Exceptions in OO Languages: Using them and Reasoning about them

Neelam Soundarajan and Stephen Fridella

Contact author: N. Soundarajan

Computer and Information Science
Ohio State University, Columbus, OH 43210
e-mail: {neelam,fridella}@cis.ohio-state.edu

Abstract. Exceptions are generally considered as *signals* with little or no internal structure to them. Using such exceptions, a method can signal to its caller that the operation could not be completed properly, and let the caller take appropriate corrective action. While this is sufficient to deal with simple situations, it severely limits the information that the method can provide to the caller when the exception is raised. In order to solve this problem, some OO languages treat exceptions as full-fledged objects. This adds to the power of the exception mechanism, but it also adds to the difficulty of the already complex task of understanding the behavior of programs that utilize exceptions. Hence, in order to effectively exploit the power of these mechanisms, we need appropriate techniques that can be used to reason precisely about the behavior of systems built using them. This paper develops such techniques and demonstrates their usefulness via some simple examples.

Keywords: Software quality; Software development methods.

1 Introduction

Exceptions are often [4, 5, 8] viewed as *signals* with no internal structure to them. A procedure or a method of a class[1] will, when it is unable to complete its task, *raise* an exception E. When this happens, the system searches for, and transfers control to, a matching *exception handler* along the calling chain. While exceptions of this kind are undoubtedly useful, and are sufficient to deal with some simple situations, the amount of information that the signaling method can provide to the caller by means of such exceptions is severely limited since the exception is little more than its name. In order to solve this problem, some OO languages such as *C++* and *Java* treat exceptions as full-fledged objects. In *Ada 95* [1], exceptions are not full-fledged objects but they are not just named signals either; instead, the Ada.Exceptions package defines the internal structure of, and a rich set of operations that may be performed on, exceptions. Allowing exceptions to have a rich internal structure, whether by treating them as full-fledged objects or via the somewhat more restricted *Ada 95* approach, adds

[1] In this paper, we will focus on class methods since we are primarily interested in Object-Oriented languages.

considerably to the power of the exception mechanism. As we will see in the next section, even relatively simple classes can usefully exploit the ability to include information in the internal structure of exceptions. But enriching the structure of exceptions also adds to the difficulty of the already complex task of understanding the behavior of programs that utilize exceptions. Hence, in order to exploit the power of these mechanisms, it is essential to develop appropriate techniques that can be used to reason precisely about the behavior of systems built using them. In this paper we develop such techniques and demonstrate their usefulness via some simple examples.

A number of authors [3, 4, 7, 8, 10] have considered ways to reason about the behavior of systems that utilize exceptions. While these systems differ from each other in details, what is common to all of them is that they deal only with the simple type of exceptions, i.e., exceptions that are named signals with no internal structure. Our goal is to extend some of this earlier work to allow for exceptions that may have complex internal structure, and may indeed be full-fledged objects.

There are two distinct problems that we need to address in a reasoning technique that deals with the behavior of systems that utilize exceptions as objects. The first, and this is common with the case of simple exceptions, is the sudden jump in the flow-of-control from the method raising the exception to the exception handler (which may be several steps away in the calling chain). The second, and this is a result of allowing exceptions to be complete objects, is the coming into existence of a new object corresponding to the exception being raised, with this object becoming available to the exception handler. Systems such as those of [4, 7] show us how to deal with the jump in the flow-of-control. The approach we develop builds on these earlier systems by providing appropriate notations to capture information about the exception object created when the exception is raised, and rules to reason about this object when analyzing the exception-handling code.

The key contributions of this paper may be summarized as follows:

- Argues for the importance of allowing exceptions to have a rich internal structure rather than just being named signals.
- Develops a reasoning technique for understanding the behavior of systems that utilize such exceptions.
- Applies the technique to analyze the behavior of a simple example.

