A Specification Logic for Exceptions and Beyond

Cristian Gherghina and Cristina David

Department of Computer Science, National University of Singapore

Abstract. Exception handling is an important language feature for building more robust software programs. It is primarily concerned with capturing abnormal events, with the help of catch handlers for supporting recovery actions. In this paper, we advocate for a specification logic that can uniformly handle exceptions, program errors and other kinds of control flows. Our logic treats exceptions as possible outcomes that could be later remedied, while errors are conditions that should be avoided by user programs. This distinction is supported through a uniform mechanism that captures static control flows (such as normal execution) and dynamic control flows (such as exceptions) within a single formalism. Following Stroustrup's definition [15, 9], our verification technique could ensure exception safety in terms of four guarantees of increasing quality, namely no-leak guarantee, basic guarantee, strong guarantee and no-throw guarantee.

1 Introduction

Exception handling is considered to be an important but yet controversial feature for many modern programming languages. It is important since software robustness is highly dependent on the presence of good exception handling codes. Exception failures can account for up to 2/3 of system crashes and 50% of system security vulnerabilities [10]. It is controversial since its dynamic semantics is often considered too complex to follow by both programmers and software tools.

Goodenough provides in [5] a possible classification of exceptions according to their usage. More specifically, they can be used:

- to permit dealing with an operation's failure as either domain or range failure. Domain failure occurs when an operation finds that some input assertion is not satisfied, while range failure occurs when the operation finds that its output assertion cannot be satisfied.
- to monitor an operation, e.g. to measure computational progress or to provide additional information and guidance should certain conditions arise.

In the context of Spec#, the authors of [7] use the terms client failures and provider failures for the domain and range failures, respectively. Client failures correspond to parameter validation, whereas provider failures occur when a procedure cannot perform the task it is supposed to. Moreover, provider failures are divided into admissible failures and observed program errors. To support admissible failures, Spec# provides checked exceptions and throws sets. In contrast, an observed program error occurs if the failure is due to an intrinsic error in the program (for e.g. an array bounds error) or a global failure that is not tied to a particular procedure (for e.g. an out-of-memory error).

Such failures are signaled by Spec# with the use of unchecked exceptions, which do not need to be listed in the procedure's throws set. Likewise for Java, its type system has been used to track a class of admissible failures through checked exceptions.

However, omitting the tracking of unchecked exceptions is a serious shortcoming, since it provides a backdoor for some programmers to deal exclusively with unchecked exceptions. This shortcut allows programmers to write code without specifying or catching any exceptions. Although it may seem convenient to the programmer, it sidesteps the intent of the exception handling mechanisms and makes it more difficult for others to use the code. A fundamental contradiction is that, while unchecked exceptions are regarded as program errors that are not meant to be caught by handlers, the runtime system continues to support the handling of both checked and unchecked exceptions.

The aim of the current work is ensure a higher level of exception safety, as defined by Stroustrup [15] and extended by Li and co-authors [9], which takes into consideration both checked and unchecked exceptions. According to Stroustrup, an operation on an object is said to be exception safe if it leaves the object in a valid state when it is terminated by throwing an exception. Based on this definition, exception safety can be classified into four guarantees of increasing quality:

- **No-leak guarantee**: For achieving this level of exception safety, an operation that throws an exception must leave its operands in well-defined states, and must ensure that every resource that has been acquired is released. For example, all memory allocated must be either deallocated or owned by some object whenever an exception is thrown.
- **Basic guarantee**: In addition to the no-leak guarantee, the basic invariants of classes are maintained, regardless of the presence of exceptions.
- **Strong guarantee**: In addition to providing the basic guarantee, the operation either succeeds, or has no effects when an exception occurs.
- **No-throw guarantee**: In addition to providing the basic guarantee, the operation is guaranteed not to throw an exception.

We propose a methodology for program verification that can guarantee higher levels of exception safety. Our approach can deal with all kinds of control flows, including both checked and unchecked exceptions, thus avoiding the unsafe practice of using the latter to circumvent exception handling. Another aspect worth mentioning is that some of the aforementioned exception safety guarantees might be expensive. For instance, strong guarantee might incur the high cost of roll-back operations as it pushes all the recovery mechanism into the callee. However, such recovery may also be performed by the caller, or there might be cases when it is not even required. Consequently, in Section 4 we improve on the definition of strong guarantee for exception safety.

Moreover, verifying a strong guarantee is generally more expensive than the verification of a weaker guarantee, as it is expected to generate more complex proof obligations (details can be found in Section 7). Hence, according to the user's intention, our system can be tuned to enforce different levels of exception safety guarantees.

1.1 Our contributions

The main contributions of our paper are highlighted below:

– We introduce a specification logic that captures the states for both normal and exceptional executions. Our design is guided by a novel unification of both static control flows (such as break and return), and dynamic control flows (such as exceptions and errors).
– We ensure exception safety in terms of the guarantees introduced in [15], and extended in [9]. Additionally, we improve the strong guarantee for exception safety. To support a tradeoff between precision and cost of verification, our verification system is flexible in enforcing different levels of exception safety.
– We have implemented a prototype verification system with the above features and validated it with a suite of exception-handling examples.

2 Related Works

Exceptions are undoubtedly important parts of programming language systems that should be adequately handled during program verification. Consequently, in recent years, there have been several research works that tackle the problem of exception handling verification and analysis.

A traditional approach is based on type systems. However, it is tedious and possibly imprecise to simply declare the set of exceptions that may escape from each method. A better scheme is for the inference of uncaught exceptions. For example, [6] employs two analyses of different granularity, at the expression and method levels, to collect constraints over the set of exception types handled by each Java program. By solving the constraint system using techniques from [4], they can obtain a fairly precise set of uncaught exceptions. For exceptions that are being ignored by type system, such as unchecked exceptions for Java and C#, there is a total loss of static information for them. Moreover, conditions leading to exceptions are often quite subtle and cannot be conveniently tracked in a type system.

