id are passed additionally as parameters to *f*.

2. **Lowering throws and catch:** Based on the targets of throw statements in the IECFG, calls to the destructors of appropriate set of local objects are inserted. The thrown object is assigned to the local exception-object of the appropriate type and the local type-id variable is set to the thrown type. A goto is then inserted either to a catch block or to the exceptional-exit node. At the catch block, the local exception-object is assigned to the argument of the catch-header.
3. **Lowering exceptional-call-return nodes:** A *switch* statement (modeled using *if*s in our IL) on the local type id is inserted, with one nested *case* for every uncaught exception type in the callee. The target node information from the IECFG is used to place calls to destructors of appropriate stack-allocated objects, for each *case*. Finally, a *goto* to the target (either a catch or an exceptional-exit) is inserted.
4. **Lowering exception exit node:** The local exception-objects and type-ids are copied into corresponding formal parameters. This serves to copy the exception objects out of the callee into the caller, which deals with the uncaught exceptions at its exceptional-call-return node.

The semantics preserving nature of the lowering algorithm can be established as follows. Our lowering mechanism is based on the observation that exception handling preserves functional scoping even though exceptions result in non-local control flow. This is because the program has to unwind the call stack to invoke the destructors of local objects that have been constructed in the functions on the call stack until the exception is caught. Therefore, it largely mirrors the flow that happens during a regular call return. Our lowering mechanism mimics this flow by placing the destructor calls before the exceptional return of every function and passing pending exception objects through the additional reference parameters that were added by the lowering.

Subtleties introduced by some C++ features are handled as follows:

Throwing Destructors. As per the C++0x standards draft, destructors throwing an exception during stack unwinding result in a call to `std::terminate()`, which by default terminates the program. However, the destructor's callees can throw exceptions as long as they do not flow out of the destructor. Multiple live exceptions arising out of this are correctly handled in our lowering algorithm by the use of (a) local exception objects, which implicitly helps to maintain a stack of multiple outstanding exceptions, and (b) a global exception flag to detect a throwing destructor instance and trigger a call to `std::terminate()`.

Virtual Functions throwing different exceptions. Multiple function targets at a call site, quite common in C++ due to virtual functions, can in general throw different exception types. Our lowering algorithm prevents ambiguity in the function signature by generating a uniform interface at the call site, that uses the union of exception types that can be thrown by each possible target of a virtual function call.

Catch-all and rethrow. A catch-all clause (`catch (...)`) does not statically indicate the type of C++ exception handled by the clause. Rethrow statements (`throw;`) do not have a throw expression as an argument. Our lowering algorithm requires type and variable information, which is obtained by using the exception information from the IECFG. Since the IECFG has an edge to a catch clause annotated with the type of each possible exception thrown, the catch-all clause is expanded to a sequence of concrete clauses. Rethrows are handled by using the exception information from the nearest enclosing catch clause.

Exception Subtyping. The lowering algorithm assumes that the type of thrown exception is the same as the type of the catch clause, which may not be true in general due to the exception subtyping rules of C++. This case is handled by generating super class (w.r.t exception subtyping rules) local and formal exception objects, and assigning into them, thrown exception objects which are subclasses of the superclass object.