The rest of the paper is organized as follows: In Section 2 we present a simple example to illustrate the advantages of considering exceptions as objects rather than merely as signals. In Section 3 we develop our reasoning technique for dealing with programs that raise and handle such exceptions. In Section 4 we apply the technique to the example of Section 2. In the final section we summarize our work, compare it with exisiting approaches to reasoning about exceptions, and identify some problems for future research.

2 Exceptions as objects

Consider a banking application consisting of two classes, Bank and Account; we will focus our attention on the Account class. It consists of a member variable[2] balance and operations getBal(), deposit(), and withdraw(), for getting the current balance in the account, and for depositing money into and withdrawing money from the account. In Figure 1 we have provided a precise specification of the

$$\underline{\mathsf{Account} :: \mathsf{getBal}() : \mathsf{real}}$$
$$\mathsf{pre} : \mathsf{true}$$
$$\mathsf{post} : (\mathsf{result} = \mathsf{balance}) \wedge !\{\mathsf{balance}\}$$

$$\underline{\mathsf{Account} :: \mathsf{deposit}(\mathsf{a{:}real}) : \mathsf{void}}$$
$$\mathsf{pre} : \mathsf{a} > 0.0$$
$$\begin{aligned}\mathsf{post} : &(\mathsf{balance@pre} \geq 1000.00 \Rightarrow \mathsf{balance} = \mathsf{balance@pre} + \mathsf{a}) \\ &\wedge (\mathsf{balance@pre} < 1000.00 \Rightarrow \mathsf{balance} = \mathsf{balance@pre} + \mathsf{a} - 1.0)\end{aligned}$$

$$\underline{\mathsf{Account} :: \mathsf{withdraw}(\mathsf{a{:}real}) : \mathsf{void}}$$
$$\mathsf{pre} : \mathsf{a} > 0.0$$
$$\begin{aligned}\mathsf{post} : &(\mathsf{balance@pre} \geq 1000.00 \Rightarrow \mathsf{balance} = \mathsf{balance@pre} - \mathsf{a}) \\ &\wedge (\mathsf{balance@pre} < 1000.00 \Rightarrow \mathsf{balance} = \mathsf{balance@pre} - \mathsf{a} - 1.0)\end{aligned}$$

Fig. 1. Specification of Account class

behavior of each operation of Account using standard pre- and post-conditions. The format we have used is consistent with that of the *Object Constraint Language (OCL)* [9] which is a part of the UML-standard[2]. The notation x@pre which may appear (only) in post-conditions denotes the value of x at the pre-condition stage, i.e., when the operation was invoked. Thus the specifications of deposit() and withdraw() tell us that there is indeed a service charge (of 1.0) for these operations if the balance in the account at the start of the operation is below 1000.0. We also use (but this is not officially part of OCL) the notation !{x,y} in the post-condition to mean that the final values of x and y are the same as their corresponding initial values. Thus the specification of getBal() in the figure tells us that it returns a result of type real; the pre-condition tells us that this operation may be invoked at any point (since true is always satisfied); and the post-condition tells us that the result returned by getBal() is the same as the value of balance, and furthermore the clause !{balance} assures us that a call to getBal() will not change the value of balance.

The specification of withdraw() in Figure 1 imposes no limits on the amount that may be withdrawn from an account. Real banks do not, of course, allow customers to withdraw arbitrary amounts. One possibility, to take account of this fact, would be to modify the pre-condition of withdraw() as follows:

$$\mathsf{a} \leq \mathsf{balance}$$

This would require the Bank class (or whichever class uses the Account class) to ensure, before invoking the operation b.withdraw(a), b being the account object

[2] For concreteness, we will use a *C++/Java*-like terminology and syntax but there is nothing language-specific about our work.

from which to withdraw, that the balance in b is no less than a. An alternative approach, and the one we will consider, is to have the withdraw() operation raise an exception if the amount being attempted to be withdrawn from the account is greater than its current balance[3]. A signal-like exception, let us call it InsufficientFunds, would seem to be all that is needed to deal with this situation, but consider the following code outline that appears in a method of the Bank class:

```
try { ... BMC ... }
catch( InsufficientFunds ) { ... EHC ... }
```

Suppose during execution of the bank method code BMC, there is a call to withdraw() which results in an exception being raised, and control is transferred to the exception handling code, EHC. If there is only one call in BMC to withdraw(), then it is clearly that call that must have resulted in the exception being raised, so we would know, for example, which particular account was involved, and we could take appropriate action in EHC, such as freezing the offending account. What if there were several calls in BMC to withdraw() and these calls involved different accounts? How then do we identify the offending account?

