Exception safety for C#

K. Rustan M. Leino Wolfram Schulte
Microsoft Research
Redmond, WA, USA
{leino,schulte}@microsoft.com

Abstract

Programming-language mechanisms for throwing and handling exceptions can simplify some computer programs. However, the use of exceptions can also be error prone, leading to new programming errors and code that is hard to understand. This paper describes ways to tame the exception usage in C#. In particular, the paper describes the treatment of exceptions in Spec#, an experimental superset of C# that includes code contracts.

0. Introduction

Modern programming languages, including C# [24], provide *exceptions* as a mechanism to help programmers handle errors systematically. While exceptions can benefit the architecture of an application, the use of exceptions is delicate and can lead to problems in its own right. For example, unrestricted use of exceptions often leaves the program in an undetermined state, from which it is difficult to recover. As another example, without care, the new control flows introduced by the throwing of exceptions can lead to precious resources being leaked.

Despite these shortcomings, we argue that throwing and handling exceptions is still the best known way to propagate error information. But we advocate that the exceptions be used according to a checkable discipline. In this paper, we describe such a discipline applied to C# and the .NET Framework. In particular, we describe the exception features in our design of Spec#, a language that is a superset of C# and includes code contracts. Our design is tailored to C# and the way methods are documented in the .NET Framework, but it more generally contributes programmer-specified exceptions in parameter validation (**otherwise** clauses) and an interplay between object invariants and checked exceptions.

We base our exception discipline on work by Goodenough [9], who classifies the failures that exceptions can signal into *client failures* and *provider failures*. (Goodenough called these *domain failures* and *range failures*, respectively, but we have found that those terms are sometimes confusing.) Client failures occur when a procedure's inputs are not acceptable. Client failures correspond to parameter validation in Spec#. Provider failures occur when a procedure is unable to complete the task it's supposed to perform. We further divide provider failures into two subclassifications, *admissible failures* and *observed program errors*. Suppose a procedure reads bits from a network channel. An example of an admissible failure would then be that the received bits contain too many parity errors. To make note of the possibility of such admissible failures and to give the implementation an out in such situations, Spec# provides *checked exceptions* and *throws sets*. The other subclassification of provider failures occurs if the failure is due to an intrinsic error in the program (*e.g.*, an array bounds error) or a global failure that's not particularly tied with the procedure (*e.g.*, an out-of-memory error). To signal such failures, Spec# uses *unchecked exceptions*, which do not need to be listed in the procedure's throws set.

The Spec# compiler enforces the exception discipline in two ways: it enforces some parts of the discipline by performing various compile-time type checks, and it enforces other parts of the discipline by introducing various run-time checks into the target .NET virtual-machine code it generates. A Spec# program can also be checked entirely at compile time, using the program checker Boogie [2]. This paper is not a reference guide for Spec#, but serves to highlight the ways in which Spec# helps programmers use exceptions.

The design of Spec# draws from several previous efforts and designs, including: the Eiffel language (which includes code contracts) [18], the Java language (which admits a CLU- or Modula-3-like discipline of using exception) [10], the Java Modeling Language (JML, which provides contracts in the context of Java) [13, 3], and the current design of the C# language and .NET Framework. Section 8 gives a more detailed look at previous work.

1. Exceptions in C# and the .NET Framework

This section briefly reviews exception handling mechanisms in C# and the .NET Framework.

C# provides basic support for exception handling. The statement $\mathbf{throw}\ E;$ interrupts the control flow and signals an error or other condition by the exception denoted by the expression E. The statement

$$\mathbf{try}\ \{\ S\ \}\ \mathbf{catch}\ (X\ x)\ \{\ T\ \}$$

catches any exception of type X thrown in statement S (or in code called from S) and then invokes the "handler" statement T, in which x denotes the exception thrown. A **try** statement can have multiple **catch** clauses, in which case the first one whose exception type matches the thrown exception is picked. The statement

$$\mathbf{try}\ \{\ S\ \}\ \mathbf{finally}\ \{\ T\ \}$$

executes statement S followed by the "clean-up code" statement T, and then, if S terminates with an exception x and T terminates normally, re-throws x. For syntactic convenience, C# allows a **try-finally** statement statement whose **try** block is a **try-catch** statement to be written simply as that **try-catch** statement followed by the **finally** block.