Example. The lowered code for our running example is shown in Fig. 11 after performing copy propagation to remove redundant local objects. Fig. 11 also shows the exception specifications for functions. (The specification has details of the pending call stacks for each uncaught exception, but is not shown here.) ■

```
string File::read(EOFException& e1,
                  ExcepId& id) {
 string str(__line);
 if (EOF) {
   EOFException tmp; e1 = tmp;
   id = EXCEP_EOF_EXCEPTION;
   ~str();
   goto EExit_1;
 }
 return str;
EExit_1: return string();
}
```

```
// EXCEPTION SPECIFICATIONS:
// void get()         throw (std::bad_cast);
// string read()      throw (EOFException);
// string readLine()  throw (IOException);
#define EXCEP_NULL 0
#define EXCEP_BAD_CAST 1
#define EXCEP_IO_EXCEPTION 2
#define EXCEP_EOF_EXCEPTION 3

class EOFException { ... };
class IOException { ... };
```

```
void get (std::bad_cast& e3, ExcepId &id) {
 string s; IOException e4;
 ExcepId lid = EXCEP_NULL;

 File *file = new_alloc("1.txt");
 if (file == NULL) {
  std::bad_cast tmp;
  e3 = tmp; id = EXCEP_BAD_CAST;
  goto EExit_2;
 }
 file->readLine(e4, lid);
 switch (lid) {
  case EXCEP_IO_EXCEPTION:
   goto Catch_L2;
   break;
  default:
   break;
 }
 delete file;
 return;
Catch_L2: {
  cout  << ''IO-Failure'';
 }
EExit_2:
}
```

```
string File::readLine(IOException& e2,
                      ExcepId& id) {
 string s; EOFException e1;
 ExcepId lid = EXCEP_NULL;

 if (invalidFile) {
   IOException tmp; e2 = tmp;
   id = EXCEP_IO_EXCEPTION;
   ~s();
   goto Exit_2;
  }
 s = read(e1, lid);
 switch (lid) {
  case EXCEP_EOF_EXCEPTION:
    goto Catch_L1;
    break;
  default:
    break;
 }
 return s;
Catch_L1: {
  EOFException& e = e1;
  return string("");
 }
Exit_2: return string();
}
```

Fig. 11. Exception specifications and exception-free program for the running example

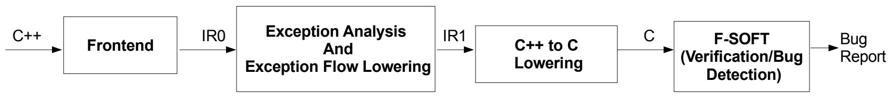
C++
Frontend
IR0
Exception Analysis And Exception Flow Lowering
IR1
C++ to C Lowering
C
F-SOFT (Verification/Bug Detection)
Bug Report

Fig. 12. Exception Analysis and Transformation Workflow

7 Implementation and Experiments

We have implemented our exception analysis and transformation algorithms in an in-house extension of CIL [14], which handles C++ programs. The exception analysis implementation has about 6,700 lines of OCAML code. Fig. 12 shows the workflow for analyzing and transforming C++ programs with exceptions. The given C++ program is initially parsed by our frontend into a simplified intermediate version of C++ (IR0) similar to the IL shown in Section 2. The IR0 code is then fed to our interprocedural exception analysis and transformation framework, which produces lowered C++ code without exceptions. The

Table 1. Results of interprocedural exception analysis and exception safety checks

Benchmark	Simplified LOC	ECFG Build Time(s)	IECFG Build & Analysis Time(s)	#Excep Edges before Analysis	#Excep Edges after Analysis	"No throw" guarantee Coverage (% Functions)	"No leak" check results (#detected /#actual)
multiple-live	479	0.01	0.01	10	6	71 %	0/0
ctor-throw	585	0.02	0.02	33	11	89 %	1/1
recursive	643	0.02	0.03	27	17	64 %	0/0
shared-inherit	667	0.02	0.04	59	27	71 %	1/1
bintree-duplicate	770	0.06	0.05	31	13	91 %	0/1
list-baditerator	784	0.04	0.04	36	17	79 %	0/2
virtual-throw	809	0.02	0.03	39	28	46 %	0/4
nested-try-catch	809	0.03	0.03	33	17	68 %	2/2
loop-break-cont	814	0.04	0.03	33	17	68 %	2/2
nested-rethrow	820	0.04	0.03	35	19	68 %	4/4
new-badalloc	849	0.02	0.03	30	15	76 %	2/2
template	860	0.03	0.04	51	28	67 %	1/1
dyn-cast	872	0.03	0.05	35	16	81 %	1/1
iolib	919	0.01	0.01	9	7	40 %	1/1
delegat-dtor-throw	1305	0.04	0.05	63	62	53 %	0/0
std-uncaught-dtor	1348	0.05	0.07	73	71	58 %	1/1

transformed C++ code is then lowered to C by a module that lowers various object-oriented features into plain C. The C++-to-C lowering module transforms features such as inheritance and virtual-function calls without the use of run-time structures such as virtual-function and virtual-offset tables. Therefore, the lowered source code is still at a relatively high-level for further static analysis. The lowered C code is then fed into F-Soft [7], where standard bug detection and verification tools that work on C are applied.

