



Cost Efficient Dependable Electronic Systems

Design and implementation of an Algorithm for the Strong Exception-Safety Guarantee in C++

Author	Gustav Munkby
Document Id	025
Date	November 2006
Availability	Public
Status	Final

CHALMERS | GÖTEBORG UNIVERSITY

MASTER'S THESIS

Design and Implementation
of an Algorithm for the Strong
Exception-Safety Guarantee in C++

GUSTAV MUNKBY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
GÖTEBORG UNIVERSITY
Göteborg, Sweden 2006

Thesis for the Degree of Master of Science (20p)

Design and Implementation of an Algorithm for the Strong Exception-Safety Guarantee in C++

Gustav Munkby

CHALMERS | GÖTEBORG UNIVERSITY



Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE – 412 96 Göteborg, Sweden
Göteborg, November 2006

Abstract

Exception handling mechanisms provide a structured way to deal with exceptional circumstances, which makes it easier to read a program and reason about it, but cannot avoid the problem that the transfer of control might leave the program in an inconsistent state—resources might leak, invariants might be violated, the program state might be changed. Since client code often needs to know how a program behaves in the presence of exceptions, the *exception safety classification* distinguishes three different classes of safety guarantees; this classification is used, for example, during the review process in the Boost organization for standardized libraries in C++. Although it is not easy to correctly classify the exception level of a procedure, no tool support was available until now; thus, designers and reviewers had to trace the control flow of a program manually and along all hidden execution paths. In this paper we present the first automated analysis for exception guarantees. The analysis addresses two of the three safety guarantees, the *strong* and the *no-throw* guarantee. The analysis is implemented in the BANGSAFE tool set, which interfaces the ELSA parser for C++ and targets C++-programs. BANGSAFE itself is implemented in Ruby.

Sammanfattning

Felhantering med hjälp av *exceptions* erbjuder ett strukturerat alternativ för att hantera undantagsfall, vilket underlättar såväl förståelsen av som möjligheten att analysera ett program. Även med *exceptions* är det svårt att korrekt hantera fel som uppkommer, vilket kan leda till resursläckor, brutna invarianter eller förändrat programtillstånd. Eftersom klienten vid proceduranrop måste veta hur programmet beter sig när ett fel inträffar, graderas procedurer enligt *exception safety*, vilket anger tre olika säkerhetsnivåer. Denna gradering används bland annat vid granskningsprocessen inom Boost för standardisering av källkodsbibliotek i C++. Trots svårheten att korrekt gradera en procedure, finns för närvarande inga analysverktyg, vilket tvingar programkonstruktörer och granskare att manuellt följa programmets alla kontrollflöden. Genom detta examensarbete, presenteras den första automatiserade analysen av *exception safety* graderingar. Analysen hanterar två av de tre säkerhetsnivåerna, den *förstärkta* och den *felfria* säkerhetsnivån. Analysen är inbyggd i verktyget BANGSAFE, vilket använder C++-parsern ELSA för att kunna hantera C++-program. Själva verktyget BANGSAFE är byggt i Ruby.

Contents

1	Introduction	13
1.1	Motivation	13
1.2	Approaches	14
1.3	Main results	16
1.4	Outline	17
2	Background	19
2.1	Exception handling	19
2.1.1	History of exception handling	19
2.1.2	Exception handling in C++	20
2.1.3	Exception analysis	21
2.2	Exception safety	22
2.3	Static Analysis	24
2.3.1	Terminology	24
2.3.2	Standard structures	24
2.4	Mathematics	25
2.4.1	Abstract algebra	25
2.4.2	Graph theory	26
3	Architecture	27
4	Approach	29
4.1	Running example	29
4.2	Parsing the input	30
4.3	Creating the control-flow graph	32
4.3.1	compute_cfg	34
4.3.2	examine_ast_node	34
4.3.3	compute_edges	35
4.3.4	traverse_ast_children	37
4.3.5	Running example	38
4.4	Abstract representation	38

4.5	Annotating the control-flow graph	41
4.5.1	Implementation	43
4.5.2	Running example	44
4.6	Analysis	45
4.6.1	Partial and total correctness	46
4.6.2	Discussion	48
4.7	Synthesis	49
5	Examples	51
5.1	The naive approach	52
5.2	The naive approach, take two	56
5.3	The messy approach	57
6	Conclusion	63
6.1	Evaluation and future work	63
6.1.1	Coverage	63
6.1.2	Precision	66
6.1.3	Tool	67
6.2	Availability	67
	References	69

List of Figures

1.1	Exception-safety guarantees	14
1.2	Questions answered by analysis	16
2.1	Exception-handling in C++	21
3.1	Architecture	27
3.2	Toolset	28
4.1	Running example source code	30
4.2	Supported input language grammar	31
4.3	Pseudo-code for <i>compute_cfg</i>	34
4.4	Pseudo-code for <i>examine_ast_node</i>	35
4.5	Pseudo-code for <i>compute_edges</i>	36
4.6	Pseudo-code for <i>traverse_ast_children</i>	37
4.7	Running example control-flow graph	39
4.8	Join operations on control-flow graphs	40
4.9	Pseudo-code for <i>annotate</i>	43
4.10	Pseudo-code for <i>annotate_call</i>	44
4.11	Running example annotated	44
4.12	Pseudo-code for <i>local_update</i>	45
4.13	Minimal two iteration control-flow graph	47
4.14	Running example BANGSAFE output	50
5.1	Source code of allocation functions	51
5.2	Source code of <i>bomb</i> class	52
5.3	Source code of naive vector constructor	53
5.4	Graph generated by CFGGEN for the naive approach	54
5.5	Graph generated by BANGGRAPH for the naive approach	55
5.6	Output produced by BANGSAFE for the naive approach	56
5.7	Source code of messy vector constructor	57
5.8	Graph generated by CFGGEN for the messy approach	58
5.9	Graph generated by BANGGRAPH for the messy approach	59

5.10	Output produced by BANGSAFE for the messy approach . . .	60
------	--	----

List of Tables

2.1	Relation between our and Alexandrescu's abstractions	23
4.1	Abstract representation	38

Chapter 1

Introduction

Exception-handling mechanisms are standard features of most modern programming languages. The fundamental idea of exceptions is to help the developer in producing a robust system, i.e., to provide the developer with a structured way of dealing with unusual circumstances. The major benefit of exception handling is to make explicit which part deals with normal circumstances, and which part with exceptional ones. This benefit is not only available to developers inspecting the code, but also to programs. Modern C++ compilers, for instance, package the exception handling code away from the normal code, to improve the performance of the code when no exception occurs.

Although exception handling is a significant improvement over previous error-handling techniques, it cannot be tacked on to a program without further thinking. The increased readability introduced by exceptions does not automatically make the program more robust.

1.1 Motivation

Within different communities, different approaches have emerged for designing with exceptions. In the C++ community, the most widely used approach is to classify procedures according to different safety levels. This classification is called the *exception safety classification* and applied when standardizing C++ components.

Exception safety classification includes three different levels, namely the basic, the strong, and the no-throw guarantee, listed in Figure 1.1. The basic guarantee is the weakest, and states that the program will still be in a valid state after an exception is thrown. The strong guarantee additionally specifies that if an exception is thrown from an operation, then the program

basic guarantee the invariants of the program are preserved.

strong guarantee in addition to the basic guarantee, the operation provides rollback semantics in the event of an exception.

no-throw guarantee the operation will not throw an exception.

Figure 1.1: Exception-safety guarantees

state shall be the same as before the operation started. Finally, the no-throw guarantee means that the procedure may not throw any exceptions.

As an example, we briefly consider an data structure representing the *time-of-day* as hours and minutes. For such data structure, the invariant specifies that the hours and minutes lie within specific boundaries. Consider the operation *next-hour* that changes the time to represent the time one hour later, but raises an exception when the resulting time would no longer be valid. To achieve the basic guarantee, the operation could reset the time to any valid value before raising the exception, whereas the strong guarantee requires that the original value is restored before raising.

Manually classifying a procedure according to the exception-safety guarantees is no easy task, mainly because many of the language-features of C++ interact to create a flow of information that is not entirely obvious to the untrained eye. Manually checking which guarantee is fulfilled requires skill and training, and also considerable time and effort. Since this effort is complicated and time-consuming, much would be gained if the process could be automated, but currently there is not much tool-support for checking exception-safety properties.

1.2 Approaches

The classical approaches to automating source-code checking include both dynamic and static approaches. Dynamic checking usually means testing, which is performed by executing the operation with a set of inputs and checking that the outputs are as expected. Static checking means that the operation is not executed, but analyzed off-line by a separate tool.

One of the few tools that exist for checking exception safety classifications is a test-suite developed by Abrahams for STLport [1], which ensures that the algorithms and components of the Standard Template Library (STL) provide maximum exception-safety guarantees. The test-suite runs the STL operations with cleverly engineered inputs, to ensure that the algorithms and

components perform as they should, even when exceptions are emitted.

Testing whether the program state is corrupted or not requires the test-suite to encode the desired properties of the program state, i.e., the program invariants, in the test-suite. These tests are almost always very difficult to write. For a system more complicated than STL, writing these tests might become infeasible.

Since testing only executes the algorithm for a subset of the possible inputs, it is up to the developer of the tests to ensure that the test data covers all possible inputs. Ensuring that all cases have been tested is even more complicated when testing exception-handling code, since the code being tested is triggered only in exceptional cases. One therefore has to carefully set up the tests to ensure that all possible exceptions are triggered.

The advantage of static checking, in contrast, is that its result does not depend on special inputs. This advantage also makes this kind of analysis fundamentally hard, since the analysis must consider every possible input to an operation, and, in the case of exception safety, needs to keep track of all possible variations of the program state. The major difficulty of a static analysis for exception safety is to ensure that the program state is always valid. Addressing this problem requires an analysis that can check that the program state remains valid after an operation. This kind of analysis is studied in the field of formal verification; generally this question is undecidable.

We conclude that a static approach is most feasible, in part because the traditional classification of exception safety is a worst-case classification, but more importantly, because the difficulty of the problem stems from the hidden flows of control. Using a dynamic approach, the developer would still have to figure out what kind of flows can occur, and set up appropriate test data to execute these flows.

Alexandrescu and Held have provided a static analysis, designed for manual application [4]. In their intermediate representation they use two properties, the purity of a procedure and the guarantee it fulfills. A procedure is pure if it has neither state modification nor side effects. This abstraction allows their analysis to identify whether a procedure fulfills the strong guarantee and the no-throw guarantee, but not the basic guarantee. Since Alexandrescu's algorithm is designed for manual use, it is described in loose terms. By automating the analysis performed by his algorithm, the manual work currently involved in exception safety classification would be significantly reduced and would prevent some human errors.

- Can the procedure exit with an exception?
- Can the procedure cause any state modifications?
- Can the two above jointly occur inside the procedure?

Figure 1.2: Questions answered by analysis

1.3 Main results

We have designed and implemented a static analysis for the strong exception-safety guarantee. The analysis consumes C++ source code and produces an abstraction over the control-flow graph. The support input is a subset of C++, which includes the exception handling features, but excludes uninstantiated templates, classes, and recursive procedures. The analysis assumes that side effects have been made explicit by a previous analysis and pessimizes on pointer variables.

Our algorithm is very similar to the manual algorithm described by Alexandrescu. We further simplify Alexandrescu’s algorithm by only considering explicit state modification to determine the purity of a procedure. The analysis will tell whether a procedure can exit with a thrown exception or cause any state modifications and, if state can be modified, whether the procedure exits with an exception. Figure 1.2 summarizes the results of the analysis in three questions. The first question determines whether the procedure can provide the no-throw guarantee and the last one determines whether the strong guarantee is fulfilled. The second question is used only to obtain the answer to the third question, and does not provide any information about exception safety in itself.

By answering the above questions, our analysis, like Alexandrescu’s, will not be able to correctly identify whether the basic guarantee holds, but will make it possible to check whether the no-throw guarantee holds and to give a pessimistic judgment on whether the strong guarantee holds. This judgment is pessimistic since the second question of Figure 1.2 is a sound but pessimistic abstraction. Ideally, the second question should address invariant invalidation, but this is a difficult property to check. It is clear that invalidated invariants imply that the program state has changed, but the reverse is not always true.

The answer to the third question is computed using an annotated version of the control-flow graph, where the annotations describe whether there are state modifications or thrown exceptions. Checking whether there exists a sequence of state modification and thrown exceptions has thus been

transformed into the task of identifying whether there exists a path in the control-flow graph, from procedure entry going through at least one state modification and reaching procedure exit by throwing an exception.