A C# exception is an object whose type is a subclass of $System.Exception$. The exception types thus form a hierarchy. Here is a portion of that hierarchy:

- $System.Exception$
  - $System.SystemException$
    - $System.ArgumentException$
    - $System.ArithmeticException$
    - $System.IndexOutOfRangeException$
    - $System.InvalidCastException$
    - $System.IO.IOException$
    - $System.OutOfMemoryException$
  - $System.ApplicationException$

Since they are objects, exceptions can contain data in object fields.

A C# program can throw any exception at any time, but the type of the exception is intended to indicate the reason for throwing the exception. Program-specific exception types are declared as new classes, typically deriving from $System.ApplicationException$. Exceptions are also thrown by the .NET virtual machine (the Common Language Runtime, CLR) in the event that the program misuses a virtual-machine instruction (such as dereferencing the null reference) or exhausts some resource (such as memory).

C#, unlike Java, doesn't have throws sets. Methods can throw any exception without making this explicit in the method signature. Thus, in C#, one has to rely completely on informal comments, which can be wrong or simply missing. It is not possible for a caller to be certain a call won't result in a thrown exception. Moreover, when an exception is thrown, it is unclear in which state the callee and its reachable data structures remain.

2. Parameter validation

A common usage of exceptions in the .NET Framework is to report that a method has been invoked under inappropriate circumstances. We refer to this usage as *parameter validation*. For example, the $GetString$ method of class $Encoding$ in the current Base Class Library (BCL),

$$\mathbf{public\ virtual\ string}\ GetString(\mathbf{byte}[]\ bytes);$$

is not allowed to be called with a null parameter. The documentation specifies this as follows:

> A $NullArgumentException$ will be thrown if $bytes$ is **null**.

Here, the throwing of a $NullArgumentException$ indicates an error at a call site. In a correctly written program, no $NullArgumentException$ will ever be thrown.

In Spec#, parameter validation is specified using *preconditions*. For example, the documentation of $GetString$ above can be incorporated directly into the program text, as a part of $GetString$'s signature:

$$\mathbf{public\ virtual\ string}\ GetString(\mathbf{byte}[]\ bytes)\ \mathbf{requires}\ bytes\ !=\ \mathbf{null}\ ;$$

This declaration makes manifest that it is an error to invoke $GetString$ unless $bytes$ is not null. The Spec# compiler emits run-time checks to enforce preconditions; if a check fails, an unchecked $RequiresViolationException$ is thrown. The preconditions are checked in the order in which they are declared (when all preconditions hold, as would be the case in a correct program, this order doesn't matter, because the compiler insists that the expressions given as preconditions have no side effects on the program state).

Preconditions are inherited. That is, a declared precondition is in effect for all overrides of a virtual method. Consequently, the Spec# compiler emits run-time checks that are applied regardless of which method implementation a call dynamically dispatches to, eliminating the possibility that an override would leave off its parameter validation. In .NET, one expects parameter validation to occur, so Spec# does not allow preconditions to be weakened in subtypes (for more information, see [2]).

To support the programming practice used with the .NET Framework, a Spec# precondition declaration can indicate an unchecked exception type other than the default $RequiresViolationException$. For example, to follow the

original documentation for *GetString* above, one can declare:

```
public virtual string GetString(byte[] bytes)
  requires bytes != null
    otherwise NullArgumentException ;
```

In the event that a caller does happen to call this method with null, the run-time check emitted by the Spec# compiler will detect the error and throw a *NullArgumentException*.

By including preconditions as an integral part of a method's signature, Spec# makes it easy to explicate the intended usage of methods, which we'd like to encourage. However, we are sensitive to the fact that some preconditions may appear too costly to check at run time. Therefore, Spec# allows .NET *custom attributes* to be associated with each precondition. Consequently, it is possible to add the run-time check only in certain builds. For example, the declaration:

```
public static
int Find(IComparable[] a, IComparable val)
  requires val != null ;
  [Conditional("DEBUG")]
  requires IsSorted(a) ;
```

instructs the compiler to emit into the debug build code for checking both preconditions, but emit into the retail build code for checking the first precondition only.

