Adapting C++ Exception Handling to an Extended COM Exception Model

Bjørn Egil Hansen
DNV AS, DT 990 Risk Management Software
Palace House, 3 Cathedral Street,
London SE1 9DE, UK
Bjorn.Egil.Hansen@dnv.com

Henrik Fredholm
Computas AS
Vollsveien 9, P.O.Box 482,
1327 Lysaker, Norway
hf@computas.no

This paper describes how correctness and robustness of component-based systems can be improved by categorising exceptions by component state and cause, and handling them accordingly. Further, it is shown how this model is supported in C++ in a COM-based environment, also simplifying the code for exception detection, signalling, and handling.

1. INTRODUCTION

The reliability of a software system is to a high degree determined by the reliability of the software components comprising the system [KriMat97]. For a software component to be reliable it must be able recover various exceptions or failures that may arise at run-time, i.e. it must be fault tolerant. Code for exception detection, handling, and signalling often amounts to a substantial portion of the code. This is especially the case in a multi-language environment, mixing different exception models, and requiring transformation between different exception representations.

To allow flexible system configuration and system evolution, e.g. making changes to a component, replacing a component with another, or adding new components, each of the components should have minimum dependency on the other components. Loose coupling between components is also important with respect to exception handling and the mechanisms used. As argued in [Cri95], a termination mechanism mixes well with the information hiding principles underlying data abstraction, and consequently also component based systems. The focus should be on the local transition and consistent local state of the component and on exceptions that may arise during a state transition. It is the responsibility of the developer of the component to detect, handle, and signal exceptions based on the state of the component, the arguments of the call, and the results from lower-level components.

In software development project there are always conflicts between different requirements, and the development team has to make trade-offs between time-to-market, functional requirements, and non-functional requirements, like reliability. The resources used on reliability are often scarce, hence it is important to concentrate the effort on those exception situations that contributes to the overall reliability of the system. Whenever reliability is sacrificed for other requirements, the failure situations should be clearly distinguishable from correct behaviour and easily traceable in the code. If the system reliability turns out to be unacceptable it should be possible to increase the overall reliability by improving individual components of the system, i.e. taking control over more of the failure situations.

This paper is based on work done in the BRIX team in DNV IT Solutions. The main purpose of this group is to provide the various application development teams with common solutions to software technical needs. Most BRIX solutions are based on Microsoft's COM technology and C++ as implementation language.

In section 2 the differences between C++ exception handling and COM exception handling are briefly outlined. Section 3 describes the BRIX language independent exception model, which is based on the COM exception model. In section four we present how the BRIX exception model is supported at C++ level. Finally, in section 4 we draw some conclusions.

2. COM EXCEPTION HANDLING IN C++

COM supports signalling of exceptions [Box98], but the mechanisms used are different from the throw and catch in C++ [ElStro90]. To signal an exception in COM, three different mechanisms are used in combination:

- By convention all interface methods on a COM objects should return a status (an HRESULT), indicating success or exception/failure of the method invocation.
- To pass extra information to the client, COM provides to API functions:
  SetErrorInfo: Used by the COM object (signaller) to pass an exception object containing the extra information to the client.
  GetErrorInfo: Used by the client to obtain the exception object.

- In addition, the COM object must implement the ISupportErrorInfo interface to indicate which interfaces support exceptions. This interface should be used by the client to determine whether or not the result of the GetErrorInfo is reliable.

This obviously increases the burden on a C++ programmer. When implementing a COM object in C++, the programmer must be careful to catch all C++ exceptions and convert them into HRESULTs, possibly augmented with extra information in an exception object.

To detect and handle exceptions from a COM object in a C++ client, the result of each COM-call must be checked specifically by looking at the returned HRESULT. In a naïve implementation of a C++ client this will typically result in an unreadable mix of HRESULT checking of the COM calls and catching of C++ exceptions of internal C++ calls.

There is also a mismatch in the exception models of COM and C++. While we in C++ may specialise exceptions in hierarchies, COM overloads the HRESULT. Thus, the meaning of an HRESULT code very likely will be different for different methods, and the code should not propagate unmodified up through the call stack.

When programming in Visual Basic or Java (on Microsoft's Java VM), the conversion between native exceptions and the HRESULT and the COM exception information object is done automatically by the respective run-time systems. Thus, in these environments the COM exception representation and signalling mechanism are integrated with native exception mechanisms, giving a more transparent programming model. In section 4 we will see how the BRIX Exception System contributes to integration with the C++ exception mechanisms.