When processing the control-flow graph, our tool incrementally combines nodes of the graph and calculates an annotation appropriate for the encountered subgraphs. Since the annotation of several nodes can be contracted into a single annotation, the algorithm scales well to the interprocedural case. The algorithm is compositional in the sense that once the proper attributes have been established for a single procedure, this procedure will not have to be reexamined again.

To ensure that this local update is in fact correct, a semiring formalism has been developed, which defines the rules for locally updating the combination of several nodes of the control-flow graph into a single annotation. If these operations are applied in the proper order, the end result will be an algorithm with linear time complexity, which provides answers to the three questions above (see Figure 1.2).

We have implemented a set of prototype tools. We target C++, since the C++ community is most concerned with the exception safety classification. The main tool is called BANGSAFE. It takes a C++ source-text and returns for each subroutine the corresponding exception-safety guarantee. The different tools in the set allow presenting and visualizing the various stages of the analysis in different ways. BANGSAFE itself is implemented in Ruby.

Parsing C++ code is no easy task. We therefore use a third-party program. The parser is called ELSA, and is developed by McPeak et al. [12, 11]. Since our tool is a prototype, many internal datastructures depend on the datastructures of ELSA. Therefore, the parser at the backend of the tools cannot simply be switched.

1.4 Outline

In Chapter 2, all material necessary to understand the rest of the thesis is described. Chapter 3 provides an overview of the overall architecture and the set of tools developed. Chapter 4 delves deep into the design and implementation of the algorithm, from the C++ input to the exception-safety guarantees emitted. Chapter 5 presents a set of examples, which show both cases where the analysis works and where it reports false positives. Conclusions are given in Chapter 6.

Chapter 2

Background

This section provides the required background material for understanding the rest of the thesis. In Section 2.1, we start with a discussion of exception handling. The discussion focuses on the history and design of exception handling, ending with a description of their syntax and semantics in C++. Section 2.2 continues with related work on exception safety. Static analysis of programs is another major part of the thesis. Section 2.3 explains the terminology, together with some standard analysis algorithms that we utilize. To reason about our proposed algorithm, we use an algebraic structure, namely a semiring. The mathematical notation used, including concepts from abstract algebra and graph theory, is the subject of Section 2.4.

2.1 Exception handling

The first part of the background provides detailed insight into the history and development of exceptions up until C++, and also discusses the semantics of exceptions in C++.

2.1.1 History of exception handling

We provide a brief summary of the history of exceptions, including the most important persons and programming languages involved, up until the introduction of C++. Ryder et al. provides a more extensive discussion of the influences of software engineering research on the development of exceptions [15].

As Ryder argues, the first languages to contain exception-like constructs were Lisp 1.5 and PL/I. In particular the support in PL/I is similar to modern exceptional-handling constructs, but proved difficult to use.

Goodenough was the first to provide a systematic view on exception handling [9]. He described the exception handling mechanism and the different design issues of introducing it into a programming language. We still use the terminology that he introduced:

[..] *exception conditions* are those brought to the attention of the operation's invoker. [...] Bringing an exception condition to an invoker's attention is called *raising an exception*. The invoker's response is called *handling* the exception.

Goodenough's version of exceptions was not necessarily designed for error-conditions, as are most exception handling constructs of today. In his design he envisioned that exceptions would be useful as a means for communicating between different levels of abstraction. Therefore, it was beneficial if handlers could *terminate* as well as *resume* an operation. In most modern programming languages, handlers are not allowed to resume after an exception has been raised.

In the same year, Randell realized the importance of programming language design for creating fault-tolerant systems and defined a structured mechanism called *recovery blocks* [13]. The mechanism is similar in style to exceptions, but includes support for automatic recovery. The description of recovery blocks is more geared towards performing the same task in several different ways, whereas exceptions include only one normal execution path, with attached flows to deal with specific exceptions that might occur.

In the late 1970s, the two languages CLU and Ada emerged, which both included exception handling support. Both languages were designed with Goodenough's work in mind, but they both abandoned the resumption model, and only allowed to terminate from the handler.

2.1.2 Exception handling in C++

In the early 1990s, exceptions were designed and implemented for C++ [17]. Contrary to earlier designs, exception handling in C++ aimed specifically at dealing with error handling. One can therefore assume, that exceptions only arise on rare occasions, which, for example, allows compilers to optimize the normal execution flow. C++ also introduced the model of exceptions as objects. This essentially means that an exception can include arbitrary information, where previously exceptions had just been identified by names.

Figure 2.1 shows the syntax of C++ exceptions. A *try* block delimits a list of statements for which emitted exceptions are handled by the associated

```
try {  
    throw <exception-object>;  
}  
catch (<exception-type>) {  
    <handler-implementation>  
}
```

Figure 2.1: Exception-handling in C++

catch clauses. Raising is performed using the *throw* expression. Since exceptions are objects, catch clauses can rely on the C++ type system to specify what exceptions it can deal with.

As in both CLU and Ada, C++ only allows for termination semantics. When a catch clause handles a specific exception, it can decide either to terminate the try block or to reraise the exception. If the handler terminates the try block, execution will continue after the try block and all its attached catch clauses. An unhandled exception is propagated up the call chain.

In C++, the philosophy is that one should not have to pay for exceptions until one is thrown. In the beginning, this has not been true, since the control flow introduced by exceptions complicates much of the optimizations one can normally do [8]. Recent compilers are better than previous generations with implementing exception handling, though.

2.1.3 Exception analysis

In our analysis, the control flow introduced by exceptions is of crucial importance. Robillard et al. describes a process similar to our approach, but his algorithm is for Java [14]. The algorithm defined by Robillard was inspired by a similar algorithm for Ada, presented by Schaefer et al. [16].

The description of our control-flow graph construction in Section 4.3 was inspired mostly by Robillard's approach, but adapted to C++ semantics. The exception semantics are very similar in Java and C++. In fact, C++ is simpler in that there are fewer ways an exception can be introduced. The most significant difference is in determining whether an exception can be caught or not. In Java, exceptions are subtypes of *Throwable*, so that one can rely on subtyping. In C++, any object can represent an exception. The catching semantics in C++ is therefore similar to argument binding.

2.2 Exception safety

The fundamental idea behind exception safety is to classify procedures according to their behavior in the presence of exceptions. This idea is not a new one. Already in the late 1970s, Cristian reasons about classifying procedures as weakly and strongly tolerant towards exceptions [5, 6]. In difference to the exception-safety guarantees, Cristian's tolerance levels are specified on an exception-by-exception basis. If a procedure fulfills the requirements of the strong exception-safety guarantee, Cristian's classification specifies it as weakly tolerant towards all exceptions occurring during its execution. A procedure that is strongly tolerant towards an exception can actually fix the problem that the exception signals. There is no corresponding guarantee in the exception safety classification.

Abrahams discusses the knowledge accumulated while implementing the C++ standard library [1]. The intention was to ensure that the standard library exhibits a useful and well-defined behavior when interacting with exceptions. He describes the notion of exception safety as a way to ensure that a component behaves 'reasonably' in the event of exceptions. In the early days, the C++ standard library specified that if client-supplied code caused exceptions, the behavior of the library becomes undefined. Abrahams and Colvin proposed a better way of defining the behavior, introducing the guarantees in Figure 1.1 [2, 3].

The idea of exception safety in C++ was not new, but the problem with template-based libraries such as the C++ STL, is that the exception-behavior depends on client-supplied code, and the classification was introduced as a contractual specification between different components. Nowadays, the exception safety classification is established and widely used. The classification is used, for example, during the review process in the Boost organization for standardized libraries in C++ [7]. Designers get help with idioms and guidelines, most notably by Sutter, who has published three books on exceptional C++. In his books he discusses the issues involved in exception handling in general and exception safety in particular [20]. Another source for designers is Stroustrup's tutorial description of the exception-safety guarantees of the C++ standard library [18]. The tutorial suggests ways how to achieve the guarantees, and how to best use the standard library together with exceptions.

The implementation of exception-safe STL that Abrahams discusses is called STLport. In combination with its implementation, he implemented a test-suite that automatically checks that the appropriate exception-safety guarantees are fulfilled [1]. To our knowledge, this is the only previous automatic approach to enforce exception safety. The test-suite is cleverly de-

	basic	strong	no-throw
pure	t	t	ϵ
impure	s	u	m

Table 2.1: Relation between Alexandrescu’s abstract representation and ours. Our abstract representation is explained in Section 4.4 and consists of five different classifications (ϵ, m, t, u, s).

signed and specifically targeted at the C++ standard library. The specific target makes it possible to ensure that even the basic guarantee holds. Unfortunately, the test-suite is of little help when testing another program.

Alexandrescu and Held describe a manual procedure to detect the exception-safety guarantee fulfilled by a procedure [4]. Their procedure is similar to the procedure defined in this thesis. Both approaches develop similar abstract representations, and both procedures process the analyzed procedure in increments, updating the intermediate representation appropriately. However, Alexandrescu’s approach is intended to be performed by humans, and therefore contains some vague and unspecified details.

Alexandrescu’s abstract representation includes two concepts, namely the exception-safety guarantee achieved so far, and the purity of a procedure. A procedure is pure if it can neither modify program state nor cause side effects. In difference, our abstract representation is a classification of the control flow of a procedure into five different equivalence classes (see Section 4.4). Table 2.1 lists the relation between our representation and Alexandrescu’s. As the table shows, for every combination of Alexandrescu’s properties we have a corresponding classification. There is one exception, though: for pure procedures, both basic and strong guarantee are represented by t . This is because a procedure that does not perform any state changes or side effects cannot invalidate the strong guarantee. Alexandrescu also describes this notion in his summary, saying that “pure functions are always strong.” The major difference between our and Alexandrescu’s approach, is that the one we describe is automated, which requires more attention to detail.

In our approach, we clearly separate the notion of side effects and state changes and focus on the latter. The separation of the two was inspired by a Sutter’s classification of exception safety using database terminology [19]. Since our approach does not deal with side effects, they must be encoded as state modifications, to obtain correct results.

2.3 Static Analysis

Static analyses can be classified in various ways. This section summarizes the most important classification criteria and standard structures.

2.3.1 Terminology

A major distinction between algorithms for static analysis is whether they deal with a problem intra- or interprocedurally. Intraprocedural means that the algorithm deals with one procedure at a time and either pessimizes over subroutines or disallows them. An interprocedural analysis, on the other hand, uses information about called procedures. Most algorithms deal with their problem in an intraprocedural manner. In our approach, the information that must be propagated from callee to caller is very small, and works well in an interprocedural manner.

Another distinction concerns whether context is considered or not. An analysis that takes the context of an operation into account is called context-sensitive. Our analysis is context-insensitive on the procedure level, since the exception safety classification of a procedure is independent of the context in which the procedure is called.

Many static analyses perform whole-program analysis, which means that the source text for the whole program must be available. This is also the case for our approach. If the source code for libraries or procedures are not available, they would need to be replaced by another implementation that provides the same exception-safety guarantee and purity as the original implementation would.

2.3.2 Standard structures

An *abstract syntax tree* is a finite, rooted tree, where the nodes correspond to non-terminals from the grammar specification of the input language. This means that the nodes all represent abstractions of syntactic constructs of the source text. The tree has an edge from a to b if b is part of the larger construct a .

Many analyses lower the abstract syntax tree to a *control-flow graph*, which is a digraph where an edge from a to b represents the transfer of control from a to b . Throughout this thesis, b will be called the *successor* of a if there is an edge from a to b in the control-flow graph. The nodes of a control-flow graph are usually the basic blocks or statements of the input program. In our analysis, the nodes correspond to expressions of the input language.

2.4 Mathematics

Within the thesis, we make use of some mathematical concepts when reasoning about the properties of the algorithm. The major fields from which we have drawn definitions include abstract algebra, and graph theory.

2.4.1 Abstract algebra

We make use of the semiring structure from the field of abstract algebra. Hebisch et al. defines the notion of a semiring as a refinement of semigroups and binary operations [10].

A binary operation, \odot , over a set, S , is a binary function taking two elements from S as input and producing an element of S as output:

$$\odot : S \times S \rightarrow S.$$

A binary operation is *associative* if the order of applying the operation is unimportant

$$\forall a, b, c \in S : a \odot (b \odot c) = (a \odot b) \odot c,$$

commutative if the ordering of the inputs does not matter,

$$\forall a, b \in S : a \odot b = b \odot a,$$

and *idempotent* if

$$\forall a \in S : a \odot a = a.$$

If there exists a neutral element, $\epsilon \in S$, such that

$$\forall a \in S : \epsilon \odot a = a \odot \epsilon = a,$$

then ϵ is called the *identity element* of \odot . A binary operation \odot is distributive over \oplus if

$$\forall a, b, c \in S : a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c).$$

A *semigroup* is a pair (S, \odot) where \odot is an associative operation over the set S . If the semigroup has an identity element, ϵ , then (S, \odot, ϵ) is a monoid. A *semiring*, (S, \oplus, \otimes) , over the set S is defined as:

- (S, \oplus) , is a commutative semigroup.
- (S, \otimes) , is a semigroup.
- \oplus is distributive over \otimes and vice versa.