3. Exceptional returns

In a correct program, the exceptions described in the previous section are never thrown. That is, the exceptions that may be thrown due to violated preconditions signal error conditions. But there is another use of exceptions, which is to signal a rare condition that may require special handling in the user program. An example of such an exception is *SocketClosedException*, which signals that the applied operation could not be performed due to some external event. Such an exception is typically caught and handled by the application, but this handling can be placed off to the side in a **catch** block, which can handle the possible failure of several operations in the corresponding **try** block.

Since it is important that there be a handler for this other kind of exception and since it is important for application writers to be aware of any such exceptions that a called method may throw, the Spec# language provides support in two ways.

First, Spec# separates exception types into two kinds. A *checked exception* signals a rare condition that is nevertheless expected to be handled by an application, an admissible failure. In contrast, an *unchecked exception* is used to signal a client failure (like *RequiresViolationException*) or observed program error (like *NullReferenceException*). Unchecked exceptions are typically caught only by the run-time system's default exception handler (which terminates the program) or by a handler in an outermost tier of an application.

An exception class is a checked-exception type if it implements the Spec#-provided interface *ICheckedException*.

Second, Spec# insists that any checked exception that can be thrown by a method be accounted for in the method's signature, namely in the method's throws set. For example, the declaration

```
public string ReadMessage()
  throws SocketClosedException ;
```

says that method *ReadMessage* may terminate exceptionally with a *SocketClosedException* (or with an exception of any subclass of *SocketClosedException*), but does not terminate exceptionally with any other kind of checked exception.

It is also possible in Spec# to say something about the program state in the event that an exception is thrown. (One can also say something about the program state in the event of normal termination, but that's not our focus here.) For example,

```
public string ReadMessage()
  throws SocketClosedException
    ensures unchanged(this^Connection) ;
```

says that if *SocketClosedException* is thrown, then the values of all fields declared for **this** in *Connection* and its superclasses are the same as they were upon entry to the method. If a method throws an exception whose type is a subtype of several exception types listed in the throws set, then all corresponding **ensures** clauses apply.

Using conservative syntactic data-flow rules somewhat akin to the rules of definite assignment, the Spec# type checker makes sure that all checked exceptions possibly thrown by a method body are accounted for in the method's throws set. Since Spec# has to retrofit C# with checked exceptions (unlike Java, where checked exceptions were built in from the start), Spec# has to put a restriction on **throw** statements to avoid being foiled by up-casts of a checked exception to an unchecked-exception type. In particular, if the static type of the expression E in

```
throw E ;
```

is not a checked exception, then Spec# checks at run time that the value produced by E indeed is not a checked exception.

A program written entirely in Spec# is type safe, also with respect to checked exceptions. That is, in any execution, any value held by a variable is allowed by the variable's declared type and any checked exception thrown by

a method is allowed by the types in the method's throws set. However, this type safety for checked exceptions is not guaranteed whenever a body of Spec# code calls code in a non-Spec# assembly, because other .NET languages don't enforce the discipline of checked exceptions.

We conclude this section by pointing out the differences between parameter validation and admissible failures. Parameter validation throws unchecked exceptions for client failures. These exceptions typically propagate up the call chain, until they are caught by a handler in an outermost tier of an application. In contrast, checked exceptions specified in throws sets become the responsibility of the caller.

Let us give an example of this difference by considering a *Hashtable* class that provides two lookup methods. One lookup method assumes the given key to be in the table, the other works even if the key is not in the table:

```
public object Lookup(object key)
  requires key != null && ContainsKey(key) ;
  throws ;
public object TryLookup(object key)
  requires true ;
  throws KeyNotInHashtable
    ensures !ContainsKey(key) &&
            unchanged(this^Hashtable) ;
```

Suppose you are a client of this class. Sometimes, you just want to look up the value at a key that you "know" is in the hashtable. In this case, you would use *Lookup*. The fact that you assume the key to be in the table means you're not going to write an exception handler for the call. If your assumption is wrong, the call to *Lookup* will result in an unchecked *RequiresViolationException*. Other times, you don't know whether or not the key is in the table. In this case, you would call the *TryLookup* method and would need to provide a handler for the checked *KeyNotInHashtable* exception.