There are two possible solutions. The first is to rewrite the code so that each call to withdraw() is in a separate try block; in this case if control transfers to the exception handling code attached to a given try block, there would be no question about which account must have been involved. While this approach works, it results in considerable code duplication and the overall code is broken up into numerous try blocks. A better approach is to include information concerning the identity of the offending account *within the exception*. If exceptions were not just signals but could have a rich internal structure, this is easily done. All we would need is for the InsufficientFunds exception to have a member variable that is initialized at the time the exception is created to be a reference to the involved account. Note that the exception is created in the withdraw() operation and that operation, of course, knows the account that is involved. Indeed the operation must have been invoked on the particular account so a reference to this account would be contained in the *Java* variable this. In *Smalltalk* the self variable would contain this reference. The Ada.Exceptions package does not provide information concerning the identity of the object an operation on which resulted in the exception being raised; we believe it would be very useful to extend Ada.Exceptions to include this information; we will return to this in the final section.

[3] Using exceptions in this example may be something of an overkill; the caller could easily check, by using the getBal() operation prior to the call to withdraw(), that there is an adequate balance in the account, so strengthening the pre-condition is probably the simpler and hence better approach. As with any powerful mechanism, there is a danger with exceptions of their being misused. Because of space limitations, we will not go into a detailed discussion of the situations under which exceptions are appropriate to use. Our use of exceptions in this example is just to help explain in a simple setting why exceptions are usefully considered as objects rather than as just signals, and to illustrate our reasoning technique, and not intended as a paradigmatic example of when and where exceptions are appropriate.

This example is rather simple and as noted earlier, the use of exceptions is probably inappropriate. But the point is where exceptions are used, the ability to include information, such as the identity of the involved object, can be valuable to the code that has to handle the exception. In the next section we turn to reasoning about the operations that raise such exceptions and the code that handles them. Before concluding this section we should note that we ignore abstraction issues in this paper; this is not because abstraction is not important but because exception handling is an orthogonal issue and is our focus.

3 Reasoning About Exceptions

Suppose f is a method of a class C and that f may raise exceptions during its execution. The specification of f will consist of:

- The *pre-condition*, $f.pre$; this is an assertion over the values of the member variables of C at the time of the call to f and the values of the parameters to f; for convenience, we will assume that all parameters are passed by-value.
- The (normal- or) *n-post-condition*, $f.post$; this is an assertion over the values of the member variables of C at the time f finishes as well as their values at the time of the call (these being identified by the suffix @pre). If f returns a result, we use the symbol result in $f.post$ to refer to it.
- For each type of exception E_i that f may raise during its execution, the (exception- or) *e-post-condition*, $f.E_i.post$; this is an assertion over the parameter values of f, the initial values of the member variables of C at the time of the call, and their final values when f terminates by raising an exception of type E_i, *and* the values of the member variables of the (exception) class E_i. $f.E_i.post$ will not include information about result since if f terminates by raising an exception, it does not return a result.

This specification is valid if whenever execution of f begins with $f.pre$ being satisfied, one of the following occurs:

1. f terminates normally with the parameter values received by f, the initial and final values of C's data members, and the value of the result (if any) returned by f satisfying the n-post-condition $f.post$;
2. Or f terminates by raising an exception of type E_i and at the time of f's termination, the parameter values of f, the initial and final values of the data members of C, and the values of the data members of E_i (in the exception object that f throws) together satisfy the e-post-condition, $f.E_i.post$;

Note that if f were capable of terminating[4] by throwing either e_1 or e_2, both of

[4] We do not consider the case of non-termination so f must necessarily terminate, either normally or by raising an exception. In other words, ours is a total correctness, rather than partial correctness, approach. But in this paper, we will ignore the details of establishing termination since these are fairly standard and the addition

which are of the same type E_i, the same e-post-condition, $f.E_i.post$, describes both situations. This is reasonable since the exception handling code in the caller will be specified by the type of the exception it can handle, so the code corresponding to the type E_i must be designed to deal with both e_1 and e_2.