Optionally, a semigroup can define a *zero element* and an *identity element*:

- ϵ is the zero iff (S, \oplus, ϵ) is a monoid.
- i is the identity iff (S, \otimes, i) is a monoid.

For a *semiring* that is idempotent under \oplus , one can define a *strict partial order*, $<$, over S as

$$\forall a, b \in S : a < b :\Leftrightarrow a \neq b \wedge a \oplus b = b.$$

An ordering is strict if

$$\forall a \in S : \neg(a < a),$$

and partial, as opposed to *total*, if

$$\exists a, b \in S : \neg(a < b) \wedge \neg(b < a) \neq a = b.$$

2.4.2 Graph theory

In graph theory, one distinguishes between directed and undirected graphs. In this thesis, all graphs discussed are directed graphs or *digraphs*; for brevity, we denote them just graphs.

A digraph G is an ordered pair (V, A) , where V is a set of nodes and A is a set of directed edges. An edge, (u, v) , connects the two nodes $u, v \in V$, and is directed from u to v . A path in a digraph is a sequence of nodes u_1, u_2, \dots, u_n , such that:

$$\forall i \in \{1, 2, \dots, n-1\} : (u_i, u_{i+1}) \in A \vee (u_{i+1}, u_i) \in A,$$

and is called *directed* if all edges go in the same direction, i.e.:

$$\forall i \in \{1, 2, \dots, n-1\} : (u_i, u_{i+1}) \in A.$$

A *tree* is a graph where there is exactly one path connecting any two nodes. A *rooted tree* is a directed graph, where a particular element is the root of the tree. There exists a directed path from the root node to all other nodes. The parent of a node v is the node u if the edge $(u, v) \in A$. The children of u are all nodes v , such that $(u, v) \in A$. The nodes u and v are called neighbors if they are children of another node w .

A *depth-first traversal* or depth-first search traverses the nodes of a graph by visiting all children of a node before visiting the node's neighbors. This traversal is defined mainly for trees. By choosing any node of a graph as the root node, however, it can be applied to a subset of ordinary graphs as well.

A *strongly-connected component* of a digraph is a maximal set of vertices in which there is a path from any one vertex to any other vertex in the set.

Chapter 3

Architecture

Our analysis is realized as a series of transformations. To get from the initial C++ input to the output containing the exception safety classification, the data goes through a series of transformations. For each transformation, a separate utility is in charge. In this section we give an overview.

Three major intermediate stages can be identified. These transformations are depicted in Figure 3.1. The first stage parses the input into an abstract syntax tree. This stage is handled by the third-party program, ELSA, which produces the abstract syntax tree in XML form. In the next stage, a control-flow graph is constructed using the information from the abstract syntax tree. Continuing, the control-flow graph is annotated with our abstract representation. The final, and most important step, is the application of our analysis-algorithm to the annotated control-flow graph to produce the exception safety classification.

The transformations are implemented as a series of Ruby command-line applications. Figure 3.2 shows the dependencies among the different applications. It is worth noting that the dependencies do not exactly match the straight line of transformations described in Figure 3.1 since the stages can share data structures, thus need not be separated through program interfaces. The functionality, however, is clearly separated for the different transformations and implemented as separate procedures on the same data

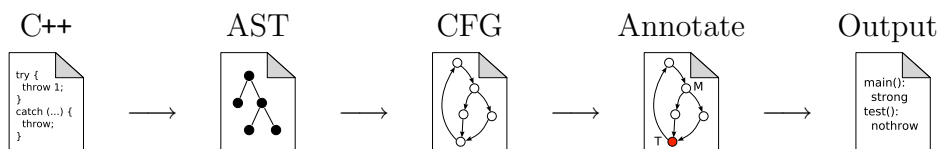


Figure 3.1: Transformations involved in the analysis

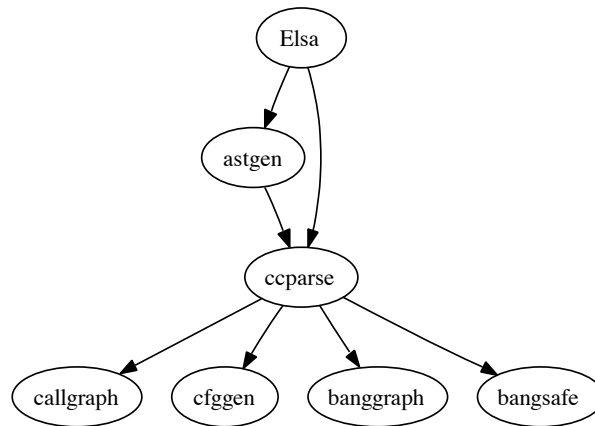


Figure 3.2: Dependencies within the toolset

structures.

ASTGEN Generates Ruby classes from ELSA’s abstract syntax tree specifications. ELSA has a tool by the same name that generates C++ classes.

CCPARSE Wraps the ELSA program by the same name, and transforms the XML that ELSA emits into an object hierarchy using the classes generated by ASTGEN. The program then serializes the object hierarchy using standard Ruby serialization.

CALLGRAPH Reads the object hierarchy produced by CCPARSE and extracts the callgraph by checking all nodes in the abstract syntax tree that correspond to a procedure call.

CFGGEN Reads the object hierarchy produced by CCPARSE and creates a control-flow graph for every procedure in the program.

BANGGRAPH Works like CFGGEN, but additionally annotates the control-flow graph with information about the state modifications and thrown exceptions. The output of this utility is trimmed to only include nodes that are relevant for exception handling.

BANGSAFE The most important tool. Works very similar to BANGGRAPH except that instead of outputting the annotated control-flow graph, it applies the analysis to the graph, and outputs the exception safety classification in terms of our abstract representation for each procedure in the input. If the classification is such that the strong guarantee is invalidated, the output is augmented with the execution-paths leading to the invalidation of the strong guarantee.

Chapter 4

Approach

This chapter contains detailed descriptions of the different transformations outlined in Chapter 3. First of all, in Section 4.1, a running example is introduced, which is used throughout the section. Following, in Section 4.2, is a description of the supported input and the abstract syntax tree construction. Section 4.3 details the construction of the control-flow graph based on the abstract syntax tree. The abstract representation used in the analysis is presented in Section 4.4. Section 4.5 describes how the control-flow graph from Section 4.3 is annotated using the abstract representation from Section 4.4. The most important part of this section is Section 4.6, where the main analysis is presented. The chapter finishes with a description of the synthesis process, which, in case of an invalidation of the strong guarantee, extracts the paths that lead to the violation and returns them to the user (Section 4.7).

4.1 Running example

When describing the different steps of the analysis, the example in Figure 4.1 will be used. The example is relatively short, to ensure that the results are presentable.

The example is an allocation-function, which allocates from a fixed-size buffer of characters. If there is no more room in the buffer, an exception will be thrown. The example includes a branch, a throw, and a state modification, to ensure that the effect of all stages of the algorithm is visible.

```
1  const unsigned int SIZE = 10;
2  char pool[SIZE];
3  char *next = pool;
4
5  char *pool_alloc() {
6      if (next == pool + SIZE) {
7          throw "Out of memory!";
8      }
9      return next++;
10 }
```

Figure 4.1: Running example source code

4.2 Parsing the input

The first stage of the process converts the C++ input into an abstract syntax tree that can be further analyzed. Figure 4.2 shows the grammar of the supported target language. The described language is characterized in two ways:

- It focuses on the features of exception handling.
- It is a subset of executable C++, to ensure that one can use standard parsers, and that the algorithm is applicable to C++.

The input actually supported by the toolset includes those features that have a simple and straight-forward transformation to the ones described in Figure 4.2. To keep the grammar readable, we have not listed those features.

We have chosen to use ELSA as the C++ parser. ELSA is a C++ parser developed by McPeak as an attempt to create a “general-purpose parsing framework and C++ parser for research use.” The motivation for choosing ELSA was partly to support this idea. More importantly, the code-base is quite small and extensible, and it is possible to produce easily parseable XML-output.

ELSA claims to support most of standard C++, but we have not verified that claim. Therefore, it is not clear exactly how much of C++ is supported by the BANGSAFE toolset. ELSA seems to be able to parse most of C++, but to what degree it properly performs type-checking, overload-resolution, etc. is not completely obvious.

ELSA produces an abstract syntax tree from the C++ input. The created tree contains not only syntactical information, but also type-checking information such as resolving the actual procedure called from a specific call-site.

$\langle \text{program} \rangle ::= \langle \text{function} \rangle^*$

$\langle \text{function} \rangle ::= \langle \text{type} \rangle \langle \text{identifier} \rangle '(' \langle \text{argument} \rangle^* ') \langle \text{compound} \rangle$

$\langle \text{argument} \rangle ::= \langle \text{type} \rangle \langle \text{identifier} \rangle$

$\langle \text{compound} \rangle ::= \{' \langle \text{statement} \rangle^* \}'$

$\langle \text{statement} \rangle ::= \langle \text{compound} \rangle \mid \langle \text{loop} \rangle \mid \langle \text{branch} \rangle \mid \langle \text{try} \rangle \mid \langle \text{expr} \rangle ';'$

$\langle \text{loop} \rangle ::= \text{'for' } '(' \langle \text{expr} \rangle ';' \langle \text{expr} \rangle ';' \langle \text{expr} \rangle ') \langle \text{compound} \rangle$

$\langle \text{branch} \rangle ::= \text{'if' } '(' \langle \text{expr} \rangle ') \langle \text{compound} \rangle \text{'else' } \langle \text{compound} \rangle$

$\langle \text{try} \rangle ::= \text{'try' } \langle \text{compound} \rangle \langle \text{handler} \rangle^+$

$\langle \text{handler} \rangle ::= \text{'catch' } '(' \langle \text{argument} \rangle ') \langle \text{compound} \rangle$

$\langle \text{expr} \rangle ::= \langle \text{literal} \rangle \mid \langle \text{operator} \rangle \mid \langle \text{call} \rangle \mid \langle \text{throw} \rangle \mid \langle \text{assignment} \rangle$

$\langle \text{operator} \rangle ::= \langle \text{op} \rangle \langle \text{expr} \rangle^+$

$\langle \text{call} \rangle ::= \langle \text{identifier} \rangle '(' \langle \text{expr} \rangle^* ')'$

$\langle \text{throw} \rangle ::= \text{'throw' } \langle \text{expr} \rangle$

$\langle \text{assignment} \rangle ::= \langle \text{expr} \rangle '=' \langle \text{expr} \rangle$

Figure 4.2: Grammar of the accepted subset of C++

The XML generated by ELSA contains lots of information, including type annotations and resolved procedure calls. For our simple running example about 2000 XML tags are emitted and the total filesize is almost 300kB.

ELSA is written in C++ and uses a hierarchy of C++-classes to represent the different syntactic constructs. The abstract syntax tree that ELSA creates is a hierarchy of objects of these different classes. This object hierarchy provides the interface one uses to access information about the tree. Ultimately, ELSA outputs the type-annotated abstract syntax tree as an XML-hierarchy, where every XML-node corresponds to a C++-object in the object hierarchy, and the C++-properties are exported as XML-attributes.

From there, our analysis takes over. Since our analysis is written in Ruby, the XML output is converted into a hierarchy of Ruby objects. While loading the XML description into Ruby, all cross-references are resolved, so that the structure within Ruby is the same as it was in C++.

4.3 Creating the control-flow graph

To later be able to compute a control-flow graph that works for propagating exceptions between procedures, we must know what exceptions a procedure-invocation might cause. We traverse the procedures in such an order that the callee is always inspected before the caller. If the procedure contains recursive function calls, we would not be able to define such an ordering. Therefore, recursion is not allowed.

We create a call graph that contains a node for every procedure, and an edge between two nodes f and g iff procedure f calls procedure g . The call-graph is constructed by searching through the abstract syntax tree corresponding to every procedure, and adding the appropriate edges. Using this graph, we can resolve the desired ordering of procedures by performing a depth-first traversal of the call-graph.