To overcome the above shortcomings, this paper proposes a more expressive specification logic, which complements the type system through selective tracking on types that are linked to control flows. A more limited idea towards verification of exception handling programs is based on model checking [9]. A good thing is that it does not require any specification for the verification of exception reliability (on the absence of uncaught exceptions and redundant handlers), but annotations are required for the verification of the no-leak guarantee. However, this approach does not presently handle higher levels of exception safety, beyond the no-leak guarantee.

A recent approach towards exception handling in a higher-order setting, is taken in [2]. Exceptions are represented as sums and exception handlers are assigned polymorphic, extensible row types. Furthermore, following a translation to an internal language, each exception handler is viewed as an alternative continuation whose domain is the sum of all exceptions that could arise at a given program point. Though CPS can be used to uniformly handle the control flows, it increases the complexity of the verification process as continuations are first class values.

Spec# also has a specification mechanism for exceptions. While its verification system is meant to analyse C# programs, exceptional specifications are currently useable for only the runtime checking module. The current Spec# prototype for static verification would unsoundly approximate each exception as false, denoting an unreachable program state [8].

Another impressive verification system, based on the Java language, is known as the KeY approach [1]. This system makes use of a modal logic, known as dynamic logic. Like Hoare logic, dynamic logic supports the computation of strongest postcondition for program codes. However, the mechanism in which this is done is quite different since program fragments may be embedded within the dynamic logic itself. This approach is more complex since rewriting rules would have to be specified for each programming construct that is allowed in the dynamic logic. For example, to support exception handling, a set of rewriting rules (that are meaning-preserving) would have to be formulated to deal with raise, break and try-catch-finally constructs, in addition to rewriting rules for block and conditionals and their possible combinations. The KeY approach is meant to be a semi-automated verification system that occasionally prompts the user for choice of rewriting rules to apply. In contrast, our approach is meant to be fully-automated verification system, once pre/post specifications have been designed.

3 Source and Specification Languages

As input language for our system, we consider a Java-like language which we call `SrcLang`. Although we make use of the class hierarchy to define a subtyping relation for exception objects, the treatment of the other object-oriented features, such as instance methods and method overriding, is outside the scope of the current paper. Our language permits only static methods and single inheritance. We have also opted for a monomorphically typed language. These simplifications are orthogonal to our goal of providing a logic for specifying exceptions. We present the full syntax for the input source language in Fig 1. Take note that $\overrightarrow{e}$ denotes $e_1, \ldots, e_n$.

$P ::= \overrightarrow{D}\ ;\ \overrightarrow{V}\ ;\ \overrightarrow{M}$	*program*
$V ::=$ `pred root::`$pname\langle\overrightarrow{v}\rangle \equiv \Phi$ `inv` π	*pred declaration*
$D ::= class\ c_1\ extends\ c_2\{\overrightarrow{t\ f}\}$	*data declaration*
$t ::= c \mid p$	*user or prim. type*
$M ::= t\ m\ (\overrightarrow{[ref]\ t\ v})$ `requires` Φ_{pr} `ensures` $\Phi_{po}\ \{e\}$	*method declaration*
$w ::= v \mid v.f$	*variable or field*
$e ::= v \mid k \mid$ `new` $c \mid v.f \mid m\ (\overrightarrow{v}) \mid \{t\ v;\ e\} \mid w{:=}e \mid e_1;\ e_2$	
$\mid$ `if` e `then` e_1 `else` e_2	
$\mid (t)\ e$	*casting*
$\mid$ `raise` e	*throw exception*
$\mid$ `break` $[L] \mid$ `return` e	*break and return*
$\mid$ `continue` $[L]$	*loop continue*
$\mid L : e$	*labelled expression*
$\mid e_1$ `finally` e_2	*finally*
$\mid$ `try` e `catch` $(c_1\ v_1)\ e_1[$`catch` $(c_i\ v_i)\ e_i]_{\mathtt{i=2}}^{\mathtt{n}}$	*multiple catch handlers*
$\mid$ `do` e_1 `while` e_2 `requires` Φ_{pr} `ensures` Φ_{po}	*loop*

Fig. 1. Source Language : `SrcLang`

Our language allows for functions (and loops) to be decorated with pre and post conditions that are verified by our tool. Unlike SPEC# or ESC/Java, where specifications for exceptions are captured by a special syntax for exceptional postconditions,

we aim for a unified logic that is capable of capturing all kinds of control flows. Our specification, as described in Fig 2, is based on separation logic formulas, introduced by John Reynolds [13], given in disjunctive normal form, but has been enhanced with a control flow annotation. Within a formula, each disjunct consists of a subformula κ referred to as heap part and π, a pure part that represents a heap-independent part of the formula.

$$
\begin{array}{lll}
\Delta & ::= \Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v \cdot \Delta \mid \Delta \wedge \beta & \textit{composite formulae} \\
\Phi & ::= \bigvee (\exists v^* \cdot \kappa \wedge \pi) & \textit{formulae} \\
\pi & ::= \gamma \wedge \phi \wedge \tau & \textit{pure constraints} \\
\gamma & ::= v_1 = v_2 \mid v = \texttt{null} \mid v_1 \neq v_2 \mid v \neq \texttt{null} \mid \gamma_1 \wedge \gamma_2 & \textit{pointer constraints} \\
\kappa & ::= \texttt{emp} \mid v :: a\langle \overrightarrow{v} \rangle \mid \kappa_1 * \kappa_2 & \textit{heap constraints} \\
ft & ::= \ c \mid predef\_flow & \textit{flow types} \\
\beta & ::= \ fv = ft \mid fv_1 = fv_2 & \textit{flow var constraints} \\
\tau & ::= \texttt{flow} \ = fset \qquad fset ::= \texttt{Ex}(ft) \mid ft - \{ft_1, .., ft_n\} & \textit{current flow} \\
\phi & ::= \texttt{true} \mid \texttt{false} \mid b_1 = b_2 \mid b_1 \leq b_2 \mid c < v \mid \phi_1 \wedge \phi_2 & \textit{Presburger constraints} \\
 & \quad \mid \phi_1 \vee \phi_2 \mid \neg \phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi & \\
b & ::= k \mid v \mid k \times b \mid b_1 + b_2 \mid -b \mid max(b_1, b_2) \mid min(b_1, b_2) &
\end{array}
$$