How do we show that f meets its specification? For this, we need *rules* corresponding to the various statements that may appear in f. For statements not concernced with exceptions, we can borrow standard rules [6], with minor modifications to let us 'carry forward' information about exceptions. For example, our rule for sequential composition is:

R1. Sequential Composition

$$\frac{\begin{array}{l}\{\,p\,\}\;S_1\;\{\,q'\,\}\;\{E_1.post,\;E_2.post,\ldots,E_n.post\}\\ \{\,q'\,\}\;S_2\;\{\,q\,\}\;\{E_1.post,\;E_2.post,\ldots,E_n.post\}\end{array}}{\{\,p\,\}\;S_1;\;S_2\;\{\,q\,\}\;\{E_1.post,\;E_2.post,\ldots,E_n.post\}}$$

$\{E_1.post,\; E_2.post, \ldots, E_n.post\}$ are the e-post-conditions corresponding to exceptions that may be raised during execution of $S_1; S_2$. To exceute $S_1; S_2$, we start with S_1; S_1 may terminate by raising an exception E_i, in which case $S_1; S_2$ also terminates with E_i; or S_1 may terminate normally, in which case we execute S_2 which may terminate normally or with an exception. The first antecedent of R1 requires us to show that S_1 will terminate normally with q' being satisfied, or by raising an exception E_i, with $E_i.post$ being satisfied. The second antecedent similarly requires us to consider S_2 but only if S_1 finishes normally, that is q' is satisfied when S_2 begins. Given these antecedents, R1 allows us to conclude that $S_1; S_2$ will terminate normally with q being satisfied at termination, or will terminate with the exception E_i being raised with $E_i.post$ being satisfied.

Although sequential composition itself cannot raise an exception, S_1 or S_2 may do so. That is why R1 requires us to show that (if and) when this happens, the corresponding e-post-condition is satisfied. But R1 seems to require the exceptions that may be raised during the executions of S_1 and S_2 to be identical to each other. What if they are not? R2 will allow us to deal with such situations:

R2. Rule of Consequence

$$\frac{\begin{array}{l}\{\,p\,\}\;S\;\{\,q\,\}\;\{E_1.post,\;\ldots,E_k.post\}\\ p' \Rightarrow p,\; q \Rightarrow q',\; E_1.post \Rightarrow E_1.post',\;\ldots,\; E_k.post \Rightarrow E_k.post'\end{array}}{\{\,p'\,\}\;S\;\{\,q'\,\}\;\{E_1.post',\;\ldots,E_k.post',E_{k+1}.post'\}}$$

This is a natural generalization of the usual rule of consequence that allows us to strengthen the pre-condition (from p to p'), weaken the n-post-condition (from q to q'); the rule also allows us to weaken e-post-conditions; and to *add* an e-post-condition corresponding to a new exception; this is reasonable because

of exceptions does not change these details.

We should note also that due to space constraints, we do not include class invariants in our specifications. In practice, these invariants can be very important especially in dealing with abstraction, since they allow us to capture important relations between the concrete view of the class and the abstract view.

if we have already shown that S must terminate normally or by raising one of $E_1, \ldots, E_k$, we can also say that S must terminate normally or by raising one of $E_1, \ldots, E_k, E_{k+1}$. It is important to be able to add exceptions in this manner; this allows us to meet the requirement of R1 that the exceptions included in the results concerning S_1 and S_2 to be identical to each other. Using R2 we can meet this requirement by adding to the result about each of S_1, S_2, the exceptions that are missing from it. This is similar to what happens in the standard rule for if-then-else where we are required to weaken the post-conditions of the two branches of the if-then-else to make them identical to each other.