4. Object invariants

If an exception is thrown in the midst of a data-structure update, then that data structure may be left in a corrupted state, that is, in a state where the data structure's invariant is broken. Spec# allows data-structure invariants to be recorded as *object invariants*. For example, the declarations

```
class Map {
  object[] domain ;
  object[] range ;
  invariant domain != null && range != null ;
  invariant domain.Length == range.Length ;
  int cardinality ;
  invariant 0 < cardinality &&
            cardinality <= domain.Length ;
```

say that in "stable states", the domain and range arrays of maps are not null, they have positive and equal capacities, and the map's cardinality is also positive and does not exceed that capacity. A "stable state" obtains when the program is not operating on the object. To make that notion precise, which is necessary in order to enforce object invariants, Spec# features a statement **expose**, which brackets the operations on an object. For example, here is a method that increases the capacity of a map:

```
public void IncreaseCapacity() {
  expose (this) {
    object[] d = new object[2 * domain.Length] ;
    object[] r = new object[2 * range.Length] ;
    domain.CopyTo(d, 0) ;
    range.CopyTo(r, 0) ;
    domain = d ;
    range = r ;
  }
}
```

The idea is that an object's fields are modified only within an **expose** statement on that object (or within calls performed within that **expose** statement), somewhat akin to the way shared variables in C# are to be accessed only within appropriate **lock** statements and the way certain disposable objects are to be used only within **using** statements. At the moment, the policy of operating on an object's fields only within an appropriate **expose** statement is not checked at run time, but is checked by the compile-time program checker Boogie. An object's invariant is supposed to hold at the end of an **expose** statement, which is checked at run time (and, by Boogie, at compile time). If the invariant does not hold, an *InvariantException* is thrown. (For more details and motivation about Boogie's checking of object invariants, see [1].)

In the examples in this section, we show the **expose** statements explicitly. However, by default, Spec# wraps the bodies of public instance methods within an

```
expose (this) { ... }
```

statement, which covers the most common usage of **expose**. Hence, explicit **expose** statements are not as ubiquitous in the code as one might first think.

Object invariants work with exceptions in the following way. If the body of an **expose** statement terminates exceptionally with a checked exception, then the object invariant is still checked, as it would be upon normal termination of the **expose** body. Since checked exceptions are presumed to be caught by the program, this checking of object invariants ensures that further operations find the object in a consistent state. For example, if the first call to *ReadMessage*

results in a *SocketClosedException* in the following code:

```
expose (this) {
  object[] d = new object[2 * domain.Length] ;
  domain.CopyTo(d, 0) ;
  domain = d ;
  domain[cardinality] = conn.ReadMessage() ;
  object[] r = new object[2 * range.Length] ;
  range.CopyTo(r, 0) ;
  range = r ;
  range[cardinality] = conn.ReadMessage() ;
  cardinality++ ;
}
```

then the checking of the object invariant at the end of the **expose** statement will result in the throwing of an *InvariantException* (carrying the thrown *SocketClosedException* as an inner exception).

Unchecked exceptions, on the other hand, are used differently, and there often isn't anything sensible an application can do in response to an unchecked exception. Therefore, Spec# does not do any checking of object invariants when the body of an **expose** statement terminates exceptionally with an unchecked exception. Instead, unchecked exceptions are propagated. For example, if the second allocation fails in the following code:

```
expose (this) {
  object[] d = new object[2 * domain.Length] ;
  domain.CopyTo(d, 0) ;
  domain = d ;
  object[] r = new object[2 * range.Length] ;
  range.CopyTo(r, 0) ;
  range = r ;
}
```

then the whole **expose** statement terminates exceptionally with an *OutOfMemoryException* and the data structure of the *Map* object is left in a corrupted state.