The remaining steps of the analysis utilize the procedure-ordering established above, and thus only process a single procedure at a time. For each procedure we first construct its control-flow graph. This control-flow graph must include the following information:

expression-level To ensure that we know the execution order of different expressions, we use the C++ expressions from the abstract syntax tree as the nodes of the control-flow graph. In later stages of the analysis, we need to determine whether a state modification happens before an exception is thrown. In C++, both state modifications and throws are caused by expression-level constructs.

exceptional-flow To ensure that we encapsulate all exceptional flow, we process the nodes in such an order that we can keep track of all active throw-sites, and insert the appropriate jump when we encounter a suitable handler or reach the end of the procedure. Whether a handler is suitable, is determined by the specification in the C++ language standard.

We cannot statically determine the exact type of a thrown exception, thus in some cases we must assume that an exception is both caught and not by a specific handler.

To be able to deal with interprocedural exceptions, we keep track of a mapping between a procedure and its uncaught exceptions. This mapping will also be used while annotating the control-flow graph.

The constructed control-flow graph is an adjacency-list graph, which means that every node keeps track of its successors. To facilitate the construction of the graph, two separate nodes are created for every entity, an entry-node and an exit-node. Introducing extra nodes simplifies the construction, since an exit node will always be connected either with a neighbor or its parent's exit node. For complicated graphs, it is also easier to check that the construction works as intended.

The top-level node of the abstract syntax tree represents the whole procedure, so that the entry-node of the control-flow graph will represent the procedure and the exit-node of the graph will correspond to the exit-node of the procedure.

The actual algorithm for constructing the control-flow graph is a depth-first traversal of the abstract syntax tree. The implementation is described in terms of four major procedures, namely *compute_cfg*, *examine_ast_node*, *compute_edges*, and *traverse_ast_children*.

We will discuss each procedure, but before we do that, we must introduce some conventions used in their descriptions. The notation $x[y]$ means that x is a lookup-table, and that we are looking up the entry indexed by y in this table. The original abstract syntax tree nodes are used throughout the control-flow graph to denote the entry-nodes. The corresponding exit-node is retrieved by calling the exit procedure. Some properties are expected to be attached to the abstract syntax tree nodes. These properties are accessed using dot (.) notation. When constructing the graph, the following procedures are used:

starts_with(u, v, g) The edge (u, v) is added to the graph g . This means that control entering the construct that u represents will continue into the construct that v represents.

```

def compute_cfg (procedure, uncaught, cfg)
  active_raises = ∅
  examine_ast_node(procedure, active_raises, uncaught, cfg)
  for raise in active_raises
  |   raise_leads_to(raise, exit(procedure), cfg)
  end
  uncaught[procedure] = active_raises
end

```

Figure 4.3: Pseudo-code for *compute_cfg*

leads_to(u, v, g) The edge ($exit(u), v$) is added to the graph g . This means that when control leaves the construct that u represents it will continue into the construct that v represents.

We now turn to the description of the four major procedures.

4.3.1 compute_cfg

This is the top-level procedure, which creates the control-flow graph of *procedure* in *cfg*. When complete, a mapping between *procedure* and the currently active raises will be added to the *uncaught* lookup-table.

From the implementation in Figure 4.3, it is clear that most of the functionality is delegated to *examine_ast_node*. The procedure takes the following parameters.

procedure The procedure currently under inspection.

uncaught A map from a procedure to the exceptions that it might emit. This mapping must contain entries for all procedures that are called within the current procedure.

cfg A representation of the control-flow graph, which is initially empty. During the traversal of the abstract syntax tree, nodes are added corresponding to the abstract syntax tree nodes, and edges will be added corresponding to a possible transfer of control.

4.3.2 examine_ast_node

This procedure is responsible for computing the edges for a node and all its abstract syntax tree children. The actual implementation is performed by the two helper procedures *compute_edges* and *traverse_ast_children*.

```
def examine_ast_node (node, active_raises, uncaught, cfg)
|   compute_edges(node, active_raises, uncaught, cfg)
|   traverse_ast_children(node, active_raises, uncaught, cfg)
end
```

Figure 4.4: Pseudo-code for `examine_ast_node`

Since the procedure just delegates work, the implementation in Figure 4.4 is fairly straight-forward. The procedure accepts the following parameters.

node The abstract syntax tree node currently under inspection.

active_raises The set of currently active raised exceptions, which is initially empty. Elements are added and removed from this set during the traversal of the abstract syntax tree. The elements remaining in this set after the whole procedure has been traversed are the uncaught exceptions.

uncaught A map from a procedure to the exceptions it might emit. This mapping is not changed during the traversal of the abstract syntax tree, and is only used to determine whether an exceptional control-flow graph edge should be added from a procedure call.

cfg as in `compute_cfg`.

4.3.3 `compute_edges`

This procedure is responsible for constructing all 'internal' edges of this node. Intuitively this means connecting the node-entry with its children, and connecting the exits from children with the node-exit.

The procedure is also responsible for interconnecting different child-nodes. In most cases, this means connecting the children in a sequence, but determining the internal edges is more complicated for branches, loops, and exception-related constructs. Inside try-statements, the start of a handler is initially not connected, since at this point we do not know whether the handler will ever be reached. For throws and calls to raising procedures, an element is added to the set of raised exceptions. For invocations, all arguments are joined in a cycle, to ensure that every argument precedes all other arguments, since the evaluation order of parameters is not specified in C++.

Figure 4.5 provides the implementation of this procedure. The parameters are exactly the same as for `examine_ast_node`.

```

def compute_edges (node, active_raises, cfg)
  case type of node
  when Loop
    starts_with(node, node.init, cfg)
    leads_to(node.init, node.test, cfg)
    leads_to(node.test, node.body, cfg)
    leads_to(node.body, node.increment, cfg)
    leads_to(node.increment, node.test, cfg)
    leads_to(node.test, exit(node), cfg)
  when Branch
    starts_with(node, node.test, cfg)
    leads_to(node.test, node.then, cfg)
    leads_to(node.then, exit(node), cfg)
    leads_to(node.test, node.else, cfg)
    leads_to(node.else, exit(node), cfg)
  when Try
    starts_with(node, node.compound, cfg)
    for handler in node.handlers
    | leads_to(handler, exit(node), cfg)
    end
  when Throw
    starts_with(node, node.expression, cfg)
    active_raises.add(node)
  when Invocation
    starts_with(node, node.arguments, cfg)
    leads_to(node.arguments, node.arguments, cfg)
    leads_to(node.arguments, exit(node), cfg)
    unless uncaught[node.func].empty?
    | active_raises.add(node)
    end
  else
    starts_with(node, node.children.first, cfg)
    for current, next in node.children
    | leads_to(current, next, cfg)
    end
    leads_to(node.children.last, exit(node), cfg)
  end
end

```

Figure 4.5: Pseudo-code for *compute_edges*

```
def traverse_ast_children (node, active_raises, cfg)
  case type of node
  when Try
    examine_ast_node(node.compound, cfg)
    for handler in node.handlers
      caught =  $\emptyset$ 
      for raise in active_raises
        if catches(handler.type, raise)
          raise_leads_to(raise, handler, cfg)
          caught.add(raise)
        end
      end
      active_raises = active_raises - caught
    end
    for handler in node.handlers
      examine_ast_node(handler, cfg)
    end
  else
    for child in node.children
      examine_ast_node(child, graph)
    end
  end
end
```

Figure 4.6: Pseudo-code for *traverse_ast_children*

4.3.4 *traverse_ast_children*

This procedure is a convenience procedure for calling *examine_ast_node* on all children of the current abstract syntax tree node.

try/catch-blocks represent the only complication since there we must process the handlers separately. For every handler, we add edges from the point where the exception is raised to the point where it is caught. We also remove all exceptions that are definitely caught from the set of active exceptions.

The implementation of this procedure is in Figure 4.6. The parameters are exactly the same as for *examine_ast_node*.




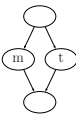

ϵ		The graph contains neither state-changes nor throw-expressions.
m		The graph contains a state modification, but no throw-expression that exits the procedure.
t		The graph contains a throw that exits the procedure, but no state modification occurs.
u		The graph contains a state modification and a throw that exits the procedure, but only one of them can happen since they are in different execution branches.
s		The graph contains a state modification that is followed by a throw that exits the procedure.

Table 4.1: Abstract representation

4.3.5 Running example

Constructing the control-flow graph for the running example produces the image in Figure 4.7. The graph has been filtered to only include the entry nodes of the graph, which ensures that the result is small and easier to read. Without the filter all nodes would have an exit counterpart, which would show that the graph does not only reflect the dependencies between expressions, but also that the nesting from the abstract syntax tree remains.

The example also shows that the generated expression-level control-flow graph can become quite large even when the C++ input is very small.

4.4 Abstract representation

As always in static analysis, the key is to find an abstract representation that is simple enough to get the job done efficiently, but still ensures that the results can be used to interpret the original problem.

Of interest in our case are state modifications, throw-expressions, and their ordering. By varying only these parameters, and using the grammar specified in Figure 4.2, we can reduce the set of all possible control-flow graphs to five equivalence classes. The abstract representation can then be interpreted as a classification of a control-flow graph into one of the equivalence classes. Table 4.1 lists the five equivalence classes, including a description of the characteristics and a signifying example.

Each equivalence-class maps naturally to one corresponding exception-safety guarantee. The equivalence-classes ϵ and m map to the no-throw

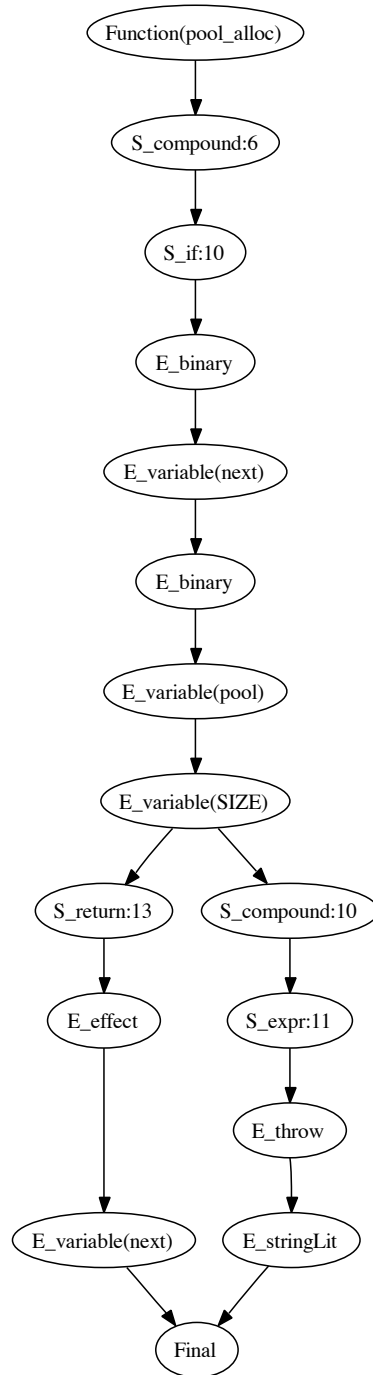


Figure 4.7: Control-flow graph generated for the running example

	ϵ	t	m	u	s
ϵ	ϵ	t	m	u	s
t	t	t	s	s	s
m	m	s	m	s	s
u	u	s	s	s	s
s	s	s	s	s	s

(a) *sequence*

	ϵ	t	m	u	s
ϵ	ϵ	t	m	u	s
t	t	t	u	u	s
m	m	u	m	u	s
u	u	u	u	u	s
s	s	s	s	s	s

(b) *branch*

Figure 4.8: Join operations on control-flow graphs

guarantee, and the equivalence-classes t and u map to the strong guarantee. Each guarantee can be represented by two different equivalence-classes, one with state modification in normal execution and one without. Table 2.1 describes the relation between this representation and Alexandrescu’s purity-based representation.

We introduce two binary operators to join control-flow graphs.

sequence Corresponds to the sequential execution of two graphs. This scenario occurs for most of the grammar rules (see Figure 4.2).

Sequencing captures the case where an m is followed by a t or vice-versa. Also note that a series of consecutive m (or t) can be abstracted to a single m (or t).

branch Represents a choice between the execution of two alternative graphs. This operator originates from the branch statement in the grammar (see Figure 4.2).

In our analysis we are interested in possible violations of a guarantee. In most cases we can therefore discard one of the branches and just keep the worst. There is one exception, though, namely when one branch leads to m and the other to t . Since both branches are equally ‘bad’, we must introduce a new symbol u to denote this type of structure.