Let us now consider stantements that raise exceptions. The first of these is the throw statement. Consider what happens when

throw $E_i(\bar{x})$

is executed. This constructs a new (exception) object of type E_i by calling the constructor function of the class E_i ($\bar{x}$ is passed as arguments to this function), and 'throws' it. We will use the symbol ε to denote this newly created exception object. Since at most one exception object will be in existence at any time, no confusion should arise by our use of a single symbol to denote this object. The following rule captures the effect of the throw:

R3. Throw Statement

$$\frac{(\, p \wedge E_i.\mathit{ppost}\,{}^{\bar{p},\,\bar{d}}_{\bar{x},\,\varepsilon.\bar{d}}\,) \Rightarrow E_i.\mathit{post}}{\{\, p\,\}\ \mathsf{throw}\ E_i(\bar{x})\ \{\,\mathit{false}\,\}\ \{E_i.\mathit{post}\}}$$

$E_i.\mathit{ppost}$ is the post-condition of the construction function of the class E_i obtained from the specification of that class[5]. This post-condition will be an assertion over $\bar{p}$, its formal parameters, and $\bar{d}$, the member variables of the class E_i. Since the actual arguments being passed to the constructor function are $\bar{x}$ and since the exception object is named ε, we need to perform the indicated substitutions; note that the class E_i may have a member variable whose name, say y, is identical to that of a member of the class C and $E_i.\mathit{post}$ may refer to both, but this will not cause any problems because the former would be denoted by ε.y in this assertion. The rule for throw then requires us to show, given the pre-condition of the throw statement and the post-condition of the constructor of the E_i class (with the specified substitutions), that the appropriate e-post-condition, $E_i.\mathit{post}$ will be satisfied. And since control will leave this point, we may assert *false* as the n-post-condition of the throw statement.

There is another statement that may also result in an exception being raised, this being a call to another method, with the called method raising an exception during its execution. This is handled by the following rule:

[5] Some languages allow a class to have more than one constructor; we ignore this possibility. More to the point, we also ignore the possibility that the constructor itself raises an exception. In general it may be a good idea to not allow constructor functions to raise exceptions, else we have to deal with 'partially constructed' objects.

R4. Method Call

$$\frac{\begin{array}{l} p \Rightarrow g.pre^{\bar{u},\,\bar{d}}_{\bar{y},\,\mathsf{x}.\bar{d}} \\ \exists \bar{d}'.\,[\,p^{\mathsf{x}.\bar{d}}_{\bar{d}'} \wedge g.post^{\bar{u},\,\#\bar{d},\,\bar{d}}_{\bar{y},\,\bar{d}',\,\mathsf{x}.\bar{d}}\,] \Rightarrow q \\ \exists \bar{d}', \bar{y'}.\,[\,p^{\bar{y},\mathsf{x}.\bar{d}}_{\bar{y'},\bar{d}'} \wedge g.E_j.post^{\bar{u},\,\#\bar{d},\,\bar{d}}_{\bar{y},\,\bar{d}',\,\mathsf{x}.\bar{d}}\,] \Rightarrow \\ \qquad\qquad\qquad E_j.post,\ \mathit{for\ all}\, E_j\ \mathit{that}\, g\ \mathit{may\ raise} \end{array}}{\{\,p\}\ \mathsf{x}.g(\bar{y})\ \{\,q\,\}\ \{E_1.post,\ E_2.post, \ldots\ E_n.post\}}$$

Note that x is the object that g is being applied to, the other arguments being $\bar{y}$. So references to member variables and formal parameters $\bar{p}$ of g, in the pre- and post-conditions of g have to be replaced by the corresponding components of x and $\bar{y}$. The first antecedent of the rule requires us to show that the pre-condition of g is satisfied immediately before the call. The second considers the case when g terminates normally; in this case we are required to show, given the n-post-condition of g (applied to the actual arguments passed to it and to x's members), and whatever information we have (concerning the other variables of the caller) from the pre-condition p of the call, that the n-post-condition q will be satisfied; note that whatever p says about the members of x (or the other arguments to g) apply to their values *before* the call, not their current values; this is taken care of by the quantifier. The last antecedent does the same with respect to the e-post-conditions for all exceptions that g may raise. This rule again is a natural generalization of the standard rule for procedure call; the key addition that R4 makes is to take account of the possibility of g terminating by raising an exception rather than normally.