We note that a simple programming pattern that avoids many possible corrupted-state errors by limiting the window of vulnerability is to first prepare new values for all fields and then perform a series of consecutive assignment statements, as in the *IncreaseCapacity* method shown above.

5. Resource Usage

A danger with exceptions that we have not yet discussed arises with objects whose resources need to be disposed. Resources like file and widget handles, sockets, and registry entries are precious; they should be released as quickly as possible and shouldn't be leaked when exceptions are thrown.

In the .NET Framework, a resource is a class or struct that implements the *Dispose* method of the interface *System.IDisposable*. Code that wants to dispose a resource should call its *Dispose* method. The contract for *Dispose* says that it has to release all the resources that it or its superclasses own. Calls to *Dispose* should be idempotent. For Spec#, we also require that *Dispose* not throw any checked exception.

The .NET Framework also supports finalizers (written as destructors in C#). They are run during garbage collection. Finalizers should be used only as a safeguard to clean up resources in case the *Dispose* method was not called. Of course, finalizers are not allowed to throw any checked exception. The finalizer of an object shouldn't be run if the object's *Dispose* method already has been called; to that end, *Dispose* should call the *GC.SuppressFinalize* method for the object it is disposing [19].

An approximation of the methodology in the previous two paragraphs is enforced by the lint-like program checker FxCop [8]. Because FxCop performs a modular analysis, it doesn't have information about all exceptions that may be thrown by a method in C#. With Spec#, a tool like FxCop can produce more targeted warnings, for example by homing in on the execution paths resulting from checked exceptions and ignoring any unchecked exceptions.

The C# programming language already provides some help in the form of a **using** statement to make calls to the *Dispose* method automatic. The **using** statement obtains one or more resources, executes the given statements, and then calls *Dispose* on the resources. The following code example creates, uses, and cleans up an instance of a *Resource* class:

```
public void ResourceUsage() {
  using (Resource r = new Resource()) {
    r.DoSomething();
  }
}
```

The use of the **using** statement, however, is limited to blocks where resources don't escape. But suppose that you want to store a new resource in a field or that you want to return the resource. Then, it might happen that a checked exception is thrown after the allocation of the resource and before the assignment to the field or return of the method. In this case, a tool like FxCop can enforce that there is an enclosing **try-catch** statement whose handler frees the resource.

We use a different strategy to deal with unchecked exceptions. Unchecked exceptions typically propagate to a handler in an outermost tier of the application. A good discipline for these catch-all exception handlers seems to be to force a garbage collection. The garbage collection then triggers a call to the finalizers of unreachable objects, which in turn triggers calls to *Dispose*.

6. Example

In this section, we show how a specification of the $ArrayList.BinarySearch$ method can be written in Spec#. Figure 0 shows how MSDN [19] specifies this method today.

In Spec#, the parameter validation rules can be made explicit, as shown in Figure 1. The first three **requires** clauses say exactly what the MSDN documentation says, namely that $index$ and $count$, which together indicate the range to be searched (as offset plus length), must represent a valid subrange of the $ArrayList$. The fourth **requires** clause says that if the $comparer$ is null, then either $value$ or all of the elements of the $ArrayList$ implement the $IComparable$ interface. (The operator $==>$ denotes logical implication; $Forall$ is the universal quantifier; the range expression $\{s : t\}$ denotes the half-open interval from s up to, but not including, t.) Finally, the last **requires** clause says that if the $comparer$ is null but $value$ is not null, then the types of all elements must be assignable to the type of $value$ or the other way around. In fact, this already points out an ambiguity in the existing documentation, which uses the wording "same" type, but that's not what is intended.

Note that the documentation for $BinarySearch$ does not say anything about the array being sorted, though it is questionable whether calling $BinarySearch$ on an unsorted array makes any sense. For a full specification, one should also add that $BinarySearch$ doesn't have side effects.

7. Discussion

With consideration for an existing C# coding pattern, Spec# also provides a more complicated form of the **otherwise** clause, wherein any expression evaluating to an exception can be used. This provides the freedom to pass specific arguments to an exception constructor, for example. However, we recommend against using this more complicated form, because the additional effort required in using it seems to far outweigh the benefits.