3. BRIX EXCEPTION MODEL

A component-based system is comprised of components accessing each other through interfaces. An interface may be defined specifically for a concrete implementation, complete both with respect to standard and exceptional behaviour. However, in COM it is common to have general interfaces allowing a variety of implementations. For a specific implementation there may be resource constraints not anticipated on the specification level, resulting in exceptions outside the specified exceptions.

One of the main goals of the BRIX Exception System is to provide the component developer with mechanisms of detection, handling, and signalling of specified exceptions, as well as mechanisms for detection and signalling of unspecified exceptions. To facilitate this, when focusing on the state transition within one component, we categorise exceptions/failures of lower-level components according to two orthogonal criteria:

- State of the lower-level component after the exception/failure has occurred:
  - **Controlled exception:** The component is in the same consistent state as before the method call.
  - **Uncontrolled exception:** The component is in an undefined and possibly inconsistent state. This corresponds to the notion of failure exceptions in [Cri95].
- Cause of the exception:
  - **Operational exception:** The cause of the exception is outside the control of the component, typically when allocating resources (e.g. memory, files, or network connections) or validating input.
  - **Implementation exception:** The cause of the error is due to faults in program code.

In the table below we assume having a higher-level component A calling a method on component B, being our focus, and B calls a method on lower-level component C. We discuss the four categories of possible exceptional situations, and propose a recommend action and an alternative action as a guide to where to concentrate the exception handling effort.

	Operational	**Implementation**
Controlled	*Controlled Operational Exception*: The cause is an operational exception the lower level component C. C has recovered to the same state as before the call. *Recommended action*: If it is a specified exception, mask the exception if possible. Otherwise, recover state and signal a Controlled Operational Exception to higher level component A.	*Controlled Implementation Exception*: The lower-level component C has rejected the call because of violation of precondition, i.e. the exception is caused either by improper use by component B, too strict implementation of the precondition validation, or incomplete interface specification. C has recovered to the same state as before the call. *Recommended action*: Use default mechanism that signals an Uncontrolled Implementation Exception to the higher-level component A. Fix bug at appropriate level.

	Operational	**Implementation**
	Alternative action: Use default mechanism that signals an Uncontrolled Operational Exception to the higher-level component A.	*Alternative action*: If the cause is an incomplete specification or a too strict precondition implementation in C, and neither of these can be changed, component B should be rewritten to avoid the problem.
Uncontrolled	*Uncontrolled Operational Failure*: The failure is due to an operational exception in the lower-level component C which C was not prepared to handle, and C has not been able to recover to a consistent state. *Recommended action*: Use default mechanism that signals an Uncontrolled Operational Exception to the higher-level component A. Make C more robust by taking control of the exception. *Alternative action*: If component C cannot be changed, component B or high-level component A should be rewritten to avoid the problem, or more run-time resources should be made available to avoid the problem	*Uncontrolled Implementation Failure*: The failure is due to an unanticipated exception occurrence in the lower-level component C, resulting in a possibly inconsistent state. Hence, invalidating the invariant or post-condition. *Recommended action*: Use default mechanism that signals an Uncontrolled Implementation Exception to the higher-level component A. Fix fault in C. *Alternative action*: If component C cannot be changed, component B or high-level component A should be rewritten to avoid the problem.

Table 1: Exception categories in BRIX

In the table we see that controlled operational exceptions are the only cases where explicit exception handling is recommended to increase the overall robustness of the system. Using the default BRIX exception mechanisms in this case will lower the robustness of the system, as the exception is transformed to an uncontrolled exception when propagated and most likely will cause the system to abort.

For the other cases it is generally difficult and time-consuming to handle them in way that increases the robustness of the system. Instead it is recommended to use the default mechanisms to signal an uncontrolled exception to the higher-level component. The cause of the failure should be fixed off-line if required to achieve long term robustness.

Our goal is not to write totally correct programs, but to write partially correct programs [Cri95], that is, programs that either produce the specified result (normal or exceptional) or a confined failure (unspecified exceptions) with respect to a complete specification. Partially correct programs are safe, in the sense that there are no unanticipated inputs (incomplete specification) and no unconfined failures (apparently correct results but actually erroneous with respect to specification). Thus, they will either produce the specified results or obvious failures.

A component may contain implementation faults. The BRIX Exception System contributes to the fault tolerance of the system by distinguishing between uncontrolled and controlled failures. Uncontrolled failures should result in abortion, while controlled failures may be tolerated, because the system is still in a well-defined state.