Let us now consider the statement that allows us to *handle* the exceptions. Consider the following segment of code:

$\mathsf{try}\{\,S\,\}$
$\mathsf{catch}(E_1\ e_1)\ \{\,S_1\,\}$
$\ldots$
$\mathsf{catch}(E_k\ e_k)\ \{\,S_1\,\}$

If S terminates normally, the catch clauses have no effect; if it terminates with an exception of type E that is not one of $\{E_1, \ldots, E_k\}$, the catch clauses have no effect, and the execution of the entire command terminates with the exception of type E being thrown (to be handled, presumably, by a catch attached to the try block of the calling method). If S terminates with an exception of type E_i, control transfers to the corresponding catch clause, e_i is bound to the exception object (ε) and S_i begins execution; at this point, the exception is considered handled, and e_i is just another object. The execution of S_i may terminate normally in which case the execution of the entire try block finishes normally (and e_i goes out of scope); or S_i might raise an exception! In the interest of keeping the presentation simple, we ignore this possibility in the rule below (but we will briefly indicate how the rule could be modified to account for this). The following rule captures this behavior of the try block. In the conclusion of the rule we use C_i to denote the i^{th} catch clause catch $(E_i\ e_i)\ S_i$.

R5. try block

$$\{\,p\,\}\;S\;\{\,q\,\}\;\{E_1.post,\;\dots\;,E_i.post\,,E_{i+1}.post\,,\dots\;,E_k.post\}$$
$$E_j.post \Rightarrow p_j\,{}^{e_j}_{\varepsilon}\;,\;j=1,\dots,i$$
$$\{\,p_j\,\}\;S_j\;\{\,q\,\}\;,\;j=1,\dots,i$$

$$\{\,p\}\;\mathsf{try}\;S\;C_1\;\dots\;C_i\;\{\,q\,\}\;\{E_{i+1}.post,\;,\dots\;,E_k.post\}$$

We are first required to establish the appropriate result about S, the main body of the try block. There are k e-post-conditions here, the first i of which will be handled by the $C_1, \dots, C_i$. Next we must ensure that if an exception of type E_j is raised, the pre-condition p_j of the corresponding catch clause, with ε being substituted for e_j, is satisfied. The third antecedent ensures that the body of each catch clause does what it is supposed to do – establish the post-condition q. The conclusion of the rule lists only the e-post-conditions $E_{i+1}.post, \dots , E_k.post$ since the exceptions of type $E_1, \dots, E_i$ will be handled by the catch clauses.

Let us now consider the possibility of S_j raising an exception. If S_j were to raise an exception of type E, the entire block would terminate with this exception – even if E were one of $\{E_1, \dots, E_i\}$; so the catch clauses only apply to exceptions raised during execution of the main body S. How do we modify R5 to allow for this possibility? Two changes would have to be made: add, to the result which is the third antecedent of the rule, e-post-conditions corresponding to the exceptions that S_j might raise; and add each of these (corresponding to each of $S_1, \dots, S_i$) to the conclusion of the rule.

We conclude this section with an observation. In practice, formal specification of program behavior is somewhat more important than formal verification using our rules. The point is that the rules give us an extra tool that be can be used to ensure that the programs behave as per their specifications, but we should not blindly rely on them since it is quite possible to make mistakes in applying them. Hence they should be used to complement our operational understanding, not replace them. But specifications serve as the *basis* for any kind of understanding since they tell us how the program is intended to behave. In our discussion of the example in the next section, we will be careful in writing down the specification of the Account class precisely but will be a bit informal in the rest of our reasoning.