For example, the .NET Framework currently includes code of the form shown in Figure 2. Note the elaborate arguments passed to the exception constructor. Not only is there a risk that arguments like these contain errors, but the benefits of having included them seem feeble. The exception indicates a client failure, which is a program error. How does knowing the name of the invalid parameter or the description of the exception thrown (translated into the local language via a resource string) help recover from a program error? It does not help any **try-catch** handler, nor does it help the user or product support staff. If the client failure is to be diagnosed, one needs a stack trace that gives enough context. And if one has the stack trace, then the name of the parameter provides no additional information.

8. Related Work

Various views have been put forth about the role of exceptions in programs and the criteria for designing exception features in programming languages (*e.g.*, [9, 16]). Stroustrup defines *exception safety* in the standard C++ library in terms of the following three guarantees [21]:

- Basic guarantee: The basic invariants of the class are maintained and no resources (which in C++ includes memory) are leaked.
- Strong guarantee: In addition to the basic guarantee, the operation either succeeds or has no effect.
- No-throw guarantee: In addition to the basic guarantee, no exception is thrown.

All operations of the standard C++ library are to adhere to the basic guarantee, key operations are to adhere to the strong guarantee, and a few simple operations may adhere to the no-throw guarantee. These guidelines can lead to more predictable behavior, but they are not enforced by the language. The language also lacks built-in class invariants. The features of Spec# enable this exception-safety policy to be stated explicitly and checked. For checked exceptions, Spec# checks that invariants are maintained and leans on tools like FxCop to check that no resources are leaked, providing the basic guarantee. When the success of an operation depends only on the operation's input, use of Spec# **requires-otherwise** declarations provides the strong guarantee. More generally, the strong guarantee can be specified with postconditions and the **unchanged** construct (or using **modifies** clauses, not described in this paper). Lastly, an empty throws set trivially specifies the no-throw guarantee.

Exceptions are used for systematic error handling in many languages. CLU [17] was the first language to include in a procedure's signature the set of exceptions it may possibly throw. If the execution of a procedure's body results in the throwing of an exception not in the procedure's declared throws set, the exception is turned into a special exception $Failure$. A similar design was later used in Modula-3 [20].

Java [10] evolves this design in three ways. First, exceptions in Java are objects whose types are rooted at a language-defined exception class, so a method is allowed to throw any exception whose type derives from a class named in the method's throws set. Second, through designated exception classes, Java makes the distinction between checked and unchecked exceptions, the latter of which are implicitly part of any throws set. Third, the check that a method

The $ArrayList.BinarySearch$ method searches a section of the sorted $ArrayList$ for an element using the specified comparer and returns the zero-based index of the element.

public virtual
int $BinarySearch$(**int** $index$, **int** $count$, **object** $value$, $IComparer$ $comparer$);

Parameters

- $index$ The zero-based starting index of the range to search.
- $count$ The length of the range to search.
- $value$ The object to locate. The value can be a null reference.
- $comparer$ The $IComparer$ implementation to use when comparing elements. -or- A null reference to use the default comparer that is the $IComparable$ implementation of each element.

Return Value
The zero-based index of $value$ in the sorted $ArrayList$, if $value$ is found; otherwise, a negative number, which is the bitwise complement of the index of the next element that is larger than $value$ or, if there is no larger element, the bitwise complement of $count$.

Exceptions

- $ArgumentException$: $index$ and $count$ do not denote a valid range in the $ArrayList$. -or- $comparer$ is a null reference and neither $value$ nor the elements of $ArrayList$ implement the $IComparable$ interface.
- $InvalidOperationException$: $comparer$ is a null reference and $value$ is not of the same type as the elements of the $ArrayList$.
- $ArgumentOutOfRangeException$: $index$ is less than zero. -or- $count$ is less than zero.

Remarks
. . . If $comparer$ is a null reference, the comparison is done using the $IComparable$ implementation provided by the element itself or by the specified $value$. . . .
Comparing a null reference with any type is allowed and does not generate an exception when using $IComparable$. . . .