Figure 4.8 defines formally the binary operations *sequence* and *branch* on the equivalence classes, $S = \{\epsilon, m, t, u, s\}$, listed in Table 4.1. The definition follows the natural description of the two operators. For both *branch* and *sequence* it is intuitively so that any x combined with ϵ remains x . The *sequence* operation for u is defined as:

$$\forall x \in S : \text{sequence}(u, x) \equiv \text{branch}(\text{sequence}(m, x), \text{sequence}(t, x)),$$

which is natural, since we want the worse alternative of two branches.

Given these definitions, we can define a semiring for our abstract representation, $(S, \text{branch}, \text{sequence})$, which additionally fulfills the requirements of the following algebraic structures:

- $(S, \text{sequence})$ is a commutative idempotent monoid with identity ϵ .
- (S, branch) is idempotent.

Since the semiring is idempotent, we can derive a partial order by defining

$$a < b \Leftrightarrow a \neq b \wedge \text{branch}(a, b) = b,$$

which results in the following natural ordering of the classes

$$\epsilon < m, t < u < s,$$

which, in turn, matches the intuitive 'badness' ordering. The ordering is used when discussing the analysis, to prove that the algorithm terminates.

In the tables above there are some situations that cannot occur for the input language we have specified, most notably

$$x \neq \epsilon : \text{sequence}(t, x),$$

since a t annotation represents an exiting exception, thus cannot be sequenced with another graph. By supporting automatic destructors, the exiting exception must not be the last expression. When adding support for automatic destructors one would probably be able to distinguish t sequenced with t , since this would result in program termination in C++.

Using above classification we have neither terminology to describe loops, nor do we have any loop operation. Yet, a loop in control-flow graph form is similar to a branch, except for an additional back-edge. The abstract representation does not explicitly deal with loops, but it will become apparent in later stages of the analysis that the operations above suffice.

4.5 Annotating the control-flow graph

Section 4.4 defined the abstraction, which we can now apply to the control-flow graph created in Section 4.3. We annotate the different expressions of the control-flow graph with an m if they constitute a state modification and with a t if they throw an exception that exits the procedure.

In C++, the program state can only be modified by a few language constructs:

- Assignments (`a = b`)
- Operators with side effects (e.g., `a++`)
- Function calls

The situation for throws is the same. Only a few language constructs can *raise* an exception:

- Throw-expressions (`throw`)
- Function calls

The above list fails to mention the complication that there are many syntactic constructions in C++ that can lead to procedure calls. These include automatic type-conversions, constructor calls, new-expressions, and expressions involving overloaded operators. For our analysis, we rely on the parser to resolve these implicit procedure calls.

In the interprocedural case the annotation is slightly complicated because in some cases the callee can terminate both normally and exceptionally. The analysis must make sure to not annotate the normal exit with t . If a procedure is annotated with s , for instance, the normal exit should be annotated with m . When a procedure is annotated with s , it is not necessarily true that it modifies state in the non-exceptional case. The case where modifications only take place before terminating with a thrown exception should be very rare, though.

We are only interested in exceptions that escape a procedure, not those that are caught locally. When introducing t annotations, we must make sure to only do so for the raises that actually terminate the procedure. Since we kept track of which exceptions were caught when constructing the control-flow graph, we actually get this information for free.

The initial abstraction that a state modification leads to program-state invalidation, does not hold if the modifications only affect local variables. Thus, if a procedure modifies a local variable, we do not want this to trigger an m annotation. We use information supplied by the parser to check whether the left-hand-side of a modification is a local variable. If the variable is not of reference-type, the assignment is ignored.

This notion of locality does not recognize pointers or references to local variables. To more precisely determine whether an assignment need not yield m , one has to use some points-to analysis to ensure that a variable of pointer type cannot point outside the local procedure. Currently this is not implemented, since it would introduce further complexity. Thus, we lose precision.

```
def annotate (node)
  return  $\epsilon$  unless node is exit
  case type of node
  when Assignment
    return {  $\epsilon$    if node.target is local variable
           {  $m$    otherwise
  when Throw
    return {  $t$    if node.exception is uncaught
           {  $\epsilon$  otherwise
  when Invocation
    return annotate_call(node)
  end
end
```

Figure 4.9: Pseudo-code for *annotate*

Our analysis loses further precision in the interprocedural case. When computing the information for a single procedure, we do not determine what parameters were modified, and thus cannot check whether the corresponding actual arguments were local variables within this procedure.

4.5.1 Implementation

The annotations of the control-flow graph are performed by the algorithm *annotate*. This procedure needs to be able to tell whether a node is an entry node, an exit node, or an exceptional exit node. Furthermore, for the interprocedural case, it needs access to the result of the analysis in Section 4.6 for all called procedures.

Figure 4.9 describes in pseudo-code how the algorithm proceeds. As described above, it will annotate the exit-node of uncaught throw-expressions with a t and non-local state-modifications with m . The interprocedural part of the algorithm is outlined in Figure 4.10 and relies on the results from Section 4.6. A call-site will have two separate exit-nodes, one for normal return and another for thrown exceptions. The normal exit-node should never be annotated with a t annotation, and only an m annotation if the callee is impure. The exceptional exit should only be annotated with the exceptional cases, which means m annotations go away unless there was an s to begin with. For a caught exception, we only care whether the state might have been modified before the exception was thrown, thus the procedure should be annotated with m only if the callee was annotated with s .

```

def annotate_call (node)
  return  $\epsilon$  unless node is exit
  x := Analysis in Section 4.6 applied to node.callee
  case node
  when uncaught exceptional exit
    return  $\begin{cases} t & \text{if } x = u \\ \epsilon & \text{if } x = m \\ x & \text{otherwise} \end{cases}$ 
  when caught exceptional exit
    return  $\begin{cases} m & \text{if } x = s \\ \epsilon & \text{otherwise} \end{cases}$ 
  else
    return  $\begin{cases} m & \text{if } x \geq m \\ \epsilon & \text{otherwise} \end{cases}$ 
  end
end

```

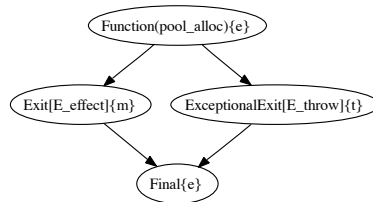
Figure 4.10: Pseudo-code for *annotate_call*

Figure 4.11: Result of annotating the running example

4.5.2 Running example

Figure 4.11 demonstrates the annotated control-flow graph of the running example. BANGGRAPH (see Chapter 3) was used to generate the figure. Its output was filtered to skip all intermediate nodes that are annotated by ϵ . The filtering ensures that the resulting graph is small and easy to read. Note that the annotations are denoted by capital letters in the output, and that ϵ is replaced by N.

We can see that the exit-node for the side-effect operator (`next++`) was properly annotated with m , since it will modify the non-local state encapsulated by the variable `next`. We can also see that the exceptional exit of the throw was annotated with a t , since this raises an exception that is not caught locally.

```

def local_update (node, cfg, states)
  succ_state =  $\epsilon$ 
  for successor in adjacent_nodes(node, cfg)
  | succ_state = branch(succ_state, states[successor])
  end
  states[node] = sequence(annotate(node), succ_state)
end

```

Figure 4.12: The local update rule, applied to every node of the annotated control-flow graph

4.6 Analysis

Once we have the annotated control-flow graph, we are ready to apply our analysis, which will condense the graph into a single annotation, and thus answer the questions in Figure 1.2. We define a local update rule, which we apply incrementally for every node of the control-flow graph until we obtain an annotation for the whole procedure.

In Section 4.5, we described how to obtain an annotation appropriate for each node of the control-flow graph. In this section we produce a different annotation for each node, v , that is appropriate for the maximal subgraph reachable from v . The maximal reachable subgraph is defined as a subgraph of the control-flow graph such that it includes all paths in the graph that start with v .

Figure 4.12 describes in detail how the local update works. The branching operator is applied to the previously recorded annotations, $states[x]$, for all adjacent nodes. Sequencing this result with the annotation for the current node yields the annotation representing the maximal subgraph reachable from the current node. This annotation is recorded so that it can be accessed without recalculation.

We apply the local update rule in a reversed depth-first traversal. This ordering is chosen to ensure that successors already encapsulate the annotation of the subgraph reachable from these nodes when we apply the local update rule.

Since the calculated annotation holds by construction for the reachable subgraph, the annotation for the procedure entry contains the final annotation of the procedure. When later inspecting another procedure that calls the analyzed procedure, we can attach this annotation to the call-site, without further computation. The single annotation for a whole procedure means the algorithm scales well to the interprocedural case and that it is compositional.

4.6.1 Partial and total correctness

For partial correctness, we must prove that the annotation of a procedure is correct. We show this by proving that the local update ensures that every node, v , gets an annotation appropriate for the maximal subgraph reachable from v .

Let $G = (V, A)$ be a graph and $v \in V$. The maximal subgraph $G_v = (V_v, A_v)$ is defined as the subgraph including all paths in G that start with v :

$$\begin{aligned} G_v &= (V_v \subseteq V, A_v \subseteq A) \\ V_v &= \{x \in V : \exists y_0, \dots, y_n \in V : \\ &\quad (v, y_0) \in A \wedge (y_0, y_1) \in A \wedge \dots \wedge (y_n, x) \in A\} \\ A_v &= \{(x, y) \in A : x \in V_v \wedge y \in V_v\}. \end{aligned}$$

A path that starts with v either consists of only v , or contains a path starting with another node y . This node y must be a successor of v and the paths that start with y can all be prefixed with v to become one of the paths used to construct G_v . We can therefore define V_v as the combination of all nodes in the maximal reachable subgraphs of v 's successors:

$$V_v = \{v\} \cup \{x \in V : \exists y \in V : x \in V_y \wedge (v, y) \in A\}$$

We divide the argumentation into two cases. First we assume there are no loops, and show that a single iteration using the update rule is sufficient. We then introduce loops and show that two traversals are sufficient.

Assuming no loops, the graph is actually a *directed acyclic graph*. By applying our depth-first traversal we visit the nodes in a reversed topological ordering. By using this ordering, we can apply the local update once for every node, v , and the calculated annotation will be appropriate for G_v . We must only prove that the local update is correct, given that the annotations of v 's successors is correct.

It suffices to consider the branch case. Because each out-edge (v, u) represents a different execution-branch, applying the *branch* operation to the annotation of V_u produces the annotation of the worst possible branch, b , according to the specification in Section 4.4. Every path through the procedure is prefixed by v , and thus *sequence*(v, b) produces the worst-case annotation for G_v .

By introducing loops, it will no longer be possible to define a topological ordering of the control-flow graph nodes. We cannot define such an ordering because we have a circular dependency in our graph. If we apply the local update in the presence of loops, the algorithm would not be able to tell the

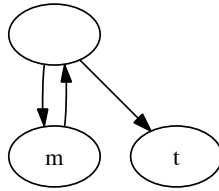


Figure 4.13: A minimal example control-flow graph, which contains a state modification followed by a raised exception that will not be caught by the first iteration of the analysis.

difference between a branch and a loop. Figure 4.13 contains the smallest possible example graph that will not be correctly classified after a single iteration of the analysis. The annotation for the whole graph should really be s in this case, but after one iteration, the analysis produces a u . The reason for this behavior is that when the node labeled m is analyzed, the t annotation has not yet propagated to the unlabeled node.

If we consider procedure-entry, however, the loops will not make any nodes reachable that were not reachable without the introduced back-edges. This means that all state modifications and thrown exceptions reachable from procedure-entry will in fact be propagated to procedure-entry by the first iteration.

As it turns out, the only annotation that is not determined solely by reachability is s (see Table 4.1). Since both a state modification and a thrown exception must be reachable for a procedure to be possibly annotated with s , the annotation from the first iteration must have been at least u , if it should ever become s .

From the argumentation above it suffices to consider how many iterations we need, to go from a u annotation to an s annotation. We first consider a single loop, where we have established u for the loop-entry by the first iteration. It is clear from the ordering and the operations, that by applying a second iteration, the annotation of the loop-entry will be s if and only if there is an m annotation within the loop. Otherwise, the annotation will stay at u . Worth noting is that since t annotations are placed only on exiting exceptions, only m annotations can be part of a loop.

If we have multiple loops, they can either be nested or in sequence. If they are in sequence, the annotations for the later loop will either remain u , in which case the second loop does not effect the first, or the annotation becomes s , which will propagate through the first loop since there is no way of going back from s (see Figure 4.8). If one loop is nested within another, the internal loop can be ignored, since the outer loop will include all ordering

information that the inner one possess.

We therefore conclude that two iterations of the above traversal of the graph ensure that the procedure-entry is annotated with the annotation from Table 4.1, which is appropriate for the procedure.