4 Example

Let us consider how our technique applies to the Account example of Section 2. As we noted in Section 2, the withdraw() function, when it raises an InsufficientFunds exception, could include a reference to the particular Account object. Further, information about the amount of money the client attempted to withdraw (this information is not in the state of the Account) may be useful. Consider the following exception class in a *Java* style syntax:

```
public class InsufficientFunds {
  public real amtOver; public Account act;
  public InsufficientFunds( real amt, Account a ) { amtOver = amt; act = a; }
}
```

Next we need to revise our original specification (see Figure 1) of withdraw() to include information about the InsufficientFunds exception it may raise:

$\underline{\mathsf{Account :: withdraw(a:real) : void}}$
$\mathsf{pre} : \mathsf{a} > 0.0$
$\mathsf{n\text{-}post} : (\mathsf{balance@pre} \geq \mathsf{a}) \wedge !\{\mathsf{a}\} \wedge$
$\quad (\mathsf{balance@pre} \geq 1000.00 \Rightarrow \mathsf{balance} = \mathsf{balance@pre} - \mathsf{a}) \wedge$
$\quad (\mathsf{balance@pre} < 1000.00 \Rightarrow \mathsf{balance} = \mathsf{balance@pre} - \mathsf{a} - 1.0)$
$\mathsf{InsufficientFunds\text{-}post} : (\mathsf{balance@pre} < \mathsf{a}) \wedge !\{\mathsf{a,balance}\} \wedge$
$\quad (\varepsilon.\mathsf{act} = \mathsf{this}) \wedge (\varepsilon.\mathsf{amtOver} = \mathsf{a})$

The e-post-condition states that when an InsufficientFunds exception is raised, the balance and a are unchanged, the original balance was less than a, and the newly created exception object (ε) will contain a reference to the account (in act) and a copy of a (in amtOver). This post-condition by itself does not guarantee that if balance@pre is less than a the exception will be raised. This follows, however, from the first clause of our (slightly modified) n-post-condition, which states that upon normal terminiation, the original value of balance must be greater than or equal to a. So this specification tells us that if balance@pre is less than a, the operation would terminate with an InsufficientFunds exception, else it will terminate normally.

Consider the following piece of client code (perhaps part of the Bank class):

```
Account ac1, ac2; ...
try { ac1.withdraw(50.0); ... ac2.withdraw(50.0); ... }
catch InsufficientFunds (e) {
  print( e.act.getName(), e.act.getBal(), e.amtOver ); }
```

This code invokes withdraw() on two different accounts, ac1, ac2. If an exception is raised, the catch clause prints the name of the account holder[6], the account's current balance, and the amount of the withdrawal that caused the overdraft.

What kind of reasoning can we do about this code? Suppose we know at the start of the try block that the balance in ac1 is more than 50.0; can we show (assuming there are no further withdraw()'s in the elided code) that the name that is printed out by the catch clause is not that of ac1's holder? Yes, indeed. What we would need to do is to have, as pre-condition for this clause, an assertion that says ε.act is ac2. Given this pre-condition, it is straightforward to use the rule for the catch clause to show that the name it will print will be that of ac2's holder. But the try block rule will require us to ensure that the code within the try block is consistent with this pre-condition, that is any statement in that block that may raise an InsufficientFunds exception will make sure that this assertion is satisfied. It would seem this would be difficult to ensure with respect to the first invocation of withdraw() because the method is applied to ac1; but this is not really a problem; recall that the e-post-condition of withdraw() tells

[6] We are assuming that the Account class provides an operation getName() which can be invoked for this purpose.

us that the balance@pre in the account must be less than 50.0, the amount being withdrawn, but the pre-condition of try, we assumed, told us this was not the case. So combining these two we can conclude that ac1 could not be the cause of the exception. (More formally, we would use R2, the rule of consequence, to show that the e-post-condition of the first call to withdraw() does imply the required pre-condition of the catch clause.)