Figure 0. MSDN documentation for $BinarySearch$.

body does not result in the throwing of a checked exception not covered by the throws set is done statically, through syntactic means. Exceptions in C# are also classes, but, alas, all exception classes are unchecked. Spec# retrofits the C# design with the addition of checked exceptions and throws sets. Since Spec# incorporates contracts, each **throws** clause can also specify a postcondition, like the exceptional postconditions in, for example, JML [13].

Two of the earliest programming languages with built-in pre- and postconditions are Gypsy [0] and Euclid [12]. Euclid compiles these specifications into run-time checks; if a check fails, the program is terminated. Eiffel [18] incorporates specifications into an object-oriented programming language, and its libraries come equipped with specifications. The specifications are enforced at run time. The Java Modeling Language (JML) [13] adds specifications to the object-oriented programming language Java [10]. The JML compiler inserts run-time checks for these specifications, and various other JML tools attempt to check the specifications statically [3]. New to Spec# are the **otherwise** clauses that allow the stylized .NET Framework documentation to be encoded as preconditions. Since the .NET Framework documentation is similar to that of the Java libraries, we expect that **otherwise** clauses could be useful for Java specifications as well.

Languages that build in specifications often provide some forms of invariants. For example, Eiffel provides class invariants that are intended to hold for each instance of the class whenever no method is active on the instance. New in Spec# is the interplay between such object-invariant checking and exceptions, and in particular checking object invariants whenever an **expose** block is exited normally or exited with a checked exception.

Drawing from programming experience with, for example, the Taos operating system, Levin and Wobber have developed a number of guidelines for using exceptions [15]. They suggest that exceptions resulting from fatal errors, which correspond to our unchecked exceptions, be handled only in specially designated *backstops* that catch all exceptions. Their recommendation is that programmers always know where these backstops are, for example, in an outer-

```
public virtual
int BinarySearch(int index, int count, object val, IComparer comparer)
  requires 0 <= index otherwise ArgumentOutOfRangeException ;
  requires 0 <= count otherwise ArgumentOutOfRangeException ;
  requires count <= this.Count − index otherwise ArgumentException ;
  requires comparer == null ==>
    value is IComparable || Forall{int i in{index : index + count}; this[i] is IComparable}
    otherwise ArgumentException ;
  requires comparer == null && value != null ==>
    Forall{int i in{index : index + count}; this[i] != null ==>
              this[i].GetType().IsAssignable(value.GetType()) ||
              value.GetType().IsAssignable(this[i].GetType())}
    otherwise InvalidOperationException ;
```

Figure 1. Spec# contract for $BinarySearch$**.**

```
public virtual
int BinarySearch(int index, int count, object val, IComparer comparer) {
  if (index < 0 || count < 0) {
    throw new ArgumentOutOfRangeException((index < 0 ? "index" : "count"),
        Environment.GetResourceString("ArgumentOutOfRange_NeedNonNegNum")) ;
  }
  if (_size − index < count) {
    throw new ArgumentException(Environment.GetResourceString("Argument_InvalidOffLen")) ;
  }
  ...
```

Figure 2. Today's parameter validation of $BinarySearch$ **in C#.**

most tier of each thread or in a window-system event handler. Otherwise, Levin and Wobber warn, "there will be an irresistible tendency to catch fatal exceptions deep in an implementation where they can't be handled in a uniform manner". The Spec# exception methodology seems consistent with these recommendations: the compiler forces callers to deal with the admissible failures documented in the throws sets of callees, whereas the (in Spec# undocumented) unchecked exceptions most easily are left unhandled until backstops.

The semantics of exceptions has been formalized, for example as *weakest preconditions* by Cristian [5]. Tools that use weakest preconditions and a mechanical theorem prover to reason statically about programs with exceptions include ESC/Modula-3 [6] and ESC/Java [7, 14]. The LOOP program verifier [22, 11] uses a Hoare logic to reason statically about programs with exceptions, applying theorem-prover tactics based on weakest preconditions. There are also other tools that reason statically about programs with exceptions, including the software model checkers Bandera [4] and Java PathFinder-2 [23].