Fig. 2. Specification Language

The heap part describes the heap footprint which is composed of $*$-separated heap nodes. These nodes, written as $v::a\langle \overrightarrow{v} \rangle$, are instances of either a user-defined class ($v::c\langle \overrightarrow{v} \rangle$) or a user-defined predicate ($v::pname\langle \overrightarrow{v} \rangle$), as an abstraction for a data structure. As shown in Fig.1, each declaration of a user-defined predicate consists of a root pointer, a predicate body and a pure formula that is an invariant of all its instances. The pure part does not capture any heap-based data structures as it contains pointer equalities/inequalities (γ), linear arithmetic (ϕ) and a special subformula τ for modelling the control flow.

A special variable `flow` is used to denote the control flow associated with the respective program state, captured as a disjunct. The possible values of control flow are either $Ex(ft)$ to denote an exact control flow type (not including its subclasses), or $ft - \{ft_1, .., ft_n\}$ to denote a control flow from ft but not from subclasses $ft_1, .., ft_n$. The control flow ft is organised as a subtyping tree hierarchy. With this hierarchy, we can be as precise as required for verifying different exception safety guarantees.

The flow type hierarchy incorporates all the possible control flow types ft, both the ones pertaining to user-defined exceptions and the predefined flow types, $predef\_flow$. This will be further elaborated in Sec.5.2.

Δ denotes a composite formula that is enhanced with an extra pure component β to capture the bindings for flow variables, fv, from the catch handlers. Lastly, each variable in our specification logic may be expressed in either primed form (e.g. v') or unprimed form (e.g. v). The former denotes the latest value of the corresponding variable, while the latter denotes the original value of the same variable. When used in the postcondition, they denote state changes that occur for parameters that are being passed by reference.

4 Examples with Higher Exception Safety Guarantees

We next illustrate our new specification mechanism through a few examples. Let us first consider the following class and predicate definitions.

```
class node { int val; node next}

pred root::ll⟨n⟩ ≡ root=null ∧ n=0 ∨
      ∃r·root::node⟨_, r⟩ ∗ r::ll⟨n−1⟩ inv n≥0;
```

Predicate `ll` defines a linear-linked list of length `n`. As elaborated earlier, each predicate describes a data structure, which is a collection of objects reachable from a base pointer denoted by `root` in the predicate definition. The expression after the **inv** keyword captures a pure formula that always holds for the given predicate.

Let us now consider the method `list_alloc` allocating memory for a linear-linked list to be pointed by `x`. The precondition requires `x` to be `null`, while the postcondition asserts that either no error was raised, i.e. the flow is `norm`, and the updated `x` points to a list with `n` elements, or an out of memory exception was raised, i.e. the flow is `out_of_mem_exc`, and `x` remains `null`.

Method `list_alloc` calls an auxiliary method `list_alloc_helper` which recursively allocates nodes in the list. If an out of memory exception is raised, the latter performs a rollback operation, inside a try-catch block, during which it frees all the memory that was acquired up to that point. Take note that, in the following examples, for illustration purposes, we provide an alternative set of library methods with explicit memory deallocation (via method `list_dealloc`) that coexists with our Java like language.

```
void list_alloc(int n, ref node x)
requires x=null ∧ n≥0
ensures (x′::ll⟨n⟩ ∧ flow=norm) ∨ (x′=null ∧ flow=out_of_mem_exc);
{list_alloc_helper(n, 0, x);}

void list_alloc_helper(int n, int i, ref node x)
requires x::ll⟨i⟩ ∧ n≥i ∧ i≥0
ensures (x′::ll⟨n⟩ ∧ flow=norm) ∨ (x′=null ∧ flow=out_of_mem_exc);
{if(n>i){
   try{x = new node(0, x);}
   catch(out_of_mem_exc exc){
      list_dealloc(i, x);
      raise (new out_of_mem_exc());}
   list_alloc_helper(n, i+1, x);}
}
```

According to [15], method `list_alloc` is said to ensure a strong guarantee on exception safety, as it either succeeds in allocating all the required memory cells, $x'::ll\langle n\rangle \wedge$ `flow=norm`, or it has no effect, $x'=null \wedge$ `flow=out_of_mem_exc`. However, this rollback operation can be expensive if we have been building a long list. Moreover, there

can be cases when such rollbacks are not needed. For instance, in the context of the aforementioned method, a caller might actually accept a smaller amount of the allocated memory.