4. BRIX SUPPORT FOR C++ EXCEPTION HANDLING IN COM

The exception handling code itself may be a significant source of errors. It is difficult to test and should be kept as simple as possible. In the following we will see how the BRIX exception mechanisms supports detection, handling, and signalling of exceptions according to the categorisation of exceptions in the previous section, in a programming environment as described in section 2. The examples below are based on [Cri95].

Firstly, to prevent C++ exceptions from propagating out of the component, two macros are used to enclose the code establishing default try and catch blocks. The macros take care of converting from C++ exceptions to HRESULT. Any exceptions not detected in the program code will be caught and signalled as an uncontrolled implementation exception. Figure 1 shows an implementation for computing factorials. Assuming *n* is specified to be ≥0, this implementation may result in either a)

```
HRESULT Calc::Fact(int n, int* pf)
{ BX_ENTER_COM(Calc,Fact)
  int k=0,m=1;
  while (k<n) { k++; m*=k; }
  *pf = m;
  BX_RETURN_COM
}
```

Figure 1: Computing factorial

an unconfined failure for values of *n*<0 by returning 1 (erroneous result), or b) a confined failure if *n!* is larger than the maximum int value returned as an uncontrolled implementation exception.

We can increase the correctness of this program by detecting and signalling a) properly. There is no support for automatic checking of preconditions like in Eiffel [Mey88]. However, this can be coded explicitly, as shown in figure 2. If a precondition fails this results in a controlled implementation exception (E_BX_PRECONDITION), thus we have eliminated a) and made the program partially correct.

```
HRESULT Calc::Fact(int n, int* pf)
{ BX_ENTER_COM(Calc,Fact)
  BX_PRECOND(n>=0)
  …
```

Figure 2: No unconfined failures

Assume the specification states that E_OVERFLOW should be returned if *n!* is too large. To take control over this situation we must add explicit detection and handling of the overflow. Figure 3 illustrates this, where we have a try-catch block to detect the overflow. In the catch block we use *bxhr* which is defined by BX_ENTER_COM. This is an instance of BxHResult, which offers a range of functionality for detecting, signalling and handling exceptions. In this example RaiseError throws an exception containing the specified HRESULT value. This exception is caught by BX_RETURN_COM and the E_OVERFLOW is returned to the client. Thus, we have made the code more correct by handling this exception as specified. Also, we have contributed to the overall robustness of the system by eliminating an uncontrolled exception.

```
…
try {
  int k=0,m=1;
  while (k<n) { k++; m*=k; }
  *pf = m;
} catch(…)
{ bxhr.RaiseError(E_OVERFLOW,…);}
…
```

Figure 3:Controlled overflow as specified

In the examples above we have seen how exceptions are detected and signalled using different mechanisms, e.g. precondition checking, explicit raising of exception, and by use of default detection and signalling. All is resulting in a corresponding HRESULT being returned to the client.

To detect, handle, and signal exceptions from a COM component, various mechanisms can be used resulting in different degrees of robustness. BxHResult redefines some of the assignment operators to provide short hand notation for detection and signalling of exceptions. In figure 4 we have a class to represent a vector of 10 ints and defining a method to compute the factorial of each of the values and store them in the same vector. Using the |= operator we say that any exceptions returned from the method on the lower-level component will result in an uncontrolled exception in this component and a corresponding C++ will be thrown. If the Fact method in figure 4 returns an exception, we may have an intermediate state where some values have been changed to their factorials where as others are unchanged, i.e. the component will be left in an uncontrolled and erroneous state.

```
class Vector
{  int v[10];
   HRESULT Factorials();
   …
}
HRESULT Vector:: Factorials ()
{ BX_ENTER_COM(Vector, Factorials)
  for (int i=0; i<10; i++)
  { bxhr|=pCalc->Fact(v[i],&v[i]);
  }
  BX_RETURN_COM
}
```

Figure 4: Uncontrolled exception

To avoid the uncontrolled exception we must be able to recover the initial state in case of exceptions. Figure 5 shows how this can be done in this simple case by keeping the new values in a temporary array until all the calculations have been done successfully. Thus, if the Fact method signals an exception, our state is still consistent and we should just signal a controlled exception to our caller. This is done by using the &= operator instead of the |= operator, which will transform any controlled exceptions to controlled implementation exceptions. However, if the Fact method signals an uncontrolled exception, as in figure 1 and 2, an uncontrolled implementation exception will be signalled from this component. By focusing on the local transition and on keeping a local consistent state, we reduce the number of uncontrolled exception propagating through the system and the likelihood of abortions.

```
  …
  int u[10];
  for (int i=0; i<10; i++)
  { bxhr&=pCalc->Fact(v[i],&u[i]);
  }
  v = u;
  …
```