In our own work, we are developing Spec# hand in hand with the automatic program verifier Boogie [2], which uses automatic program-verification techniques like those in ESC [6, 7]. For instance, Boogie checks at compile-time that a method's preconditions hold at each call site. If Boogie can verify that the precondition always holds, then there is an opportunity for the compiler to suppress the corresponding run-time check.

9. Conclusion

Spec# includes features that allow programs to take advantage of the good side of exceptions—clean and systematic error handling—while taking measures to reduce the possibilities of exceptions becoming a problem in their own right.

Inspired by Goodenough [9], Spec# classifies failures

into client failures, admissible failures, and observed program errors. Following Java [10], Spec# also distinguishes between checked and unchecked exceptions. Unchecked exceptions are used to signal client failures and observed program errors, whereas checked exceptions are used to signal admissible failures.

Our **requires-otherwise** declarations allow us to retrofit C# with precondition declarations that match current programming practice used with the .NET Framework.

We have also described a discipline for the interplay between declared object invariants and checked exceptions. The discipline is completely enforced by a combination of compile-time and run-time checks. The Spec# exception discipline also seems to lend itself to better checking of resource usage and controlled disposal.

Our grand vision for Spec# is to develop a programming system in which one can verify partial specifications for a modern object-oriented language [2]. Exception safety is a prerequisite for making strides toward this ambitious goal.

Acknowledgments

We would like to thank the rest of the Boogie team (Mike Barnett, Rob DeLine, and Manuel Fähndrich) and the FxCop team (Michael Fanning *et al.*) for discussions. A big thanks goes to Mike Barnett and Herman Venter, who have been invaluable in the implementation of the Spec# compiler and development environment.

References

[0] A. L. Ambler, D. I. Good, J. C. Browne, W. F. Burger, R. M. Cohen, C. G. Hoch, and R. E. Wells. GYPSY: A language for specification and implementation of verifiable programs. *SIGPLAN Notices*, 12(3):1–10, Mar. 1977.

[1] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. Manuscript KRML 136, Microsoft Research, May 2004. Submitted.

[3] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS '03)*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier, June 2003. Revised version to appear in International Journal on Software Tools for Technology Transfer.

[4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448. ACM, June 2000.

[5] F. Cristian. Correct and robust programs. *IEEE Transactions on Software Engineering*, 10:163–174, 1984.

[6] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, Dec. 1998.

[7] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 of *SIGPLAN Notices*, pages 234–245. ACM, May 2002.

[8] FxCop team page. `http://www.gotdotnet.com/team/fxcop/`, 2004.

[9] J. B. Goodenough. Structured exception handling. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages*, pages 204–224. ACM, Jan. 1975.

[10] J. Gosling, B. Joy, and G. Steele. *The Java™ Language Specification*. Addison-Wesley, 1996.

[11] B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hußmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001*, volume 2029 of *Lecture Notes in Computer Science*, pages 284–299. Springer, Apr. 2001.

[12] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language Euclid. Technical Report CSL-81-12, Xerox PARC, Oct. 1981. An earlier version of this report appeared in *SIGPLAN Notices*, vol. 12, no. 2, ACM, Feb. 1977.

[13] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06f, Iowa State University, Department of Computer Science, July 1999.

[14] K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999. Also available as Technical Note 1999-002, Compaq Systems Research Center.

[15] R. Levin and T. Wobber. Some (experience-based) guidelines on using exceptions. Personal communication, 2004.

[16] B. Liskov. A history of CLU. In *History of Programming Languages Conference*, volume 28, number 3 of *SIGPLAN Notices*, pages 133–147. ACM, Mar. 1993.

[17] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.

[18] B. Meyer. *Object-oriented software construction*. Series in Computer Science. Prentice-Hall International, 1988.

[19] MSDN. `http://msdn.microsoft.com`.

[20] G. Nelson, editor. *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice-Hall, 1991.

[21] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, special edition, 2000.

[22] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer, Apr. 2001.

[23] W. Visser, K. Havelund, G. P. Brat, and S. Park. Model checking programs. In *The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, pages 3–12. IEEE Computer Society, Sept. 2000.

[24] M. Williams. *Microsoft Visual C# .NET*. Microsoft Press, 2002.