To show total correctness, we also need to prove that the algorithm terminates. In the simplest implementation we could add a loop around the condensation-algorithm and loop until we reach a fix-point. Since our abstract representation can be represented as a finite partial order and our operations are idempotent, we know that a fix-point exists and that the loop terminates. Either we reach a maximum, or we do not make any progress in one run; in either case the algorithm terminates. From the discussion of partial correctness, however, we can more precisely claim that two iterations will suffice.

4.6.2 Discussion

The complexity of the algorithm depends on the complexity of some operations applied once for every node and others applied once for every edge of the control-flow graph. The dominating operations for edges are branch-operator and state-lookup, the dominating operations for nodes are sequence-operator, state-lookup, and node-annotation.

The two operators are constant-time, whereas state-lookup and node-annotation both include looking up in a lookup-table. The complexity of table-lookup is expressed as $tl(x)$, where x is the size of the table. The worst-case complexity for the local update applied to every node of the graph is:

$$O(tl(|V|)|V| + tl(|P|)|V| + tl(|V|)|A|),$$

where $|V|$ and $|A|$ denote the number of nodes and edges in the control-flow graph, and $|P|$ is the total number of procedures in the program. By using hashables for lookup with $O(tl(x)) = 1$, we obtain the following complexity for the whole analysis:

$$O(|V| + |A|).$$

If we assume that the user of the algorithm correctly sets up his algorithm so that any side effects are modeled as explicit state modifications, then the precision of the algorithm is determined mainly by the following abstractions.

- Treating every state modification as an invariant invalidation. This essentially prevents the algorithm from detecting cases where the state

is first invalidated, and then by a later state modification revalidated. This pattern is very common in real code. However, with the enumeration described in the next section, these false-positives will in most cases be easily confirmed/refuted by a user.

- Detecting whether a variable is only local in scope, is only done syntactically. Therefore, modifications to references and pointer variables will always lead to m annotations. The precision of this part of the analysis has been discussed in more detail in Section 4.5.
- Dealing with indeterministic execution order as cycles. This ensures that all the possible orders of execution is actually encapsulated within the control-flow graph. Unfortunately it also means that an expression becomes dependant on itself. This becomes a problem if one of the arguments is a procedure call and the callee is annotated with, for example, u . The caller will then automatically be annotated s even if there is no other annotations.
- Merging several branches and always taking the worst. This will ensure that no possible problems are missed, but will in some cases result in a pessimistic judgment of a procedure. This is not a property of the algorithm, but is a property of exception safety classification in general.
- Assuming that all syntactically possible branches are executed and that all loops terminate. This means that infeasible paths are considered feasible.

4.7 Synthesis

For the program to be usable, it does not suffice to output just whether the strong guarantee was fulfilled or not. For one, a developer would normally want to know why a procedure does not fulfill the strong guarantee, so that it becomes easier to fix. At the same time, the analysis—as most analyses—can produce false positives due to its abstractions; to safely ignore these, a user needs additional information.

Once the procedure is annotated as violating the strong guarantee, we determine all possible paths that lead to invalidation. This is achieved in the simplest of ways, by enumerating all possible paths from procedure-entry to exit, and keeping the ones that contain a sequence of state modification and thrown exceptions. Since loops would result in infinitely many paths through the control-flow graph, we ensure termination by merging strongly-connected

```
> bangsafe running.dump  
  
Annotation for 'pool_alloc' is u
```

Figure 4.14: The result of BANGSAFE for the running example

components before the enumeration. Merging means that we do not list all alternative paths through loops, but once again are pessimistic and choose the worst.

The complexity of the synthesis is exponential in the number of sequential branches. If we assume that the number of sequenced branches is low, which is most probably generally the case, then the algorithm is in fact linear in the number of nodes of the control-flow graph.

We conclude by listing the output produced by BANGSAFE for the running example in Figure 4.14. The output from BANGSAFE yields that the final annotation for the running example is u , which is correct since the procedure contained two branches, where one was a throw and the other a modification of state. This means that our running example indeed provides the strong guarantee.

Chapter 5

Examples

In this section, we walk the reader through three different examples, to illustrate how the different transformations of our algorithm work. The first and the second example demonstrate correctly detected guarantees, and the last example demonstrates a false positive.

The examples that we will look at are all taken from Stroustrup [18], but have been simplified slightly. Most notably, we replaced class-based allocation by two functions (see Figure 5.1).

We also replaced the templates in Stroustrup’s examples by instantiated templates, since our tool does not support uninstantiated templates.

All three examples use implementations of a constructor for a *vector* class in the style of the C++ standard library, which is implemented using a dynamically allocated buffer and three pointers. The constructor we are looking at is supposed to allocate space for n elements and construct n copies of an initial value in the allocated space.

Since our interest in the examples is limited to the analysis of exceptions

```
template <typename T>
T* cpp_malloc(size_type n) {
    return static_cast<T*> (::operator new(n * sizeof(T)));
}

template <typename T>
void cpp_free(T* p) {
    ::operator delete(p);
}
```

Figure 5.1: Allocation functions used to replace class-based allocation

```
int g_bomb_copies = 0;
struct bomb {
    bomb(){ }
    bomb(const bomb &) {
        if (true) { throw std::bad_alloc(); }
        else { ++g_bomb_copies; }
    }
    ~bomb() { --g_bomb_copies; }
};
```

Figure 5.2: The *bomb* class, introduced to enforce a throwing copy constructor

and exception safety, we do not explain any further details of the implementation. For more information, Stroustrup's paper is a good source.

As soon becomes clear, the problem discussed within Stroustrup's examples stems from the fact that copy construction of objects can fail. To enforce failure, for demonstration purposes, we devise a *bomb* class, which is specifically designed to throw exceptions when being copy-constructed. Figure 5.2 shows the details of the *bomb* class. The additionally keeps a counter of the live *bomb*-objects, in order to make sure that the analysis identifies both the constructor and destructor call as state-modifying. We must make these explicit modifications, since our algorithm currently does not have support for constructor and destructor semantics.

5.1 The naive approach

The first example is what Stroustrup calls the naive approach. Figure 5.3 contains the implementation used in this example. There are two sources of exceptions within the procedure:

- *cpp_malloc* throws if no memory is available.
- The copy constructor of the element type T throws if it cannot copy the initial value *val*.

By careful manual analysis, it is possible to figure out that if the latter throws, one has already allocated memory, which would be necessary to free. Yet, memory leakage is not the only problem. One might actually successfully

```
1  template<class T>
2  vector<T>::vector(size_type n, const T & val) {
3      v = cpp_malloc<T>(n);
4      space = last = v + n;
5      for (T* p=v; p != last; ++p) {
6          new (p) T(val);
7      }
8  }
```

Figure 5.3: (Simplified) source code of a naive vector implementation [18]

construct a few copies before a copy construction fails. A correct program also needs to destruct these objects.

Our analysis will produce a trace describing that the strong guarantee is invalidated because it is possible to perform a *cpp_malloc* that is followed by at least one copy construction before throwing an exiting exception.

The complete example, including template instantiations, the vector class, and needed support procedures consists of together 50 lines of C++ source code. The resulting XML-file that the ELSA parser outputs consists of a just over 4000 XML-nodes and is around 400kB in size.

Generating the control-flow graph for the constructor instantiated with $T=\text{bomb}$, we get a very large graph, which is presented in Figure 5.4 in a filtered manner, to make it more presentable. The filtering is initiated by supplying a flag to the CFGGEN application, which removes any nodes that cannot possibly lead to any modification of state or any exceptions being raised.

In Figure 5.4, we see that the control-flow graph describes the control flow of Figure 5.3 properly. Amongst other things, it has identified the two exceptional exits previously mentioned. One of them is the copy construction of the initial value and the other is the procedure call to *cpp_malloc*. The first exceptional exit is not directly visible in the source code, but is represented by a new-expression. Since the control-flow graph reflects the source code, one of the exceptional exits therefore is from the new-expression, and the other one, as expected, from *cpp_malloc*.

The control-flow graph is then annotated according to the specification in Section 4.5. The resulting graph is depicted in Figure 5.5. Since the graph produced by BANGGRAPH only includes annotated nodes, and only exit nodes are annotated, this graph is pruned even more. We can see that the exceptional exits have been annotated with t , and that we have five different m annotations. Since we are looking at a constructor, and the assignments

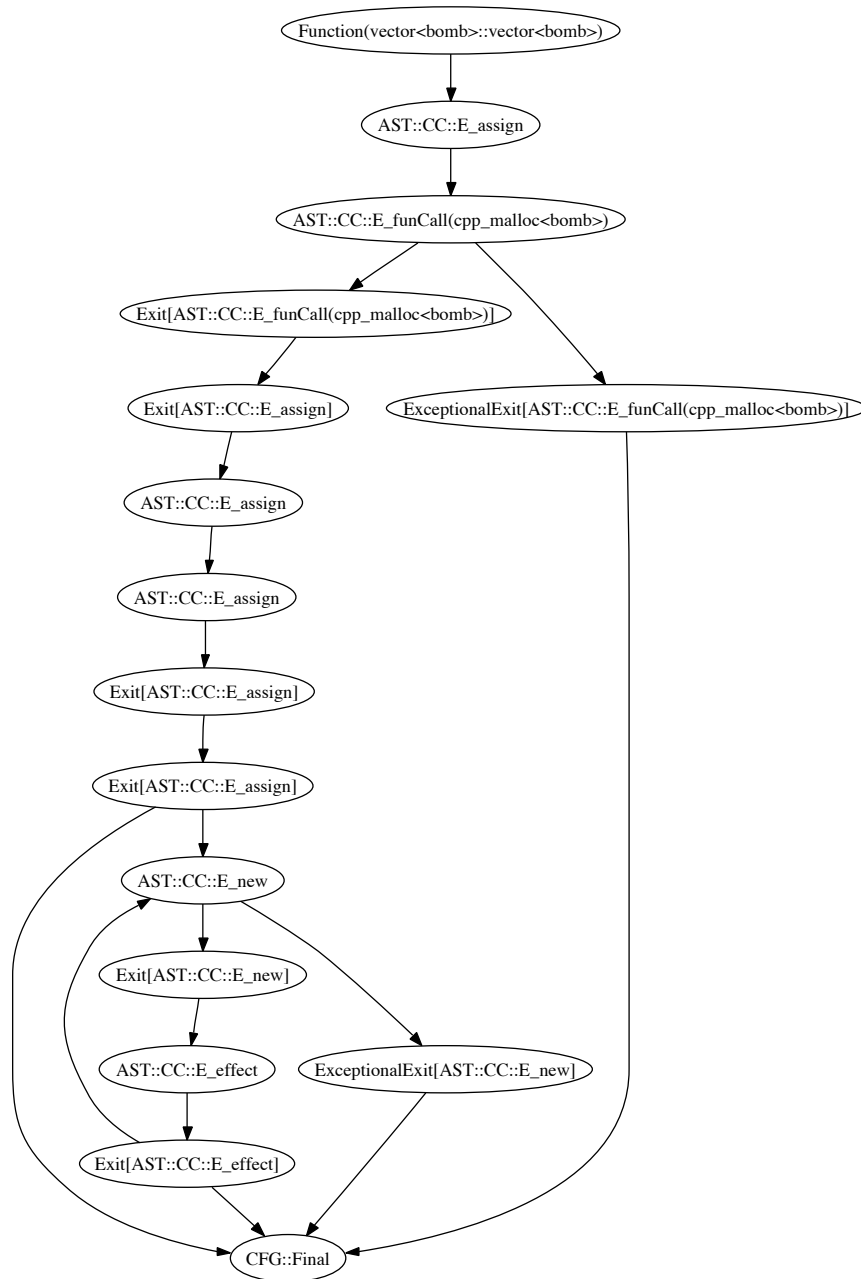


Figure 5.4: Graph generated by CFGGEN for the naive approach

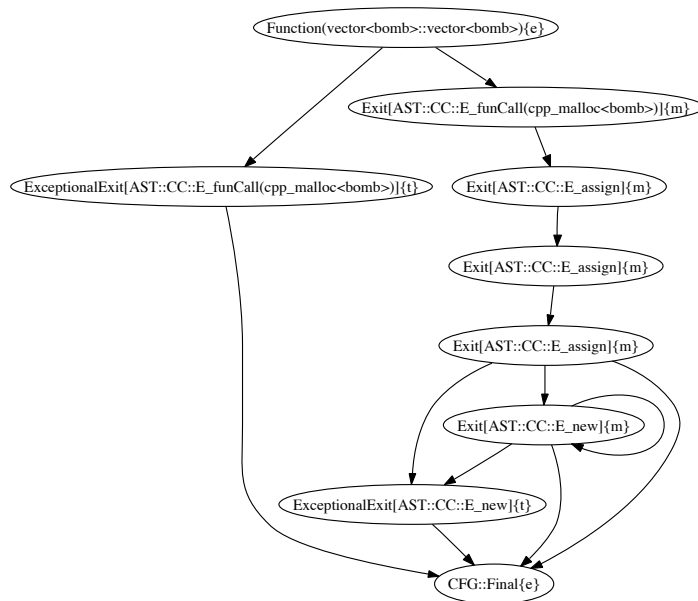


Figure 5.5: Graph generated by BANGGRAPH for the naive approach

are to instance-variables, those assignments ideally should not produce m annotations, which would leave only the one annotation for the `cpp_malloc` invocation and one for the copy construction in the new-expression. However, support for the special semantics of constructors has not yet been implemented. If we remove the back-edge in Figure 5.5 and thereby consider all successful copy constructions as a single operation, we can identify five distinct execution paths through the procedure:

- Failing early with `cpp_malloc`.
- Succeeding by skipping the loop, avoiding both normal and exceptional exits of the new-expression.
- Succeeding by completing the copy construction of all entries, skipping the exceptional exit of the new-expression.
- Failing at the first copy construction, going directly to the exceptional exit of the new-expression.
- Failing at a subsequent copy construction, entering first the normal exit of the new-expression, continuing to the exceptional exit.