We propose to improve Stroustrup's definition on strong guarantee as follows: An operation is considered to provide a strong guarantee for exception safety if, in addition to providing basic guarantees, it either succeeds, or its effect is precisely known to the caller. Given this new definition, the method `list_alloc_helper_prime` defined below is also said to satisfy the strong guarantee. Compared to method `list_alloc_helper`, the newly revised method has an extra pass-by-reference parameter, `no_cells`, to denote the number of memory cells that were already allocated. Through this output parameter, the caller is duly informed on the length of list that was actually allocated. This method now two possible outcomes :

- it succeeds and all the required memory cells were allocated, $\mathtt{x}'::\mathtt{ll}\langle\mathtt{n}\rangle \wedge \mathtt{no\_cells}'=\mathtt{n}$;
- an out of memory exception is thrown and `no_cells` captures the number of successfully allocated memory cells. The caller is duly informed on the amount of acquired memory through the `no_cells` parameter, and could either use it, run a recovery code to deallocate it, or mention its size and location to its own caller.

```
void list_alloc_helper_prime(int n, int i, ref node x, ref int no_cells)
requires x::ll⟨i⟩ ∧ n≥i ∧ i≥0
ensures (x′::ll⟨n⟩ ∧ no_cells′=n ∧ flow=norm) ∨
        (x′::ll⟨no_cells′⟩ ∧ flow=out_of_mem_exc);
{if(n>i){
   try{x = new node(0, x);}
   catch(out_of_mem_exc exc){
      no_cells=i;
      raise (new out_of_mem_exc());}
   list_alloc_helper(n, i+1, x);}
else no_cells=n;}
```

Next, we will illustrate how to make use of our verification mechanism for enforcing the no-throw guarantee from [15]. For this purpose, let us consider the following `swap` method which exchanges the data fields stored in two disjoint nodes.

```
void swap(node x, node y)
requires x::node⟨v1, q1⟩*y::node⟨v2, q2⟩
ensures x::node⟨v2, q1⟩*y::node⟨v1, q2⟩ ∧ flow=norm;
{  int tmp = x.val;
   x.val = y.val;
   y.val = tmp;}
```

The precondition requires the nodes pointed by `x` and `y`, respectively, to be disjoint, while the postcondition captures the fact that the values stored inside the two nodes are swapped. Additionally, the postcondition asserts that the only possible flow at the end of the method is the normal flow, `flow=norm`, i.e. no exception was raised. This specification meets the definition of no-throw guarantee given in [15]. Any presence of exception, including an exception caused by null pointer dereferencing, would require

the postcondition to explicitly capture each such exceptional flow. Conversely, any absence of exceptional flow in our logic is an affirmation for the no-throw guarantee.

5 Verification for Unified Control Flows

In this section, we present the rules for verifying programs with support for a wide range of control flows. To keep the rules simple, we shall use a core imperative language that was originally presented in [3]. First, we briefly present the key features of our core language, before proceeding to show how our verification rules handle control flows that are organised in a tree hierarchy.

5.1 Core language

For a easier specification of our verification rules, we translate the source language to a small core language [3]. This core language is detailed in Fig 3. The novelty is represented by two unified constructs for handling control flows. The first one is the output construct $\texttt{fn}\#\texttt{v}$ which can be used to provide an output with a normal flow via $\texttt{norm}\#\texttt{v}$, or a thrown exception via $\texttt{ty(v)}\#\texttt{v}$. The type of a raised exception object v is captured as its control flow.

P	$::= \overrightarrow{D}\ ;\ \overrightarrow{V}\ ;\ \overrightarrow{M}$	$program$
D	$::= \texttt{class}\ c_1\ \texttt{extends}\ c_2\ \{\overrightarrow{t\ v}\}$	$data\ declaration$
V	$::= \texttt{pred}\ \texttt{root::}pname\langle\overrightarrow{v}\rangle \equiv \Phi\ \texttt{inv}\ \pi$	$pred\ declaration$
ft	$::= c \mid predef\_flows$	$flow\ types$
M	$::= t\ m(\overrightarrow{[\texttt{ref}]\ t\ v})\ \texttt{requires}\ \Phi_{pr}\ \texttt{ensures}\ \Phi_{po}\ \{e\}$	$method\ decl$
e	$::= fn\#x$	$output\ (flow\&value)$
	$\mid v.f$	$field\ access$
	$\mid w{:=}v$	$assignment$
	$\mid m(\overrightarrow{v})$	$method\ call$
	$\mid \{t\ v;\ e\}$	$local\ var\ block$
	$\mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2$	$conditional$
	$\mid \texttt{try}\ e_1\ \texttt{catch}\ (ft[@v_1]\ v_2)\ e_2$	$catch\ handler$
fn	$::= \texttt{Ex}(ft) \mid fv \mid \texttt{ty}(v) \mid v.1$	$flow$
t	$::= c \mid p$	$user\ or\ prim.\ type$
x	$::= v \mid k \mid \texttt{new}\ c \mid (fv, v) \mid v.2$	$basic\ value$
w	$::= v \mid v.f$	$var/field$

Fig. 3. Core Language : $\texttt{Core}-\texttt{U}$

The second special construct has the form $\texttt{try}\ \texttt{e}_1\ \texttt{catch}\ ((\texttt{ft@fv})\ \texttt{v})\ \texttt{e}_2$ intended to evaluate expression $\texttt{e}_1$ that could be caught by handler $\texttt{e}_2$ if the detected control flow is a subtype of ft. This construct uses two bound variables, namely fv to capture the control flow and v to capture the output value. It is more general than the try-catch construct used in Java, since it can capture both normal flow and abnormal control flows, including break, continue and procedural return. As a simplification, the usual sequential composition $\texttt{e}_1;\texttt{e}_2$ is now a syntactic sugar for $\texttt{try}\ e_1\ \texttt{catch}\ ((\texttt{norm@\_})\_)\ e_2$, whose captured value is ignored by use of an anonymous variable .

We extend the core language to embed control flows directly as values, by allowing a pair of control flow and its value (fv, v) to be specified. With this notation, we can save each exception and its output value as an embedded pair that could be later re-thrown. Operations `v.1` and `v.2` are used to access the control flow and the value, respectively, from an embedded pair in `v`.