5 Discussion

Exception handling mechanisms provide a powerful tool for structuring complex programs. They allow the main code to be separated cleanly from the code for dealing with exceptional situations. The simplest kind of exceptions are just named signals; while such exceptions are useful, the amount of information that the signaling method can provide to the handler by means of such exceptions is very limited. Our analysis showed that even simple examples can benefit greatly from having exceptions that have internal structure.

Ada 95 provides for a limited amount of information to be included in exceptions. Specifically, the operation that raises the exception may provide an arbitrary string to be stored in the exception, and the handler may retrieve this using the Exception_Information operation provided by the Ada.Exceptions package. In our example the withdraw() operation could have used this to store the name of the holder of the account in the InsufficientFunds exception. While this would allow the handler to, for example, print out this name, it would not allow us to do anything more complex, such as *freezing* the account[7]. We believe this is a common pattern – often the handler will need to know which object was responsible for the exception being thrown; hence it would be very useful for the Ada.Exceptions package to be extended to provide an access variable that would allow the handler code to refer to this object. Even this may be a bit limited; it would not allow, for example, the withdraw() operation to include information about *how much* was attempted to be withdrawn when the exception was raised. To include such information, it would seem exceptions would have to be treated as full-fledged objects.

Exception handling, like most powerful mechanisms, is fairly complex mechanism and warrants careful treatment in the form of formal specifications of program behavior. We introduced a notation, a direct generalization of the standard pre- and post-condition notation, to precisely express the behavior of operations of classes that may raise exceptions. The key was to include, in addition to the normal post-condition, a set of e-post-conditions, corresponding to the various exceptions that an operation may raise; and include information in the e-post-conditions about not only the state of the object raising the exception but also about the exception object. Given such a specification, we can reason about the behavior of the exception handling code that has to deal with the exceptions

[7] Freezing the account would of course require the Account class to provide a suitable operation; the point here is the handler also needs to know *which* account to freeze.

that may be raised. And we presented rules that may be used for this task. Previous work, for example [4, 7, 8], in the area has considered ways of dealing with simple types of exceptions with no internal structure to them. The specification and reasoning system we have presented is an important extension of these earlier systems to account for exceptions with rich, internal structure. Using our approach, designers will be able to specify and reason about the behavior of systems that take advantage of exceptions as full-fledged objects.

We conclude with a couple of problems for future work. In the kind of exception mechanism we have considered, once an exception is raised, the method that raises exception terminates. If the caller handles the exception, then control continues with whatever statement follows the try-catch block of the caller. An alternative that has been suggested is to allow control to *resume* in the method that raised the exception. This may be more appropriate in certain circumstances, and we plan to investigate how to change our formalism to apply to this type of exception mechanism. A second problem that is worth investigating is the relation between inheritance and exceptions. This is important because in languages such as *Java*, an exception handler will catch not only exceptions of the specified type but also any subtype. Changing our formalism to deal with this should be relatively easier; for example, we would have to revise the rule for the try, so that we consider a given catch clause not only if the type of the exception matches but also if it is a subtype.

References

1. J. Barnes. *Programming in Ada 95.* Addison-Wesley, 1998.
2. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.
3. N. Cocco and S. Dulli. A mechanism for exception handling and its verification rules. *Comput. Lang.*, 7:89–102, 1982.
4. F. Cristian. Correct and robust programs. *IEEE Trans. on Software Eng.*, 10:163–174, 1984.
5. J.B. Goodenough. Exception handling: issues and a proposed notation. *Comm. ACM*, 18:436–449, 1975.
6. D. Gries. *Science of Programming.* Springer-Verlag, 1981.
7. D.C. Luckham and W. Polak. Ada exception handling: An axiomatic approach. *ACM TOPLAS*, 2:225–233, 1980.
8. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.
9. J. Warmer and A. Kleppe. *The Object Constraint Langauge.* Addison-Wesley, 1999.
10. S. Yemini and D. Berry. A modular verifiable exception handling mechanism. *ACM TOPLAS*, 7:214–243, 1985.

This article was processed using the LaTeX macro package with LLNCS style