We have evaluated our exception analysis and transformation algorithms on a set of C++ programs [13]. The programs test usage of various C++ exception features in realistic scenarios, some of which are close to standard C++ collection class usage [21]. We used the results of our analysis to test the "no throw" guarantee, immediately before lowering, and the results of our transformation to test the "no leak" guarantee using F-Soft. For the experiments, exceptions of type $t_{unknown}$ from external library calls were omitted. Tab. 1 shows the results. The running time for ECFG construction for all programs is low, while the IECFG construction and analysis is quite comparable, with most of the time spent in the interprocedural exception analysis. Our interprocedural exception analysis is able to achieve an average reduction of about 38% in the number of exception edges, with the IECFG constructed immediately before the interprocedural analysis serving as the baseline. On an average, around 66% of the functions in a program were certified as "no throw".

The last column shows the results of running F-Soft, specifically a memory leak detector module, on the lowered programs. 13 of the 16 benchmarks that we used for these experiments had memory leaks along exception paths, and F-Soft reported all memory leaks in 10 of the 13 benchmarks. F-Soft failed to find memory leaks for 3 benchmarks due to timeouts and reported bogus witnesses only for `new-badalloc` due to the limitation of our in-house C-lowering. For these experiments we used a time-bound of 10 minutes for the verification.

Table 2. Results of interprocedural exception analysis on open-source benchmarks

Open-source Benchmark	Simplified LOC	ECFG Build Time(s)	IECFG Build & Analysis Time(s)	#Excep Edges before Analysis	#Excep Edges after Analysis	"No throw" guarantee Coverage (% Functions)
tinyxml	4884	0.39	1.41	1204	830	74 %
mailutils	8365	0.19	0.36	494	316	78 %
coldet	8422	0.27	0.22	591	20	98 %
id3lib	14070	1.73	4.18	2091	372	93 %

One of the reasons for the timeouts is that the lowering algorithm generates programs that is atypical of the C source code that F-Soft has previously analyzed, which affects the performance of the model checker. As an example, we have found that the addition of destructor calls during stack unwinding on exceptional edges introduces many additional destructor call sites; in the benchmark `std-uncaught-dtor`, there were 31 call sites to a particular class destructor. The additional destructor calls during stack unwinding yield function call graphs that are very different from what F-Soft usually encounters. Therefore, additional heuristics, such as selective function inlining for destructor calls, will likely improve the performance of the model checker on the models generated by the exception analysis module.

Results on open-source benchmarks. We have also applied the IECFG construction algorithm on a set of open-source benchmarks shown in Tab. 2. The coldet benchmark is an open source collision detection library used in game programming. GNU mailutils is an open source collection of mail utilities, servers, and clients. TinyXML v2.5.3 is a light-weight XML parser, which is widely used in open-source and commercial products. The open source library id3lib v3.8.3 is used for reading, writing, and manipulating ID3v1 and ID3v2 tags, which are the metadata formats for MP3s. Tab. 2 shows the reduction in the number of exception edges due to our interprocedural analysis. A direct consequence of this reduction is seen in the "no throw" guarantee numbers, which represent the percentage of the total functions in the program, for which we are able to guarantee that no exceptions will be thrown by them. For coldet, which had the maximum reduction in the number of edges, the number of functions guaranteed not to throw is about 98%. We are encouraged by the results of our experiments on the open source benchmarks. For these benchmarks, the time taken to compute the IECFG is less than 5s. Therefore, we believe that the analysis will scale to even larger examples.

Memory leaks in `mailutils` applications. We also applied the memory-leak checker module of F-Soft on two applications that use the mailutils library: (1) `iconv`, which converts strings from one character encoding to another using the mailutils library, and (2) `murun`, which tests the various kinds of streams in the mailutils library. F-Soft reported one memory leak in `iconv` and three memory leaks in `murun` involving exceptional control flow. The offending code snippet in `iconv` is shown in Fig. 13. In the `try` block, the invocation of the constructor

FilterIconvStream() for variable cvt may throw an exception. However, when the exception is handled by the catch block, the memory allocated at the start of the try block is not deallocated, which results in a memory leak. The leaks reported in murun also have a similar flavor.