5.2 Control Flow Hierarchy

To support the generalized try-catch construct we provide a unified view on all control flows through the use of a tree hierarchy supporting a subtyping relation $<:$. All control flow types are subtypes of $\top$. Control flows that can be caught by catch handlers are subtypes of `c−flow`, while `abort` denotes control flows that can never be caught. The latter category includes program error, program termination and non-termination. Control flows corresponding to exceptions (both checked and unchecked) are placed in the subtree hierarchy under the `exc` class. Regarding the static (or local) control-flows, they are grouped under `local` which includes `norm` to denote normal flow, `ret` to signal a method return (covering also methods with multi-return options [14]), `brk` to denote the break out of a loop, and `cont` to denote a jump to the beginning of a loop. The use of a tree hierarchy facilitates formal reasoning, since the disjointedness property between any two flow types can be statically determined without ambiguity.

As opposed to other systems which enforce the restriction that the try-catch construct applies only to exceptional flows, our unified view on control flows will generalize the try-catch construct across the entire domain of control flow types. This domain extension permits a much more streamlined verification mechanism .

A graphical representation of the entire flow hierarchy is given in Fig.4. Each arrow $c_2 \rightarrow c_1$ denotes a subtyping relation $c1 <: c2$.

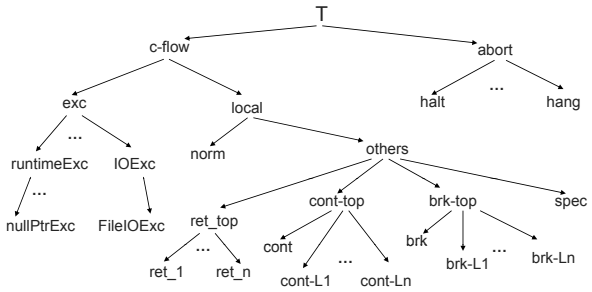

Fig. 4. A Subtype Hierarchy on Control Flows

5.3 Verification Rules

Our verification system requires pre/post conditions to be declared for each method and each loop in the input program. Loops are then transformed to tail-recursive methods where the parameters are passed by reference. With these specifications being given, we can apply modular verification to each method's body using Hoare-style triples $\vdash \{\Delta_1\}\, e\, \{\Delta_2\}$. These are forward verification rules that expect Δ_1 to be given before

computing Δ_2. Furthermore, Δ_1 is a captured program state whereby $\texttt{flow}=\top$, while Δ_2 is a disjunctive heap state capturing the entire set of control flows that may escape during the execution of e. For example, we may have $\texttt{x}'{=}\texttt{x}{+}1{\wedge}\texttt{flow}{=}\texttt{norm}{\vee}\texttt{x}'{=}\texttt{x}{\wedge}\texttt{flow}{=}\texttt{exc}$ to denote a state change for variable x for normal control flow, while an exception leaves the state of x unchanged.

In the remainder of the current section we will focus mainly on the verification of output (with flow&value) and try-catch constructs in Figure 5, since the other rules are largely conventional.

$$\frac{}{\textit{split}(\Delta, c, fv, v) = ((\exists \texttt{flow}\,.[res{\mapsto}v]\Delta \wedge \texttt{flow} <: c \wedge \texttt{flow} = fv), \Delta \wedge \neg(\texttt{flow} <: c))}\ [\textsc{fv-split}]$$

$$\frac{(fv = fset) \in \Delta}{resolve(\Delta, fv) = fset}\ [\textsc{fv-resolve-fv}] \qquad \frac{(v{=}(fv, \_)) \in \Delta \quad resolve(\Delta, fv) = fset}{resolve(\Delta, v.1) = fset}\ [\textsc{fv-resolve-pair-fst}] \qquad \frac{(v{=}(\_, w)) \in \Delta}{resolve(\Delta, v.2) = w}\ [\textsc{fv-resolve-pair-snd}]$$

$$\frac{resolve(\Delta, ft) = fset \quad \Delta_1{=}(\exists \texttt{res}.\Delta \wedge \texttt{flow} = fset) \wedge \texttt{res}{::}\texttt{c}\langle\ldots\rangle}{\vdash \{\Delta\}\, ft\#new\ c\, \{\Delta_1\}}\ [\textsc{fv-output-new}] \qquad \frac{(type(v) = c) \in \Delta}{resolve(\Delta, \texttt{ty}(v)) = c}\ [\textsc{fv-resolve-type}]$$

$$\frac{resolve(\Delta, ft) = fset \quad \Delta_1{=}(\exists \texttt{res}.\Delta \wedge \texttt{flow} = fset) \wedge \texttt{res} = (fv, v)}{\vdash \{\Delta\}\, ft\#(fv, v)\, \{\Delta_1\}}\ [\textsc{fv-output-pair}] \qquad \frac{}{resolve(\Delta, \texttt{Ex}(c)) = \texttt{Ex}(c)}\ [\textsc{fv-resolve-const}]$$

$$\frac{\vdash \{\Delta\}\, e_1\, \{\Delta_1\} \quad (\Delta_2, \Delta_3) = \textit{split}(\Delta_1, c, fv, v) \quad \vdash \{\Delta_2\}\, e_2\, \{\Delta_4\}}{\vdash \{\Delta\}\, \texttt{try}\ e_1\ \texttt{catch}\ (c@fv\ v)\ e_2\, \{\Delta_3 \vee \exists v, fv \cdot \Delta_4\}}\ [\textsc{fv-try-catch}]$$

$$\frac{resolve(\Delta, ft) = fset \quad \Delta_1{=}(\exists \texttt{res}.\Delta \wedge \texttt{flow} = fset) \wedge \texttt{res} = k}{\vdash \{\Delta\}\, ft\#k\, \{\Delta_1\}}\ [\textsc{fv-output-const}] \qquad \frac{resolve(\Delta, ft) = fset \quad \Delta_1{=}\exists \texttt{res}.(\Delta \wedge \texttt{flow} = fset) \wedge \texttt{res} = v'}{\vdash \{\Delta\}\, ft\#v\, \{\Delta_1\}}\ [\textsc{fv-output-var}]$$

$$\frac{t\ mn((\overrightarrow{t\ v}))\ \texttt{requires}\ \Phi_{pr}\ \texttt{ensures}\ \Phi_{po}\ \{..\} \in P \quad \rho{=}[u_i'/v_i] \quad \Delta \vdash \rho\Phi_{pr} * \Delta_1 \quad \Delta_2{=}(\Delta_1 * \rho\Phi_{po})}{\vdash \{\Delta\}\, mn(\overrightarrow{u})\, \{\Delta_2\}}\ [\textsc{fv-call}]$$