Figure 5: Controlled exception

Further, assume that the Factorials specification also states that an E_OVERFLOW exception should be signalled in case of overflows. With respect to this specification, the implementation in figure 5 is partial correct and will result in a confined failure in case of overflow. In cases where the state is still consistent and the specified exceptions from the method on lower-level component is the same as for this method, we can use another operator %= to signal the same exception to the higher-level component, as illustrated in figure 6. Hence, the implementation will be correct with respect to the overflow exception.

```
  …
  int u[10];
  for (int i=0; i<10; i++)
  { bxhr%=pCalc->Fact(v[i],&u[i]);
  }
  v = u;
  …
```

Figure 6: Propagation of exception

The examples in figure 4 to 6 show compact detection and signalling of uncontrolled, controlled, and specified exceptions using the redefined operators |=, &=, and %=, respectively, and how we gradually can increase the robustness and correctness of the system by making the local transitions more robust and correct.

In other situations it might be required that the exceptions are masked or at least handled more specifically. Figure 7 shows one way this can be done using the redefined ^= operator. Instead of throwing an exception, the results from the call will be kept in *bxhr* for subsequent handling of the exception. In the case of a time-out, the exception is recovered and the call is retried once. If this fails the exception is propagated using the &= operator. If the first exception was not a time-out exception, the same exception is signalled to the caller. Exceptions can also be handled in the normal C++ way by enclosing a sequence of statements with a try and catch block.

```
…
bxhr ^= pSrv->Submit(data);
if (bxhr==E_TIMEOUT)
{ // Retry once
  bxhr.ErrorRecovered();
  bxhr &= pSrv->Submit(data);
}
else if (bxhr!=S_OK)
  bxhr.RaiseError(…);
…
```

Figure 7: Masking/handling exceptions

Other features of the BRIX exception system which have not been emphasised here include checking of intermediate and final states by use of assertions and post-conditions, possibly resulting in uncontrolled implementation failures. Also there is rich support for logging of exceptional events, which may serve various purposes: simplifies debugging of the system both during development and operation, supports operator/end-user in identifying possible lack of resources (causing operational exceptions).

5. CONCLUSION

The BRIX Exception System contributes to correctness and robustness in several ways:

- By distinguishing controlled and uncontrolled exceptions, we believe to achieve fault-tolerance both with respect to specification and implementation faults. Also we avoid unconfined errors by having an explicit notion of and support for uncontrolled exceptions. Hence, it will be easier to achieve partial correctness [Cri95], assuming that the specification is complete.
- By taking control over more failure situations at the component level, we may increase the overall robustness of a system by upgrading or replacing individual components.
- By focusing on the local state and transition, providing a programmed exception handling style using pre/post-conditions and assertions for state validation and mechanisms for tight control of the results from lower-level components. Contrary to [Mey88] all pre/post condition validation has to be done explicitly. However, when handling complex states and transitions, expressing pre/post-condition may be difficult resulting in possible unconfined failures [Cir95], and intermediate checks may be preferable.
- By virtually forcing the developer to decide on the local implications of exceptions occurring for each line of code. Also the BRIX exceptions mechanisms make it simple to implement those decisions by providing default propagation mechanisms and overloaded operators for exception checking.

6. ACKNOWLEDGEMENT

Many thanks to Are F. Tjønn and Johannes Hermanrud who supported both the development and the writing of this paper. Thanks also to Daniel Vatier and Egil P. Andersen for reviewing the paper and to rest of the BRIX team for providing an inspiring working environment.

7. REFERENCES

[Box98] Box, Don, *Essential COM*, Addison Wesley, 1998.

[Cri95] Cristian, Flaviu, “Exception Handling and Tolerance of Software Faults”, chapter 4, pages 81-107, in *Software Fault Tolerance*, Lyu, M. (ed.), Wiley, 1995.

[ElStro90] Ellis, Margaret A and Stroustrup Bjarne, *The Annotated C++ Reference Manual*, Addison Wesley, 1990.

[KriMat97] Krishnamurthy, S and Mathur, A.P., "On the Estimation of Reliability of a Software System Using Reliabilities of its Components", *Eighth International Symposium on Software Reliability Engineering (ISSRE '97)*, November 2-5, 1997 Albuquerque, US

[Mey88] Meyer, Bertrand, *Object-Oriented Software Construction*, Prentice Hall, 1988.