In addition, F-Soft reported a leak in iconv where the memory allocated in the constructor of class FilterIconvStream is never deallocated. Note that this leak occurs even when no exceptions are thrown by the application.

```
...
try {
  StdioStream *in = new StdioStream (stdin, 0);
  in->open ();
  FilterIconvStream cvt (*in, (string)argv[1], (string)argv[2], 0,
     mu_fallback_none);
  cvt.open ();
  delete in;
  ...

}
catch (Exception& e) {
  cerr << e.method () << ":␣" << e.what () << endl;
  exit (1);
}
```

Fig. 13. Memory leak along an exception path in iconv

8 Related Work

Most related work deals with exceptions in Java. Earlier, we discussed many differences between exceptions in C++ and Java, thus requiring different approaches. Sinha and Harrold [16] incorporate the control flow effects due to explicit Java exceptions in an interprocedural control flow graph (ICFG) using a flow-sensitive type analysis, and discuss their applications to control dependence analysis and slicing. The ICFG used in their analysis has no exceptional-call-return node and can have multiple exceptional-exit nodes in a function. In contrast, our IECFG is modular and has an exceptional-call-return node for every function call, which is required for modeling implicit C++ destructor calls. Jo et al. [8,9,3] construct an exception flow graph for Java, using a set constraint analysis that is required to iterate to convergence. Gherghina and David [6] present a specification logic for exceptions for Java-like languages and verify exception-safety guarantees. Their specification logic does not model destructors along exception paths because they target Java-like languages, and therefore, cannot be used for verifying C++ programs. Mao and Lu [12] perform the analysis for C++, without explicitly modeling destructors. Robillard and Murphy [15] develop an analysis that handles both checked and unchecked exceptions in Java. In contrast to all these approaches, our analysis based on the Signed-TypeSet domain can be terminated safely at any point after the IECFGs have been constructed, thus permitting sound static analysis on subparts of the IECFG. Given the prevalence of separate compilation in large systems, a modular approach that can be safely terminated at any step is essential for scalability and adoption into compilers. The Signed-TypeSet domain utilizes a similar idea as was used in the form of *difference sets* for class hierarchyanalysis [4].

Weimer and Necula [20] propose an intraprocedural, path-sensitive analysis for checking typestate specifications along exception paths in Java. Buse and Weimer [2] propose a similiar symbolic analysis for automatic documentation of Java exceptions. Bravenboer and Smaragdakis [1] propose a solution for Java, where pointer analysis and exception analysis problems are framed and solved in a mutually recursive manner, with each improving the precision of the other. Fu and Ryder [5] propose a static analysis that computes chains of semantically related exception flow links, by composing existing exception analyses to give longer exception paths. Our IECFG construction and exception analysis for C++, could potentially be enhanced with all of the above techniques to improve the accuracy of our exception model. Li et al. [11], propose a technique to check the "no leak" guarantee for Java, using a combination of static analysis and model checking. Similarly, Torlak and Chandra present an interprocedural static analysis algorithm to detect resource leaks in Java programs [19]. In our work, we use F-Soft [7] for checking for resource leaks, following our exception analysis and lowering transformation.

9 Conclusions and Future Work

This paper introduced an interprocedural analysis framework for accurately modeling C++ exceptions. In this framework, control flow induced by exceptions is captured in a modular interprocedural exception control flow graph (IECFG). This graph is refined by a novel dataflow analysis algorithm, which abstracts the types of exception objects over a domain of signed set of types. Unlike exception analyses proposed elsewhere for other languages, this analysis can be safely terminated at well-defined points during interprocedural propagation, thereby allowing clients to trade-off speed over precision. The paper then presented a lowering transformation that uses the computed IECFG to generate an exception free program. This transformation is designed specifically to permit easier and more precise static analysis on the generated code. Finally, the paper demonstrated two applications of the framework: (a) automatic inference of exception specifications for C++ functions and (b) checking the "no throw" and "no leak" exception safety properties.