Fig. 5. Some Verification Rules

The output construct sets a control flow with a given value. To achieve this, we require each control flow variable to be resolved to an appropriate flow set ($fset$) value, before we can set it as the current control flow. We rely on an auxiliary set of rules, called FV–RESOLVE, to obtain the corresponding control flow set from a given program state.

The verification system makes use of the `res` variable in order to store the result of the current operation. Note that for an `output(flow&value)#` construct, the result is the value, which is bound to the `res` variable as seen in the OUTPUT rules.

Regarding the FV–TRY–CATCH rule, we first compute the post-state of expression $\texttt{e}_1$ as Δ_1. Since Δ_1 may capture a range of control flows, it has to be split into two components, Δ_2 and Δ_3, with the help of the FV–SPLIT rule. The Δ_2 component will model the program states with control flows that can be captured by the catch handler, whereas Δ_3 will model those states with control flows that escape from the catch handler. Moreover, for the case when the control flow is being caught by the handler, the control flow type is bound to fv, and its thrown value is bound to v. These bindings are kept in Δ_2 which is made available as the pre-state for $\texttt{e}_2$. As these local variables are only valid in the catch handler, we quantify them away in the resulting postcondition.

6 Correctness

In the current section we provide a description of the operational semantics for our calculus. The machine configuration is represented by $\langle e, h, s\rangle$ where e denotes the current program code, h denotes the current heap for mapping addresses to objects, and s denotes the current runtime stack for mapping variables to values. We assume sets *Loc* of locations (positive integer values), *Val* of values (either a constant, a location or a pair of control flow type and value), *Var* of variables (program variables and other meta variables), and *ObjVal* of object values stored in the heap, with $c[f_1{\mapsto}\nu_1, .., f_n{\mapsto}\nu_n]$ denoting an object value of class c where $\nu_1, .., \nu_n$ are current values (from the domain *Val* such that *Loc* $\subset$ *Val*) of the corresponding fields $f_1, .., f_n$. Let $s, h \models \Phi$ denote that stack s and heap h form a model of the constraint Φ, with h, s from the following concrete domains:

$$h \in \mathit{Heaps} =_{df} \mathit{Loc} \rightharpoonup_{fin} \mathit{ObjVal}$$
$$s \in \mathit{Stacks} =_{df} \mathit{Var} \rightarrow \mathit{Val}$$

A complete definition of the model for separation constraints can be found in [11]. For the dynamic semantics to follow through, we have introduced a couple of intermediate constructs. Their syntax is extended from the original expression syntax as shown next, where $l \in \mathit{Loc}$.

$$\begin{array}{ll} e ::= ft\#l & flow\ and\ location \\ \quad \mid \texttt{BLK}(\{\overrightarrow{v}\}, e_1) & block\ construct \end{array}$$

For the case of $\texttt{BLK}(\{\overrightarrow{v}\}, e_1)$, e_1 denotes a residual code of the current block. This new construct is used for handling try-catch constructs, method calls and local blocks. Its main purpose is to provide a lexical scope for local variables that are removed once its expression has been completely evaluated.

6.1 Small-Step Semantics

The small-step dynamic semantics is defined using the transition $\langle e, h, s\rangle{\hookrightarrow}\langle e_1, h_1, s_1\rangle$, which means that if e is evaluated in stack s, heap h, then e reduces in one step to e_1

and generates the new stack s_1 and new heap h_1. The small-step rules are given in Fig. 6. In the rules for local declaration, method call and try catch, the function $newid()$ returns a fresh identifier, while $[u'/u]$ represents the substitution of u by u'. In order to avoid dynamic binding, every variable whose binding is added to the stack is substituted by a fresh identifier. For instance, in the case of the method call, after performing the renaming of the callee's formal parameters, the mappings for the fresh identifiers are temporarily added to the stack, s. These bindings will be removed after the evaluation of the callee's body.

Regarding those constructs from the input language that may implicitly raise exceptions, such as $v.f$ or `new` c, take note that they are handled by the translation described in [3]. More specifically, our translation rules will make the raised exceptions explicit. For illustration, let us consider the translation of $v.f$:

$$v.f \Rightarrow_T \texttt{if } v{=}\texttt{null then nullPtrExc}\#v \texttt{ else } v.f$$

In the remainder of this section, we will define some notions required for proving the soundness of our verification rules. First, we provide the definition for closed configuration, which ensures that variable access does not get stuck.