Running BANGSAFE on the example produces the trace in Figure 5.6. This trace has not been pruned, in order to show the volume of information

```

> bangsafe vector-naive.dump

Annotation for 'bomb' is e
Annotation for 'bad_alloc' is e
Annotation for 'bomb' is u
Annotation for 'vector' is e
Annotation for 'operator=' is e
Annotation for '~bomb' is e
Annotation for 'operator new' is u
Annotation for 'cpp_malloc<int>' is u
Annotation for 'vector<int>::vector<int>' is u
Annotation for 'bad_alloc' is e
Annotation for 'operator delete' is e
Annotation for 'cpp_free<>' is e
Annotation for '~vector' is e
Annotation for 'cpp_malloc<>' is u
Annotation for 'operator new' is e
Annotation for '~vector' is e
Annotation for 'operator=' is e
Annotation for 'vector<T>::vector<>' is u
Annotation for 'cpp_malloc<bomb>' is u
Annotation for 'vector<bomb>::vector<bomb>' is s
  Detected invalidating path:
  > m:3:AST::CC::E_funCall(cpp_malloc<bomb>)
  > m:3:AST::CC::E_assign
  > m:4:AST::CC::E_assign
  > m:4:AST::CC::E_assign
  > m:6:AST::CC::E_new
  > t:6:AST::CC::E_new(AST::CC::E_throw)
Annotation for '~bad_alloc' is e
Annotation for 'operator=' is e
Annotation for 'vector' is e
Annotation for 'operator=' is e

```

Figure 5.6: Output produced by BANGSAFE for the naive approach

within the logfile. Look for *vector<bomb>::vector<bomb>* within the trace to find the interesting portion of the output. You can see in the invalidating path that the analysis identifies the worst case where both *cpp_malloc* and one successful copy construction have been applied. The copy construction is, as explained earlier, embedded within the new-expression.

5.2 The naive approach, take two

Now suppose that we instantiate the same constructor with *int* instead of *bomb*. Then the example provides the strong guarantee because the copy construction of *int* cannot throw. The analysis can confirm that. The trace in Figure 5.6 shows that the annotation for *vector<int>::vector<int>* is *u*, which indicates that no state modification took place before the exception was thrown; thus the strong guarantee holds. This result is correct, since only

```
1  template <class T>
2  vector<T>::vector(size_type n, const T& val)
3  {
4      v = cpp_malloc<T>(n);
5      try {
6          uninitialized_fill(v, v+n, val);
7          space = last = v+n;
8      }
9      catch (...) {
10         cpp_free(v);
11         throw;
12     }
13 }
```

Figure 5.7: Simplified source code of corrected vector implementation [18]

allocation can fail, but all allocation takes place before any state modification.

The output of BANGGRAPH for the instantiation with *int* is omitted, since it looks identical to Figure 5.5, except that the node labeled as an exceptional exit from the new-expression no longer exists.

5.3 The messy approach

Stroustrup solves the two exception safety issues within the naive approach in a straightforward way, by using the standard routine *uninitialized_fill* to construct the objects and wrapping the whole block of code in a try-catch block to ensure that any allocated memory is freed on failure. Figure 5.7 shows the implementation after these corrections. Since the corrected implementation does not look pretty at all, we refer to it as the messy example.

The complete example, including all necessary support procedures, consists of 70 lines of C++ source code. The resulting XML-file that the ELSA parser produces consists of around 5000 XML-nodes and is almost 500kB in size.

Generating the control-flow graph the same way as for the naive example, we obtain the slightly more complicated graph in Figure 5.8, since the source code of the constructor is now more complicated. Worth mentioning in this graph is that the exit node for the handler has no predecessor. The reason for this is that the throw-expression will terminate without ever connecting to the normal control flow.

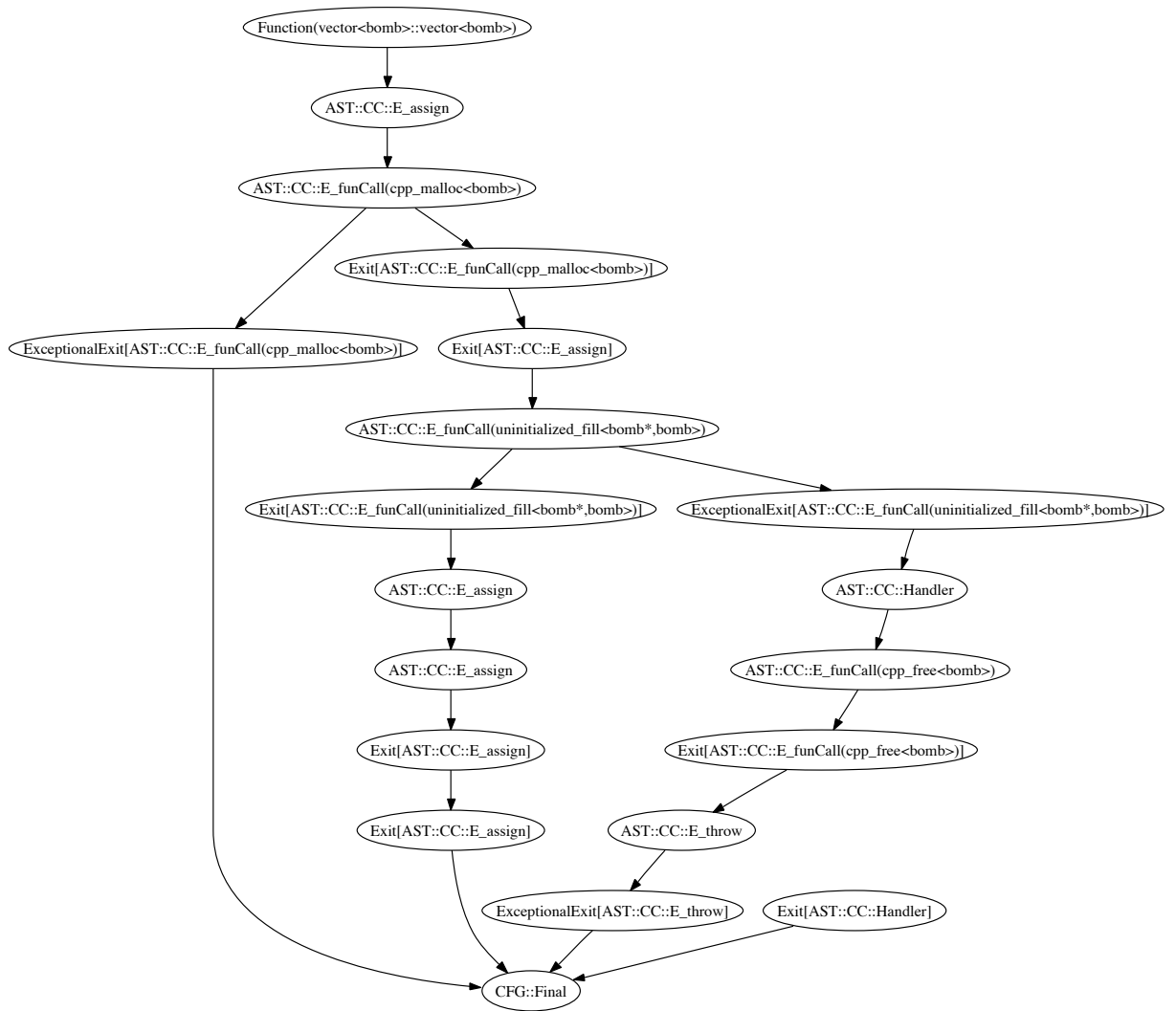


Figure 5.8: Graph generated by CFGGEN for the messy approach

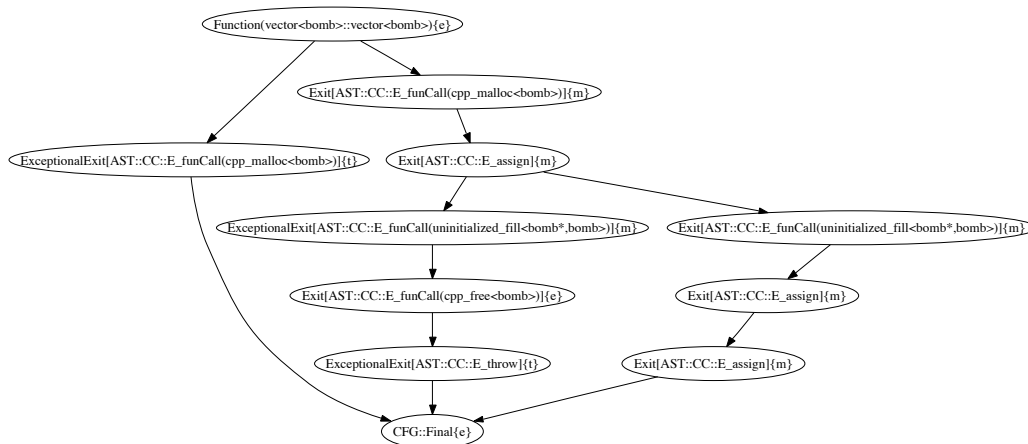


Figure 5.9: Graph generated by BANGGRAPH for the messy approach

Figure 5.9 adds the annotations. As we can clearly see, the three different paths exhibit different exceptional behavior. One of them is the quick failure if the call to *cpp_malloc* fails, and another is a sequence of just m annotations. The third path is definitely the most interesting one. It represents the case when the allocation succeeds but the call to *uninitialized_fill* fails. We can identify a sequence of m followed by t for this path, and thus we know that the algorithm will provide an s annotation for the whole procedure, which indicates a violation of the strong guarantee. In reality this path constitutes no problem—it represents a false positive: the state is restored because the memory allocated is freed, *uninitialized_fill* actually provides the strong guarantee and the assignment is to an instance variable in the constructor, thus could never have modified the program state.

The overly conservative treatment of assignments could be dealt with by adding special support for constructor semantics. More problematic is that the algorithm does not realize that a call to *cpp_malloc* followed by *cpp_free* is in fact valid. The problem here is that *cpp_malloc* temporarily destroys the program state and then *cpp_free* restores it. Using our abstraction, this restoration cannot be detected.

The complete output from the BANGSAFE run in Figure 5.10 looks very similar to the naive example. The only difference is the rewritten constructor and the addition of *uninitialized_fill*. Once more, the entry corresponding to *vector<bomb>::vector<bomb>* is the interesting one.