In the future, we intend to perform additional experiments on larger benchmarks. As shown in Sect. 7, the IECFG computation presented here should scale well for larger benchmarks. Finally, we are investigating to selectively allow conditional exception edges during the IECFG construction. Such conditional exception edges could be used to model cases involving throwing destructors or other standard library objects such as `cout` more precisely. For example, a throwing destructor would be allowed to propagate the fact that `std::uncaught_exception()` was queried before throwing an exception. This could be used to eliminate spurious calls to `std::terminate()` when returning from such destructors. Similarly, we can annotate other calls, such as uses of `cout` with the information that it may throw an exception, if the surrounding context had set the relevant information using the `ios::exceptions()` method. Such selective guards on exception edges would not substantially decrease the performance of the analysis but would allow further reduction of computed exception-catch links.

References

1. Bravenboer, M., Smaragdakis, Y.: Exception analysis and points-to analysis: Better together. In: International Symposium on Software Testing and Analysis (ISSTA), pp. 1–12. ACM, New York (2009)
2. Buse, R.P., Weimer, W.R.: Automatic documentation inference for exceptions. In: ISSTA, pp. 273–282. ACM, New York (2008)
3. Chang, B.-M., Jo, J.-W., Yi, K., Choe, K.-M.: Interprocedural exception analysis for Java. In: Proc. of Symp. on Applied Computing, pp. 620–625 (2001)
4. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 77–101. Springer, Heidelberg (1995)
5. Fu, C., Ryder, B.: Exception-chain analysis: Revealing exception handling architecture in Java server applications. In: ICSE, pp. 230–239 (May 2007)
6. Gherghina, C., David, C.: A specification logic for exceptions and beyond. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 173–187. Springer, Heidelberg (2010)
7. Ivančić, F., Shlyakhter, I., Gupta, A., Ganai, M., Kahlon, V., Wang, C., Yang, Z.: Model checking C programs using F-Soft. In: IEEE International Conference on Computer Design, pp. 297–308 (October 2005)
8. Jo, J.-W., Chang, B.-M.: Constructing Control Flow Graph for Java by Decoupling Exception Flow from Normal Flow. In: Laganá, A., Gavrilova, M.L., Kumar, V., Mun, Y., Tan, C.J.K., Gervasi, O. (eds.) ICCSA 2004. LNCS, vol. 3043, pp. 106–113. Springer, Heidelberg (2004)
9. Jo, J.-W., Chang, B.-M., Yi, K., Choe, K.-M.: An uncaught exception analysis for Java. Journal of Systems and Software 72(1), 59–69 (2004)
10. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: International Symposium on Code Generation and Optimization (CGO), Palo Alto, California (March 2004)
11. Li, X., Hoover, H., Rudnicki, P.: Towards automatic exception safety verification. In: Proc. of Formal Methods, pp. 396–411. Springer, Heidelberg (2006)
12. Mao, C.-Y., Lu, Y.-S.: Improving the robustness and reliability of object-oriented programs through exception analysis and testing. In: IEEE International Conference on Engineering of Complex Computer Systems, vol. 0, pp. 432–439 (2005)
13. NECLA verification benchmarks, `http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php`
14. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Int. Conf. on Comp. Construct, pp. 213–228. Springer, Heidelberg (2002)
15. Robillard, M.P., Murphy, G.C.: Static analysis to support the evolution of exception structure in object-oriented systems. ACM Transactions on Software Engineering Methodologies 12(2), 191–221 (2003)
16. Sinha, S., Harrold, M.J.: Analysis and testing of programs with exception handling constructs. IEEE Trans. on Software Engineering 26, 849–871 (2000)

17. C.standards commitee. Working draft, standard for programming language C++ (2010),
`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3126.pdf` (accessed September 26, 2010)
18. Stroustrup, B.: Exception safety: Concepts and techniques. In: Romanovsky, A., Cheraghchi, H.S., Lee, S.H., Babu, C. S. (eds.) ECOOP-WS 2000. LNCS, vol. 2022, pp. 60–76. Springer, Heidelberg (2001)
19. Torlak, E., Chandra, S.: Effective interprocedural resource leak detection. In: Int. Conf. on Softw. Eng., pp. 535–544. ACM, New York (2010)
20. Weimer, W., Necula, G.C.: Exceptional situations and program reliability. ACM Trans. Programming Languauges and Systems 30(2), 1–51 (2008)
21. Weiss, M.A.: Data Structures and Algorithm Analysis in C++. Addison-Wesley Longman Publishing Co., Inc., Boston (1998)