$$\langle fn\#v,h,s\rangle\hookrightarrow\langle s(fn)\#s(v),h,s\rangle \qquad \langle fn\#(fv,v),h,s\rangle\hookrightarrow\langle s(fn)\#(s(fv),s(v)),h,s\rangle$$

$$\langle fn\#k,h,s\rangle\hookrightarrow\langle s(fn)\#k,h,s\rangle \qquad \langle v_1{:=}v_2,h,s\rangle\hookrightarrow\langle \texttt{norm}\#(),h,s[v_1{\mapsto}s(v_2)]\rangle$$

$$\langle v.f,h,s\rangle\hookrightarrow\langle \texttt{norm}\#(h(s(v)).f),h,s\rangle \qquad \langle v_1.f{:=}v_2,h,s\rangle\hookrightarrow\langle \texttt{norm}\#(),h[s(v_1).f{\mapsto}s(v_2)],s\rangle$$

$$\frac{l = \mathit{fresh}()}{\langle fn\#\texttt{new } c,h,s\rangle\hookrightarrow\langle s(fn)\#l,h{+}[l{\mapsto}c(\overrightarrow{\perp})],s\rangle}$$

$$\frac{u{=}newid()}{\langle\{t\ v;e\},h,s\rangle\hookrightarrow\langle\texttt{BLK}(\{u\},e[u/v]),h,s{+}[u{\mapsto}\perp]\rangle}$$

$$\frac{c_1{<:}c \quad u,fu{=}newid()}{\langle\texttt{try } c_1\#a_1 \texttt{ catch } c@fv\#v\ e_2,h,s\rangle\hookrightarrow\langle\texttt{BLK}(\{fu,u\},e_2[u/v,fu/fv]),h,s{+}[fu{\mapsto}c_1,u{\mapsto}a_1]\rangle}$$

$$\frac{\neg(c_1{<:}c)}{\langle\texttt{try } c_1\#a_1 \texttt{ catch } c@fv\#v\ e_2,h,s\rangle\hookrightarrow\langle c_1\#a_1,h,s\rangle}$$

$$\frac{\langle e,h,s\rangle\hookrightarrow\langle e_1,h_1,s_1\rangle}{\langle\texttt{try } e \texttt{ catch } c@fv\#v\ e_2,h,s\rangle\hookrightarrow\langle\texttt{try } e_1 \texttt{ catch } c@fv\#v\ e_2,h_1,s_1\rangle}$$

$$\frac{s(v){=}\texttt{true}}{\langle\texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2,h,s\rangle\hookrightarrow\langle e_1,h,s\rangle} \qquad \frac{s(v){=}\texttt{false}}{\langle\texttt{if } v \texttt{ then } e_1 \texttt{ else } e_2,h,s\rangle\hookrightarrow\langle e_2,h,s\rangle}$$

$$\frac{t_0\ mn\ (\overrightarrow{t\ u})\ \{e\} \quad \overrightarrow{u'{=}newid()}}{\langle mn(\overrightarrow{v}),h,s\rangle\hookrightarrow\langle\texttt{BLK}(\{\overrightarrow{u}'\},e[\overrightarrow{u'/u}]),h,s{+}[\overrightarrow{u'{\mapsto}s(v)}]\rangle}$$

$$\frac{\langle e,h,s\rangle\hookrightarrow\langle e_1,h_1,s_1\rangle}{\langle\texttt{BLK}(\{\overrightarrow{v}\},e),h,s\rangle\hookrightarrow\langle\texttt{BLK}(\{\overrightarrow{v}\},e_1),h_1,s_1\rangle}$$

$$\langle\texttt{BLK}(\{\overrightarrow{v}\},c\#a),h,s\rangle\hookrightarrow\langle c\#a,h,s{-}\{\overrightarrow{v}\}\rangle$$

Fig. 6. Small-Step Semantics

Definition 6.1 (Closed Configuration) *A configuration $\langle e, h, s\rangle$ is said to be* closed *if $FV(e) \subseteq dom(s)$ and $addr(s) \cup addr(h) \subseteq dom(h)$.*

Method FV collects the free variables in an expression, while $addr$ returns all addresses from the stack and heap. As their definitions are standard, we omit them from the current work. Next, we define divergent computation for small-step semantics, as follows:

Definition 6.2 (Divergence) *A configuration $\langle e, h, s\rangle$ is said to be* divergent *if its small-step transition $\langle e, h, s\rangle \hookrightarrow^* \langle e', h', s'\rangle$ never terminates with a final expression for e'. We shall represent this divergent computation using $\langle e, h, s\rangle \not\hookrightarrow^*$.*

6.2 Soundness of Verification

The soundness of our verification rules is defined with respect to the small-step dynamic semantics given in Section 6.1. Before stating the soundness theorems, we need to extract the post-state of a heap constraint by:

Definition 6.3 (Poststate) *Given a constraint Δ, Post(Δ) captures the relation between primed variables of Δ. That is :*

$$\begin{array}{l} \mathit{Post}(\Delta) \ =_{df} \ \rho\ (\exists V \cdot \Delta), \qquad \mathit{where} \\ V = \{v_1, .., v_n\} \ \mathit{denotes\ all\ unprimed\ program\ variables\ in}\ \Delta \\ \rho = [v_1/v_1', .., v_n/v_n']. \end{array}$$

Theorem 6.1 (Preservation) *Consider a closed configuration $\langle e, h, s\rangle$. If*

$$\vdash \{\Delta\}\ e\ \{\Delta_2\} \quad \mathit{and} \quad s, h \models \mathit{Post}(\Delta) \quad \mathit{and} \quad \langle e, h, s\rangle \hookrightarrow \langle e_1, h_1, s_1\rangle,$$

then there exists Δ_1, such that $s_1, h_1 \models \mathit{Post}(\Delta_1)$ and $\vdash \{\Delta_1\}\ e_1\ \{\Delta_2\}$.

Proof: By structural induction on e.

Theorem 6.2 (Progress) *Consider a closed configuration $\langle e, h, s\rangle$. If*

$$\vdash \{\Delta\}\ e\ \{\Delta_1\} \quad \mathit{and} \quad s, h \models \mathit{Post}(\Delta),$$

then either e is a value, or there exist s_1, h_1, and e_1, such that $\langle e, h, s\rangle \hookrightarrow \langle e_1, h_1, s_1\rangle$.