As briefly mentioned above, our analysis generates a false positive also for the *uninitialized_fill* procedure; the reason once again being that we cannot detect that an operation is reverted. The procedure *uninitialized_fill* encapsulates the functionality of constructing objects as copies from an initial

```

> bangsafe vector-messy.dump

Annotation for 'bad_alloc' is e
Annotation for 'bomb' is u
Annotation for 'uninitialized_fill<bomb*,bomb>' is s
  Detected invalidating path:
    > m:6:AST::CC::E_new
    > t:10:AST::CC::E_throw()
Annotation for 'bomb' is e
Annotation for 'uninitialized_fill<>' is e
Annotation for 'operator=' is e
Annotation for '~bomb' is e
Annotation for 'operator new' is u
Annotation for 'cpp_malloc<int>' is u
Annotation for 'vector' is e
Annotation for 'bad_alloc' is e
Annotation for '~vector' is e
Annotation for 'cpp_malloc<>' is u
Annotation for 'operator new' is e
Annotation for '~vector' is e
Annotation for 'operator delete' is e
Annotation for 'cpp_free<int>' is e
Annotation for 'uninitialized_fill<int*,int>' is e
Annotation for 'vector<int>::vector<int>' is s
  Detected invalidating path:
    > m:4:AST::CC::E_funCall(cpp_malloc<int>)
    > m:4:AST::CC::E_assign
    > t:11:AST::CC::E_throw()
Annotation for 'cpp_free<>' is e
Annotation for 'operator=' is e
Annotation for 'vector<T>::vector<>' is s
  Detected invalidating path:
    > m:4:AST::CC::E_funCall(cpp_malloc<>)
    > m:4:AST::CC::E_assign
    > t:11:AST::CC::E_throw()
Annotation for 'cpp_free<bomb>' is e
Annotation for 'cpp_malloc<bomb>' is u
Annotation for 'vector<bomb>::vector<bomb>' is s
  Detected invalidating path:
    > m:4:AST::CC::E_funCall(cpp_malloc<bomb>)
    > m:4:AST::CC::E_assign
    > m:6:AST::CC::E_funCall(uninitialized_fill<bomb*,bomb>)
    > t:11:AST::CC::E_throw()
Annotation for '~bad_alloc' is e
Annotation for 'operator=' is e
Annotation for 'operator=' is e
Annotation for 'vector' is e

```

Figure 5.10: Output produced by BANGSAFE for the messy approach

value, and in case of error calls the destructor for all constructed objects. Figure 5.10 should show that *uninitialized_fill*<*bomb**,*bomb*> performs a sequence of operations including copy construction, destruction and throw. The destruction is not present, however, since the ELSA parser is not able to correctly identify the C++ syntax.

Chapter 6

Conclusion

We have designed and implemented a static analysis for the strong exception-safety guarantee. Section 6.1 contains an evaluation of our approach mixed with thoughts on improvements and further work. Section 6.2 describes how to get access to the tool.

6.1 Evaluation and future work

Since we have not used our tool for any extensive real-world benchmarking, the evaluation in this section is based on a theoretical assessment of the functionality that would be needed to obtain an industrial-strength tool for analyzing the exception safety classification of a procedure.

We think the analysis could be improved mainly by supporting additional C++ language features and by improving the precision of the results. These topics are discussed in Section 6.1.1 and Section 6.1.2 respectively. Section 6.1.3 discusses improvements to the BANGSAFE tool itself.

6.1.1 Coverage

In this section we discuss features of the C++ language that we currently do not address, how they could be implemented and what needs to be changed for that.

Our analysis currently only has support for instantiated templates. We support instantiated templates because an instantiated procedure is very similar to a normal procedure. We would, however, like to support uninstantiated templates, because the exception safety classification is most useful there—the reason for the classification approach to exception safety was that the C++ standard library makes substantial use of templates.

Supporting uninstantiated templates is not easy, mainly because a lot of expressions can implicitly lead to a procedure call in C++. Examples include automatic type conversions, operator overloading, and return statements. In the instantiated approach, we can rely on the parser to lower the abstract syntax tree to explicitly include the procedure calls, since the parser must do this to correctly type-check the program. For uninstantiated templates, however, only limited type-checking takes place.

In accordance with the classification idea, the most natural thing for our analysis would be to pessimise. Pessimising means that the analysis would assume that any syntactic construct that could possibly lead to a procedure invocation actually does so, and that the called procedure would be classified as *s*. We would then no longer be able to determine what type of exceptions the procedure might throw.

The introduction of support for uninstantiated templates also raises a program interface issue. In addition to the normal output of the maximum guarantee achieved, it would also be of interest to emit which requirements a template argument must meet for the procedure to fulfill the strong guarantee. With the current trace, the emitted information is a sequence of expressions from which it would be quite complicated to manually extract the requirements on the template arguments. This extraction would be separate from the core algorithm, similar to the current synthesis algorithm. For procedures that take several template parameters, the presentation of these requirements would probably be quite difficult, because the requirements of one type would depend on the other types.

Another C++ language feature of interest is classes, including assignments to fields, constructor semantics, and virtual functions. Especially from the third example in Chapter 5, it is clear that constructors have special semantics that need special attention to produce precise results. Further work is required to determine how the introduction of classes affect the analysis. It is clear, however, that the overall effort is substantial, since there exists separate analyses to deal solely with, for instance, virtual functions invocations.

Automatic destructors is another language feature that is frequently used in idioms and real-world programs to obtain the strong guarantee. Currently, however, the control-flow graph generation does not include implicit destructor invocations. Automatic destructors can be used to ensure that the effects of an operation are automatically undone when an exception occurs. Since the analysis cannot determine if one operation undoes another, the analysis will unfortunately report these cases as false positives. These false positives should be easy to identify in the trace, but it might be worth investigating if they could be dealt with automatically. Implementing support for automatic destructors also implies that an exiting exception is not necessarily

the last expression. The C++ standard specifies that the function *terminate* will be called if an exception is thrown inside an automatic destructor. In its present form, the analysis would not detect this behavior, even though it has all required information. A natural extension of the abstract representation would be to add a class of control-flow graphs that encapsulates this double exception throw. Sequencing two *t* annotations would yield this classification, and there would be no way to escape this class.

Another deficiency of the control-flow graph generation is how it deals with indeterministic execution-order, for example, in procedure invocations, where the order of argument evaluation is undefined. In the current implementation, a back-edge is added from the last argument to the first. The back-edge ensures that all arguments depend on all other arguments, but unfortunately makes the argument-expression depend on itself as well, which is clearly not the case. This self-dependency could introduce false positives if another procedure call was part of the arguments. The self-dependency could be avoided by adding the argument list as two separate directed paths in the control-flow graph, one of the paths connected from first to last argument, and the other in the reverse ordering. This would ensure that every argument node is dependent on all other arguments, but would in return require more duplication in the graph structure.

The algorithm currently cannot handle throw-specifications, which is a way in C++ to annotate a procedure with the exceptions it may throw. Support for exception specifications would partly allow the analysis to deal with procedures without having their source code available. However, we can only answer one of the three questions in Figure 1.2.

An exception that is thrown by a procedure and not part of the procedure's throw specification will result in a call to *unexpected*, which in most cases lead to program termination. Adding support for exception specifications would allow us to compare specified exceptions with computed exceptions and alert the user on mismatch. For such analysis, the part of the architecture that constructs the control-flow graph would suffice.

Recursive procedure calls are not supported, since the algorithm requires that the result of all procedures called are available when analyzing the current procedure. The issue with recursive procedure calls lies within control-flow graph construction and occurs because somewhere in the cycle of procedure calls we have to make assumptions about the exception behavior of one of the procedures involved in the recursion. For every exception that can be raised while recursing, we have to guess whether the procedures involved in the recursion will pass this exception on to their callers or not. If our assumption about the involved procedures behavior is false, we will either not include exceptions that might in reality be thrown, or we will complicate the

control flow within the involved procedures, which results in longer execution time and decreased precision for the analysis. Further work is required to figure out how to make the right assumptions in the case of recursive procedures. Since the number of possibly thrown exceptions is bounded, it should also be possible to reiterate over the involved procedures to improve precision.

Our analysis currently does not support goto-expressions. A goto would introduce an arbitrary edge into the control-flow graph. The syntactic constructs in our current language only include branches and back-edges in the control-flow graph, whereas a goto could introduce forward- and cross-edges that would complicate the graph structure. We would need to investigate the impact these complications have on the analysis before such support could be added.

6.1.2 Precision

The precision of the analysis was discussed in detail in Section 4.6, so that we now can focus on the possible improvements. The ideas for improving the precision mostly involve integrating our analysis with other analyses, such as points-to analyses.

Since we want the analysis to work properly for the strong guarantee without having to encode the program invariants, the precision will always be bounded by our assumption to detect invariant invalidation through explicit state modification. We have chosen state modification since this is easy to detect and a necessary condition for any invariant invalidation, given that the full source code is provided.

The semantics of C++ allows us to determine that some of these modifications do not imply invariant invalidation, for example when the effects are just local and will not escape the procedure boundaries.

Our analysis tries to detect that only local variables are modified, but could be improved for the intraprocedural case, and even more so for the interprocedural case. In the intraprocedural case, we are able to identify an assignment as local, only if it is directly to a local non-reference variable. If the assignment is to a dereferenced pointer, or a reference, we cannot say anything. We want to investigate the benefits of adding support for a points-to analysis. Intuitively it feels as if the points-to analysis would be most efficient in combination with an improved interprocedural analysis. Currently, we do not keep track of what has been modified inside a procedure. If we could determine that a procedure modifies parameter x , we could then in the interprocedural case check whether the argument supplied to the callee was local to the caller or not.

Another major reason for loss of precision is the fact that we consider all paths through the control-flow graph of a procedure feasible. In reality some paths are in fact infeasible, and should be ignored. This problem is common to many analyses, and integration with analyses that perform, for example, dead code elimination would probably help.

6.1.3 Tool

The most interesting aspect of extending the BANGSAFE tool would be to take advantage of the compositional behavior of the underlying algorithm. This would allow the user to supply a set of inputs mapping procedures against the appropriate guarantee and this would make BANGSAFE into a fragment program analysis. Another important improvement would be to make BANGSAFE more interactive. Especially for the messy example in Section 5.3, it would have been nice if the user would have to opportunity to correct the classification of *uninitialized_fill* before this result was used to compute further classifications.

6.2 Availability

Instructions for downloading and installing BANGSAFE can be found here:
<http://sms.cs.chalmers.se/bangsafe>

Bibliography

- [1] David Abrahams. Exception-safety in generic components. In Mehdi Jazayeri, Rüdiger Loos, and David R. Musser, editors, *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 69–79. Springer, 1998.
- [2] David Abrahams and Greg Colvin. Making the C++standard library exception safe. Technical Report N1086 = 97-0048R1, C++Standards Committee, 1997.
- [3] David Abrahams and Greg Colvin. Making the C++standard library more exception safe. Technical Report N1114 = 97-0076, C++Standards Committee, 1997.
- [4] Andrei Alexandrescu and David B. Held. Smart pointers reloaded (ii): Exception safety analysis. *C/C++Users Journal*, 21(12):40–44, December 2003.
- [5] Flaviu Cristian. A recovery mechanism for modular software. In *ICSE '79: Proceedings of the 4th International Conference on Software Engineering*, pages 42–50.A, Piscataway, NJ, USA, 1979. IEEE Press.
- [6] Flaviu Cristian. Exception handling and software fault tolerance. In *FTCS '80: Proceedings of the 10th IEEE International Symposium on Fault Tolerant Computing*, pages 97–103. IEEE Computer Society Press, 1980.
- [7] Beman Dawes. Boost library requirements and guidelines. http://www.boost.org/more/lib_guide.htm, November 2003.
- [8] Christophe de Dinechin. C++exception handling. *IEEE Concurrency*, 8(4):72–79, 2000.

-
- [9] John B. Goodenough. Structured exception handling. In *POPL '75: Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 204–224, New York, NY, USA, 1975. ACM Press.
- [10] Udo Hebisch and Hanns J. Weinert. *Semirings: Algebraic Theory & Applications in Computer Science*. World Scientific, January 1999.
- [11] Scott McPeak. Elkhound and Elsa. <http://www.cs.berkeley.edu/~smcpeak/elkhound/>, August 2005.
- [12] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In Evelyn Duesterwald, editor, *CC*, volume 2985 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 2004.
- [13] Brian Randell. System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*, pages 437–449, 1975.
- [14] Martin P. Robillard and Gail C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering Methodology*, 12(2):191–221, 2003.
- [15] Barbara G. Ryder and Mary Lou Soffa. Influences on the design of exception handling ACM SIGSOFT project on the impact of software engineering research on programming language design. *SIGSOFT Software Engineering Notes*, 28(4):29–35, 2003.
- [16] Carl F. Schaefer and Gary N. Bundy. Static analysis of exception handling in Ada. *Software - Practice & Experience*, 23(10):1157–1174, 1993.
- [17] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley Professional, March 1994.
- [18] Bjarne Stroustrup. *The C++ Programming Language*, chapter Appendix E. Addison-Wesley Professional, special edition, February 2000.
- [19] Herb Sutter. ACID Programming. `comp.lang.c++.moderated`, September 1999. Guru of the Week 61: <http://www.gotw.ca/gotw/061.htm>.
- [20] Herb Sutter. *Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems and Solutions*. AW C++ in Depth Series. Addison Wesley, August 2004.