Proof: By structural induction on e.

Theorem 6.3 (Soundness) *Consider a closed configuration $\langle e, h, s\rangle$. Assuming that $\vdash \{\Delta\}\ e\ \{\Delta'\}$ and s,h $\models$ Post(Δ), then either $\langle e, h, s\rangle \hookrightarrow^* \langle v, h', s'\rangle$ terminates with a value v such that $(s' + [\mathtt{res} \mapsto v], h') \models Post(\Delta')$ holds, or it diverges $\langle e, h, s\rangle \not\hookrightarrow^*$.*

Proof Sketch: If the evaluation of e does not diverge, it will terminate in a finite number of steps (say n): $\langle e, h, s\rangle \hookrightarrow \langle e_1, h_1, s_1\rangle \hookrightarrow \cdots \hookrightarrow \langle e_n, h_n, s_n\rangle$. By Theorem 6.1, there exist $\Delta_1, .., \Delta_n$ such that, $s_i, h_i \models \mathit{Post}(\Delta_i)$, and $\vdash \{\Delta_i\}\ e_i\ \{\Delta'\}$. By Theorem 6.2, The final result e_n must be some value v (or it will make another reduction).

7 Experiments

Our verification system is built using Objective Caml. The proof obligations generated by the verification are discharged by the entailment checking procedure with the help of Omega Calculator [12].

In order to prove the viability of our verification method we tried our prototype implementation against a few examples from SPECjvm2008, a widely used Java benchmark created by SPEC. Due to the focus of our work, we only considered those tests from SPECjvm2008 that are related to exception handling. After annotating the tested methods with pre and post conditions, we were able to successfully verify all the tests. Due to the fact that the KeY approach is semi-automated in the sense that it occasionally prompts the user for choice of rewriting rules, while our approach is fully-automated, we cannot perform a direct comparison of the verification timings.

Among these examples, MyClass is a Java program emphasizing the use of exception handling in the presence of user defined exceptions. Its aim is to detect mishandling of the exception class hierarchy. The main objective of While and ContinueLabel examples is testing abrupt termination in the presence of loops. PayCard is a Java class from a real life Java application that makes heavy use of exceptions while modelling the behaviour of a credit card.

Figure 7 contains the timings obtained when using our system to verify the aforementioned examples.

Programs	LOC	Time (seconds)	Focus
Break (KeY)	20	0.11	`break` handling
MyClass (KeY)	33	0.10	exception hierarchy
While (KeY)	130	2.47	while loops and `break`
ContinueLabel (KeY)	100	0.95	imbricated while loops and `continue`
PayCard (KeY)	70	0.91	general exception handling
SPECjvm2008	190	1.20	general exception handling

Fig. 7. Verification Times

We also verified the examples presented in the paper. Method `list_alloc` was verified in 0.41 seconds, `list_alloc1` in 0.26 seconds and method `swap` in 0.09 seconds. Take note that the verification of `list_alloc1`, which ensures an improved strong exception safety guarantee as according to our approach, is faster by 36% than the verification of `list_alloc` which enforces the original strong guarantee defined in [15].

8 Concluding Remarks

We have presented a new approach to the verification of exception-handling programs based on a specification logic that can uniformly handle exceptions, program errors and other kinds of control flows. The specification logic is currently built on top of the formalism of separation logic, as the latter can give precise description to heap-based data structures. Our main motivation for proposing this new specification logic is to adapt the verification method to help ensure exception safety in terms of the four

guarantees of increasing quality introduced in [15] and extended in [9], namely no-leak guarantee, basic guarantee, strong guarantee and no-throw guarantee. During the evaluation process, we found the strong guarantee to be restrictive for some scenarios, as it always forces a recovery mechanism on the callee, should exceptions occur. Hence, we propose to generalise the definition of strong guarantee for exception safety. Our approach has been formalised and implemented in a prototype system, and tested on a suite of exception-handling examples. We hope it would eventually become a useful tool to help programmers build more robust software.

References

1. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.
2. Matthias Blume, Umut A. Acar, and Wonseok Chae. Exception handlers as extensible cases. In *APLAS*, pages 273–289, 2008.
3. C. David, C. Gherghina, and W. N. Chin. Translation and optimization for a core calculus with exceptions. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press, 2009.
4. Manuel Fähndrich and Alexander Aiken. Program analysis using mixed term and set constraints. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 114–126, London, UK, 1997. Springer-Verlag.
5. 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.
6. Jang-Wu Jo, Byeong-Mo Chang, Kwangkeun Yi, and Kwang-Moo Choe. An uncaught exception analysis for Java. *Journal of Systems and Software*, 72(1):59–69, 2004.
7. K. Rustan M. Leino and Wolfram Schulte. Exception Safety for C#. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference*, pages 218–227, Washington, DC, USA, 2004. IEEE Computer Society.
8. Rustan Leino. personal communication, Jan 2009.
9. Xin Li, H. James Hoover, and Piotr Rudnicki. Towards automatic exception safety verification. In *FM*, pages 396–411, 2006.
10. Roy A. Maxion and Robert T. Olszewski. Improving software robustness with dependability cases. In *28th International Symposium on Fault Tolerant Computing*, pages 346–355, 1998.
11. H.H. Nguyen, C. David, S.C. Qin, and W.N. Chin. Automated Verification of Shape And Size Properties via Separation Logic. In *VMCAI*, Nice, France, Jan 2007.
12. W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.
13. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE LICS*, pages 55–74, Copenhagen, Denmark, Jul 2002.
14. Olin Shivers and David Fisher. Multi-return function call. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 79–89, New York, NY, USA, 2004. ACM.
15. Bjarne Stroustrup. Exception safety: Concepts and techniques. In *Advances in Exception Handling Techniques*, pages 60–76, 2000.